**INTERIM TECHNICAL REPORT**

# A SUPPORT ARCHITECTURE FOR RELIABLE DISTRIBUTED COMPUTING SYSTEMS

**Prepared for**

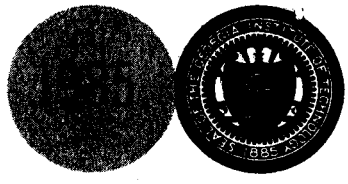
**National Aeronautics and Space Administration**
**Langley Research Center**

**Under**

**Grant No. NAG-1-430**

**Final Report for Period November 9, 1983 to December 3, 1985**

**GEORGIA INSTITUTE OF TECHNOLOGY**
**A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA**
**SCHOOL OF INFORMATION AND COMPUTER SCIENCE**
**ATLANTA, GEORGIA 30332**



# A Support Architecture for Reliable Distributed Computing Systems

Interim Technical Report
November 9, 1983 - December 3 1985

From:

Georgia Tech Research Corporation
Atlanta, Georgia 30332

To:

National Aeronautics and Space Administration
Langley Research Center

Grant No.: NAG-1-430

Endorsements:

Partha Dasgupta for Martin McKendry

Martin S. McKendry
Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572

# A Support Architecture for Reliable Distributed Systems

## 1. Introduction

The Clouds Project at Georgia Tech is conducting research aimed at building a reliable distributed operating system. The primary objectives of the Clouds operating system are:

1] The operating systems will be distributed over several sites. The sites will have a fair degree of autonomity. Yet the distributed system should work as an integrated system. Thus the system should support location independency for data, users and processes.

2] Reliability is a key requirement. Large distributed systems use significant number of hardware components and communication interfaces, all of which are prone to failures. The system should be able to function normally even with several failed components.

3] The processing environment should guard against both hardware and software failures. The permanent data stored in the system should be consistent.

4] Distributed systems often have dynamic configurations. That is, newer hardware gets added, or faulty hardware is removed. The system function should not be hampered by such maintainance chores. Thus the system should be dynamically reconfigurable.

5] The system should be capable of monitoring itself. This encompasses hardware monitoring for keeping track of hardware failures as well as monitoring key software resources (for example daemons, network servers, and so on.) On detection of failure the system should be able to self-heal (restart daemons) or self-reconfigure (eliminate faulty sites).

6] The users should be shielded from both the configuration of the system (site independence) as well as its failure modes. For example, if the site a user is connected to fails, he should be transferred to an active site transparently.

7] Many of the above functions can be implemented on conventional systems, but would make the system extremely slow. Thus efficiency is an important design criteria.

The above requirements can be handled by a distributed system and have been designed into the Clouds operating system. Most of the functions have been designed into the kernel of the system, without making the kernel too complex, bulky or inefficient. The design philosophies adopted for the Clouds operating system are:

1] An object-based, passive system, paradigm is used as the basic architecture. All system functions, data, user programs and resources are encapsulated as passive objects. The objects can be invoked at appropriate entry points by processes.

2] The objects in Clouds represent nearly everything the system has to offer. The site independence philosophy is implemented by making the object name space (system names) flat and site independent. When a process on any machine invokes an object located anywhere, no site names are used. Hence the location of any particular object is unknown to a process.

3] Reliability is achieved through two techniques. One of them is the action and recovery concept. The action mechanisms are supported at the kernel level. Actions are atomic units of work. Any unfinished or failed action is recovered and has no effect until it completes. The recovery mechanisms are supported inside every object an action touches.

4] Reliability is further extended by the self monitoring and self reconfiguration subsystems. This is a set of monitoring processes that use "probes" to keep track of all key system resources, both hardware and software. On detection of failed or flaky components, the monitoring system invokes the reconfiguration system which rectifies or eliminates (if possible) the faulty components, and initiates recovery of affected actions. The monitoring and reconfiguration subsystems are also monitored by the monitoring system.

5] The consistency requirements of the data are handled by the recovery mechanisms and by concurrency control techniques. The concurrency control is handled by synchronization paradigms that are an integral part of the object handling primitives. The synchronization of processes executing in an object is handled automatically by semaphores that are a part of the object. This gives rise to a two-phase locking algorithm that is supported by the kernel as a default. The object programmer has the choice of overriding these controls and use custom built concurrency control, depending upon the application. It is also possible to turn off the default recovery and commit strategies.

6] Efficiency has been of concern. The object invocation, recovery and synchronization are handled by the kernel. It turns out that these can be done at the kernel level without much overhead. Since the entire Clouds design is primarily based on object manipulations, invocation and synchronization will be the most used operations. Implementing them at the kernel level will result in an efficient system.

7] The site independence at the user level is handled in part by using intelligent terminals. The user terminals are not hardwired into any machine or site, but are on an ethernet, accessible by any site. Each user session is, of course, handled by one

particular site, but any failure causing the controlling site to be unaccessible causes the user to be transferred to another site. This is handled cooperatively by the user terminal and the other sites. Thus the user terminals are actually intelligent microprocesser systems on the Clouds ethernet. In addition to cooperation with the Clouds network, the user terminals run "Bubbles", a multiwindowing, user-friendly interface to Clouds.

## 2. Progress Report

The following is a brief report of the current status of the implementation of Clouds.

### 2.1. Equipment

The test equipment for implementing Clouds was funded by the National Science Foundation and has arrived. Three VAX/750 computers interconnected by a ethernet was installed in November 1984. They have been heavily used to develop the Clouds kernel and allied software described later. Three IBM-PC/XT computers, arrived in July 1985 and are being used to develop the intelligent terminal interface to Clouds. One IBM-PC/AT is scheduled to arrive soon and will be used as the primary development system for Bubbles and the ethernet handling code for the terminal interface.

### 2.2. Clouds Kernel Design

The kernel design has been through several design phases and is nearly complete. The design effort has produced a cohesive set of implementation guides to the entire Clouds kernel.

The current designs are based on assumptions about efficiency and ease of implementation that seem to be intuitively clear at this time. We may have to reiterate some design decisions and modify some strategies after more hard data is available from the implementation experience.

2.2.1. Kernel Implementation The Clouds kernel consists of several major subsystems: the object manager, which is responsible for mapping objects into virtual memory and invoking object operations (including the initiation of remote object requests); the process manager, which controls the slave process pool available on each node in a Clouds system and also supplies primitives for synchronization and process dispatch; the storage manager, which provides permanent storage for object data and paging storage for the virtual memory system; the communications manager, which is responsible for controlling inter-machine communications (currently via the ethernet); and the action manager, which is responsible for managing action events. The subsystems have been in various stages of completion, but have recently been integrated so a fairly complete, running version of the Clouds kernel exists. The current status of each of the subsystems is described below. For more details, see the attached technical reports ([Pitt85] and [Spaf84]).

2.2.1.1. Object Management The object management subsystem is almost completely coded and substantial sections have been tested. In particular, a primitive remote procedure call (RPC) mechanism has been implemented and tested. The complete RPC mechanism implementation awaits the implementation of the action management subsystem. Object mapping is implemented and is being tested with the virtual memory support provided by the storage management subsystem. Page fault handling will be done in tandem by the object manager and the storage manager. After the original fault is caught by the system, the object manager determines where the fault occurred (in a client object, in system space, or in a per process space) and makes a storage management call passing that information. Storage management is then responsible for selecting a physical page (through a call provided by the virtual memory system) and filling that physical page from the proper block on secondary storage.

The object invocation routines are being refined and implemented. Object and storage management are required to interact heavily to process an object operation call. Object management must first determine that the operation call is valid. It then initiates a search (possibly a network wide search) for the object. The storage management subsystem is responsible for determining whether the object exists on its local node and for activating the disk segment for the object if the object is found. Object management resumes control to initiate the operation call. The object management interface also provides the hooks necessary for the eventual presence of the action management subsystem.

Because of the cooperation required between object, storage, and action management, several iteration of the interface design were made before finally settling with the current design. It is felt that the current design meets all the requirements of the various subsystems involved.

The object manager and action manager normally supply certain special object operations, such as "create instance", "commit action", and "destroy action." Clouds programmers are able to reprogram these operations, so that in addition to performing necessary functions, the operations to the particular initializations, customized recovery operations, and cleanup that are specified. The object and action manager provide this support as part of the kernel interface, which can be accessed through the runtime system used by the Aeolus programming language.

2.2.1.2. Process Management The process manager is completely coded and tested. It provides a very rich set of synchronization primitives, which include semaphores, read/write locks, and general event mechanisms. Facilities for blocking with a time-out value are included. Code for the initialization of the slave process pool is running, as it that for dispatching processes. Slave processes are created at system initialization and are available for use as requests arrive. This accelerates the creation of processes for requests such as RPC's. The process management subsystem supports a primitive

round-robin scheduler with five priority levels.

2.2.1.3. Communication Management The communications management subsystem currently consists of the ethernet driver and associated software. This code has been tested and integrated into the Clouds kernel. The driver supports communications protocols not only for Clouds machines but also for machines running Unix 4.2bsd. Support for communication with Unix systems was implemented because it provides several possibilities supporting further development of the Clouds system. One such possibility is the development of a virtual disk for the Clouds kernel. Clouds kernel device requests to the storage manager could actually be handled by a disk running under a Unix system using the Clouds-Unix protocol on the Ethernet. This would provide either additional devices (very quickly, since the same interface at the Clouds end could be re-used) and also a facility for dumping status information for offline debugging. The communications subsystem recently was interfaced with the object management subsystem to provide a primitive working RPC mechanism.

2.2.1.4. Storage Management The storage management subsystem of the kernel consists of three classes of objects: devices, partitions, and segments. These object classes contain the structures and algorithms required to provide recoverable object data under the Clouds kernel. The subsystem is primarily concerned with the storage of on secondary storage devices, but is also necessarily involved in the management of virtual memory and action/object management.

There is a working device object (for the RL02 removable storage disk), that supports not only the conventional device operations (read and writes), but also provides a mechanism whereby the storage manager can insure that writes to devices performed by actions are done before the action completes. System failures will not catch object data in an inconsistent state. By having this support at such a low level, the storage manager relieves action management of some of its burden. In fact, the storage manager provides action management with a few simple calls that perform all the functions required to provide recoverable transfers of data to secondary storage.

The storage available on the RL02 device is not extensive (10 megabytes per pack) and the RL02 is not meant to be the primary drive for the Clouds system. However, it does provide a suitable testing device. A driver object is under development for another DEC drive, the RA81. This is a more sophisticated and larger drive (456 megabytes, fixed medium) than the RL02 and consequently the implementation of the RA81 object has been more complex than that for the RL01. The device object for this device in partially implemented and is being tested incrementally.

It should be noted that the design and development of the various devices discussed (and indeed those that will come later) have all been done using a standard interface to the Clouds system. This will allow us to bring new classes of devices onto the Clouds system with little difficulty. The only difference between the RL02 device driver

and the one being developed for the RA81 is that the RL02 object was written assuming only one such device existed (for simplicity). The RA81 object is being written for multiple drives per controller which will require some structural changes for the kernel, which have already been designed.

The device objects are being implemented with the idea that devices may be dynamically added to, removed from, and initialized on a running Clouds system. Currently, support is provided to allow operators to manually mount, unmount, or initialize devices, but further work could be done to automate this process.

The partition object is almost complete and running. Partitions can be created and removed from a device (dynamically, although support for this is not as clean as would be liked), read and write operations on partition blocks are available, storage may be allocated and deallocated from the partition, and the partition directory may add, remove, or locate items on the partition. The major component missing from the partition object is the partition activate call. This call brings certain partition structures into virtual memory and performs consistency checks on the partition data. Also the partition activation call initiates the action management cleanup that is performed by the storage manager. This processing is currently being integrated into the activation call.

The code for segment object is undergoing testing and final implementation. The segment object provides an interface to the storage management subsystem for the rest of the kernel. Primarily, the interface is through the abstraction of the segment type. The segment type is generally (though not always) simply a convenient alternate view of some client or system object (as a sequence of uninterpretted bytes). This allows the kernel to handle the various types of objects in a uniform manner. Coding necessary for handling segment level reads and writes and for handling page faults (in cooperation with object management) is complete and is in testing.

The segment object also provides the recoverability of object data. It provides the action management subsystem with a set of routines which transfer the data from virtual memory to secondary storage in a consistent manner. When invoked by the action management subsystem, the routines determine which parts of an object were modified by an action, and how to transfer the modified portions to secondary storage. The algorithms that do this are detailed in [Pitts85], along with an overall description of the storage manager. The algorithms described in the referenced report use the technique of shadowing current versions of a segment, making the shadow versions permanent on action commit. The data recovery routines are still under-going implementation at present. Since action management will not be available before the completion of the storage manager, testing of the recovery features of storage management will be done by simulating requests by the action management system.

2.2.1.5. Action Management The design of action management is complete. It includes several areas, but primarily it is concerned with the control of action events in the Clouds system. Other features included in the action management design are a simple algorithm for global object searches, the use of a global, kernel database. a time driven orphan detection mechanism (developed by Martin McKendry and Maurice Herlihy at Carnegie-Mellon University), a design for a global Lamport clock for the Clouds system which supports the orphan detection mechanism, and a design for a generalized locking facility for programming objects. Although coding has not started, the design includes enough implementation details so that this effort can proceed quickly.

The orphan detection algorithm mentioned above is quite different than that originally described in [Allc83a]. The new orphan detection mechanism attaches two time values to all action requests, in addition to the usual time-out value for deadlock recovery. These values are a quiesce time and a release time. After the quiesce time for an action has passed that action can initiate no further requests. The release time indicates the earliest time an action can release any locks that it hold. The release time is always greater than the quiesce time. Orphan are prevented from producing erroneous results by the eventual passing of their quiesce times. Generally, the period until an action's quiesce time is not long, requiring a refresh phase which increases first the release time and then the quiesce time of the action. This allows the action to continue work.

Action management's lock facility allows the creation of not only simple read/write locks, but also locks with more complex compatibility modes. For example, it is possible to create a lock with more than two modes and then specify how the modes conflict. In fact, one could create read/write locks in this fashion, but it is expected that the read/write locking style will be popular enough to justify a separate implementation. Also, locks need not be create for a specific instance of a structure, but may be defined over a whole domain of such structures. The data (a file, for example) need not exist at the time the lock is taken. This flexible locking mechanism, along with the redefinition of special object operations, allows the Clouds programmer to customize the recovery of the objects and action that are developed in addition to having available the default system recovery.

The search algorithm, as mentioned, is simple but attempts to do as much as possible to limit searches for objects. This is because in order maintain the location independence of objects from the sites on which they reside, the local object cannot determine the where an object is by examining the capability for the object. The object may be local or remote; invocations should be handled transparently. As the number of nodes in a Clouds system increases, the effort and time spent searching for an object could become quite significant. Therefore, information is kept in a global state database which aids in the search. The information in this low-level database is not guaranteed to be exact or complete. It does provide some hint of where the

object might be by maintaining several sorts of information; the last place an object was found, where the object was moved by the system, or even where the device on which the object last resided was moved. Always, as a heuristic, the node whose name is contained in the birthplace of the object is a high priority.

The global state database was mentioned as a source for clues for the object search, but it is actually more than that. Many types of system information is placed in the database, then to be propagated through the network using the algorithms described [Allc83a]. In particular, action state information, workloads, uplists, and other system imformation can be propagated in this manner.

### 2.2.2. Compiler Development

The systems programming language for Clouds implementation is Aeolus. Currently all development is being done on a VAX running Unix 4.2bsd, using "C". This is pending the full implementation and testing of the Aeolus Compiler. Once the compiler is implemented, and it interfaces to the Clouds system (the compiler generates objects), further development will use Aeolus.

The compiler implementation for Aeolus is currently underway. The Amsterdam Compiler Kit (ACK) is being used to generate code for both the Clouds system running on VAXen, and the Bubbles system running on 8088/8086 based systems (IBM-PC/XT/AT). Work on semantic routines for Aeolus is proceeding in parallel with the development of intermediate code for ACK. This work is being done in Pastel, an extended Pascal dialect.

## 2.3. Fault Tolerance and Probes

Use of probes in monitoring and fault tolerance is being studied. Probes are somewhat like messages, but unlike messages they are handled by traps handlers in processes and special probe handlers in objects. Thus probes can be sent to both passive as well as active entities. This gives rise to a powerful paradigm that is useful for a lot of activities, from monitoring, status enquiries to emergency messages. An application of probes to fault tolerant scheduling has been discussed in [McKe84c].

## 2.4. User Interfaces

The Clouds user interfaces are at several levels. The Clouds system runs a shell that allows hierarchical name spaces and common shell functions as the service routine for each user. The interface to this shell is via the intelligent terminals. This part of Clouds is still under the design phases.

The Human Factors group at Georgia Tech is looking at advanced user interfaces which will use the properties of "transitionality" to handle novice and advanced users at their own levels of sophistication. The transitional user interfaces will be built both at the

intelligent terminal level as well as the Clouds shell level.

## 2.5. Publications

The publications that have resulted from this research have been referenced below.

## 3. References

[Allc82]
Allchin, J. E., and M. S. McKendry, Object-Based Synchronization and Recovery, Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982

[Allc83a]
Allchin, J. E., An Architecture for Reliable Decentralized Systems, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also released as technical report GIT-ICS-83/23)

[Allc83b]
Allchin, J. E., and M. S. McKendry, Synchronization and Recovery of Actions, Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983

[Allc85]
Allchin J. E., Dasgupta P., LeBlanc R. J., McKendry M. S., Spafford E., The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System. (draft)

[Kenl85]8
Kenley, G. An Action Management System for a Distributed Operating System, Masters Thesis, School of ICS, Georgia Tech.

[LeBl85]
LeBlanc, R. J., and C. T. Wilkes, Systems Programming with Objects and Actions, Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, Colorado, May 1985 (also available as Technical Report GIT-ICS-85/03)

[McKe82]
McKendry M. S. and Allchin J. E. Object-Based Synchronization and Recovery Technical Report GIT-ICS-82/15, Georgia Inst. of Tech.

[McKe83]
McKendry, M. S., J. E. Allchin, and W. C. Thibault, Architecture for a Global Operating System, IEEE Infocom, April 1983

[McKe84a]
McKendry M. S. Clouds: A Fault-Tolerant Distributed Operating System. Technical Report, Georgia Inst. of Tech.

[McKe84b]
McKendry, M. S., Ordering Actions for Visibility, Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)

[McKe84c]
McKendry M.S., Fault Tolerant Scheduling Mechanisms, School of ICS, Technical Report, Georgia Tech.

[Pitt85]
Pitts, D.V. and E. Spafford, Notes on a Storage Manager for the Clouds Kernel, School of ICS, Technical Report, Georgia Tech.

[Spaf84a]
Spafford E. Kernel Structures for a Reliable Multicomputer Operating System Thesis Proposal, Georgia Institute of Tech.

[Spaf84b]
Spafford, E. and M.S. McKendry, Kernel Structures for Clouds, School of ICS, Technical Report, Georgia Tech.

# Notes on a Storage Manager for the Clouds Kernel*



**Technical Report GIT-ICS-85/02**

**January 1985**

**Last Revision: October 20, 1985**

*David V. Pitts*
*Eugene H. Spafford*

The Clouds Project, School of Information and Computer Science
Georgia Institute of Technology, Atlanta, Georgia 30332

* This research is funded in part by NASA grant NAG-1-430 and bye NSF grant DCR-8316590

CONTENTS

LIST OF FIGURES

# Notes on a Storage Manager for the Clouds Kernel*

## Technical Report GIT-ICS-85/02

## January 1985

## Last Revision: October 20, 1985

*David V. Pitts*
*Eugene H. Spafford*

The Clouds Project, School of Information and Computer Science
Georgia Institute of Technology, Atlanta, Georgia 30332

Abstract: The Clouds project is research directed towards producing a reliable distributed computing system. The initial goal of the project is to produce a kernel which provides a reliable environment with which a distributed operating system can be built. The Clouds kernel consists of a set of replicated sub-kernels, each of which runs on a machine in the Clouds system. Each sub-kernel is responsible for the management of resources on its machine; the sub-kernel components communicate to provide the cooperation necessary to meld the various machines into one kernel.

This report documents a portion of that research, namely, the implementation of a kernel-level storage manager that supports reliability. The storage manager is a part of each sub-kernel and maintains the secondary storage residing at each machine in our distributed system. In addition to providing the usual data transfer services, the storage manager ensures that data being stored survives machine and system crashes, and that the secondary storage of a failed machine is recovered (made consistent) automatically when the machine is restarted. Since the storage manager is a part of the Clouds kernel, efficiency of operation is also a concern. We wish to reduce the overhead required to ensure the recoverability of secondary storage as much as possible, while adhering to the design goals associated with the storage manager.

* This research is funded in part by NASA grant NAG-1-430 and bye NSF grant DCR-8316590

## 1. Background

In this section we present an overview of the Clouds kernel and discuss the philosophy behind its development. The Clouds approach to providing reliability is through the use of actions and objects, as discussed in [1], [2], [3], [4]. The Clouds kernel provides higher level applications such as operating systems with three primitives: *processes, actions,* and *objects.* An *object* is a typed collection of data which is manipulated by a set of operations. The data structures and the set of operations for the object define its type. An *action* is the unit of (fault tolerant) work in the Clouds system. Actions guarantee failure atomicity of the operations performed by them: the operation appears to either occur totally or not at all. *Processes* in Clouds are similar to processes found in other systems, and represent a thread of execution and control. Actions and objects are passive, waiting for a process to activate them. The model of computation for the Clouds system is that of a set of processes making operation calls on objects to perform services required by the system. In order to make these services reliable, the object operation calls are performed under the auspices of an action.

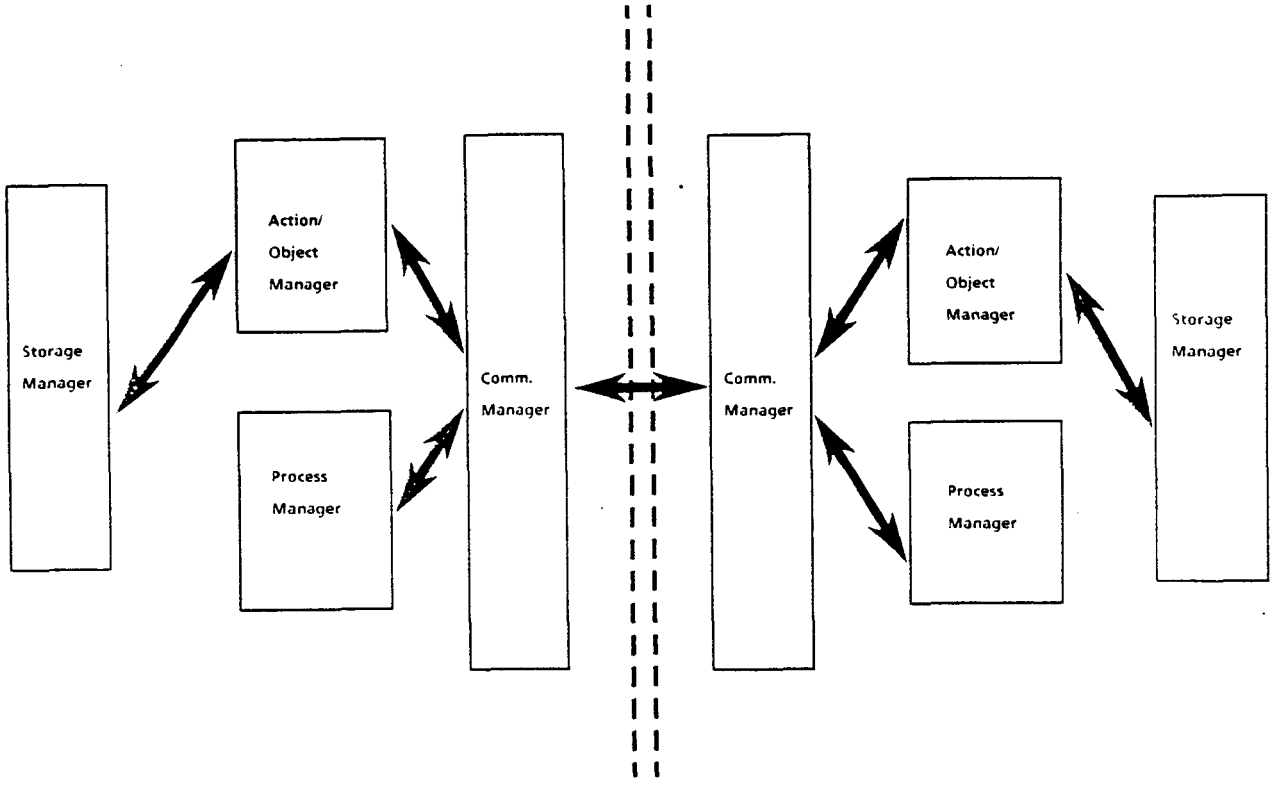



**Figure 1.** Architecture of the Clouds kernel

Clouds actions provide a mechanism that allows the programmer to violate the conventional notions of correctness and consistency, as defined by strict serializability, when programming reliable objects. The programmer can use any semantic knowledge about the intended activation of an object to program a customized method for providing the recovery of the object. This is done by the programmer writing new *abort* and *commit* operations for the object, which indicate how the object data must be recovered. By allowing object recovery to be customized in this way, we hope to provide increased concurrency in the execution of services compared to using the usual recovery and synchronization rules (i.e., serializability), and so improve the performance of the Clouds system.

The Clouds kernel has four major components: the object/action manager(s), the process manager, the communications manager, and the storage manager. Figure 1 depicts the architecture of the Clouds kernel for a system consisting of two nodes. The kernel is made up of two sub-kernels, one of which resides on each node that is part of the Clouds system. Each of the components of the kernel can communicate with its corresponding components on other nodes through the proper protocols.

The object manager is responsible for providing the object operation invocation mechanism. Each object is named by a unique capability comprised of a system name (called a *sysname*) and a series of rights which indicate which object operations are available to the invoking process. The object manager checks the capability provided by the operation call, locates the desired object instance, formats and maps the operation parameters, and activates the object. The object manager is involved with handling action bookkeeping, as necessary. The object manager also hides references to objects on other machines by providing a remote procedure call mechanism (RPC). The object manager makes an RPC look exactly like a local operation call.

The process manager creates, destroys, and dispatches processes. It manages local processes, as well as slave processes started to handle RPC's from other machines. The process manager is not a global scheduler; it simply manages local resources.

The communications manager is responsible for the transmission of information among the machines in the Clouds network. It maintains information about the connectivity of the network, the status of the various lines to which each machine is connected, and queues of outgoing and incoming data. The data that goes through the communications manager is uninterpreted — it might be an RPC or a part of a file that is being transmitted across the network. More detailed descriptions of the object, process, and communications managers can be found in [5] and [6].

The function of the storage manager was described above. It coordinates with the object manager to provide the correct commits and aborts of actions on object data residing on secondary storage. In the remainder of this report, storage will refer to the secondary storage (disk, tape, etc.) attached to a machine. Memory will refer to the main memory of the machine.

## 2. Hardware and Environment

The Clouds kernel is currently under implementation on three VAX 11/750's.[1] The machines have eight megabytes of main memory altogether and are interconnected via a 10 Mb/sec Ethernet. Also connected to the Ethernet are a set of IBM-PC's, which will serve as intelligent work stations. Future versions of the system may be connected by multiple networks of varying bandwidth.

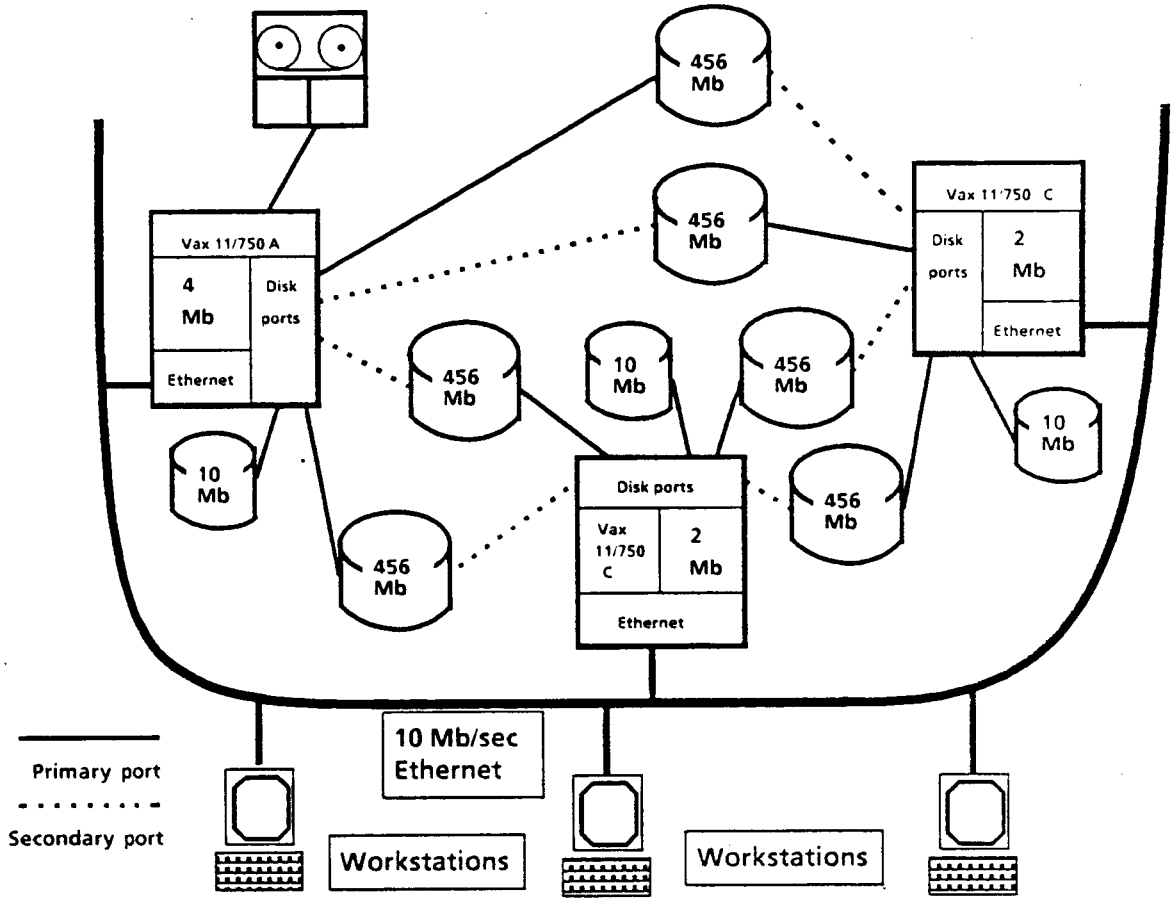



**Figure 2.** Clouds hardware configuration

Our prototype will have three types of storage devices available for the kernel. There may be a tape drive on one machine that will be used to archive data and perform conventional system backups. Each machine will have a RL02 removable pack disk drive, in which each pack provides 10 Mb of storage. We expect that RL02 media will be used as short term archive devices and boot devices. Finally, each system will also have up to four RA81 disk drives. Each such drive has a permanently mounted pack providing 456 megabytes of storage (unformatted). The RA81 drives are dual-ported; two controllers may be coupled to the drive simultaneously. However, the drive is on-line to only one of the controllers at any time. The switching of the device between controllers is done primarily by a front panel switch, but switching can be done under program control. The disks are controlled by UDA50 controllers

1. VAX is a trademark of the Digital Equipment Corporation

which use DEC's Mass Storage Control Protocol (MSCP). These devices are expected to provide the primary object storage for the Clouds system. Figure 2 shows the Clouds prototype system.

## 3. The Design Approach

The Clouds kernel provides user-defined objects[2] as the building blocks (along with atomic actions) of a reliable distributed system. The arguments for using an object-oriented approach in general, and as used in the Clouds project in particular, are presented elsewhere [7] and [3] and we will not repeat those rationales here. We feel that the kernel, in addition to supporting objects for higher levels of software, should also reflect the use of an object-oriented methodology in its design and implementation. To this end we have identified basic components of the kernel and isolated them as modules that are accessible only through a set of procedures defined for each module. These objects are then used to form the major systems of the kernel: the object manager, the process manager, the communications manager, and the storage manager.

We attempt to present kernel objects as typical Clouds objects that provide (restricted) access to functions and services provided by the kernel. However, there are differences between the objects that form the kernel and those that are supported by the kernel. The first such difference is in the implementation. User-defined objects will be created by users with an object-oriented language, such as Aeolus [8], [9]. This language enforces the use of an object-oriented methodology. Our kernel objects are currently implemented as C modules and most of the responsibility for adhering to the philosophy of object-oriented design is the responsibility of the programmer, not the programming language. Still, we believe the careful use of this object methodology despite the lack of support in the language provides benefits in the implementation and later maintenance of the kernel. It also may make the later conversion of the kernel to some other object-oriented language, such as Aeolus, more convenient.

The other difference reflects our concern for the efficiency of kernel functions and the operation invocation mechanism for objects. Many of the functions of the kernel are time-critical, or because of their frequent use require very efficient implementations. The operation invocation mechanism has overhead that we suspect cannot be afforded in these situations. Therefore, operation calls on kernel objects are handled differently than operation calls on user-defined objects. Calls on kernel operations may be performed as ordinary procedure calls or even as simple transfers to blocks of code. However, the appearance outside the kernel and the overall philosophy is that of an object-oriented kernel.

Some kernel objects are not generally available outside the kernel. For example, this is the case with the objects comprising the storage manager. Operating system code may occasionally require direct access to secondary storage, but it is hoped that for the most part the abstractions provided by objects will suffice.

The storage manager is based on three sets of objects: device objects, partition objects, and segment objects. Each of these objects manages the same actual item (secondary storage), but provides different abstractions. The device objects provide conventional device-level access to secondary storage. Partition objects allow devices to be sectioned logically according to the intended use of the storage on a device. Segment objects are the secondary storage extensions of the segment type provided by the kernel. Recoverable permanent objects are implemented at the level of segment objects.

The remainder of this report outlines a design for a storage manager for the Clouds kernel. It covers the important aspects of the structure and function of the storage manager, and discusses how the storage manager is used by and uses other parts of the kernel. The next three sections deal with the design and implementation of the device object, the partition object and the segment object. Those sections specify the data structures required plus the interface to the

2. Also referred to as *client objects*.

objects. Section 7 then covers how these objects are used by the kernel. In that section we discuss some of the issues related to the reliability of the storage manager and its relationship to the rest of the kernel. Section 8 contains a summary of this report, and a few conclusions and reflections on the storage manager.

# 4. Device Objects

As with conventional systems, the storage manager for the Clouds kernel provides a device level interface to secondary storage. This level of interaction with secondary storage is almost exclusively the province of the Clouds kernel. In fact, even within the kernel, most accesses to secondary storage are performed at some other (higher) level; only the storage manager makes frequent use of device objects.

## 4.1 Device Media

The storage manager views devices as two parts: the device itself and the medium currently being used by the device. This viewpoint is moot for fixed media disks, but for other forms of secondary storage, such as tape and removable disk storage, it provides additional flexibility in the configuration of a system. This separation is complete; a sysname exists not only for each device in use on a system, but also for each medium. However, in many cases the distinction between accessing specific media and accessing devices is not important, so we wish to hide this separation. Therefore, the storage manager provides a mechanism for binding a medium to a device.

Bindings between media and devices are generally performed at the initialization of the system and involve the association of device and medium. Binding a medium to a device may also involve the formatting of the medium. In this latter case, a new sysname is generated for the medium. This formatting or initialization of a medium will destroy any previous information that existed on the medium. The old sysname will no longer give access to any medium. The formatting of a blank or obsolete medium includes initializing the tables and structures that the storage manager requires. These structures are discussed in section 2.1.

In other cases, an existing medium is bound to a device. An existing medium is one which has a sysname and is formatted. The binding will involve the reading of the sysname from the medium and comparing it with the sysname passed to the storage manager. The binding will take place only if a match occurs. We are not attempting to address security issues with this design. Our interest is to provide flexibility, while maintaining some control over what is accessible. The use of sysnames to access media provides this control.

Once a medium has been bound to a device, any reference to the device refers to the bound medium. The usual sort of device calls then need only refer to the device. This device-medium binding stays in effect until it is explicitly broken by the storage manager.

In addition to setting up this correspondence between device and medium, this binding also initializes an instance of a storage object in memory. In particular, critical tables and other structures required by the device are brought into system memory. We will now look at the data requirements of device objects.

## 4.2 Device Object Structures

The storage on a medium is presented as a sequence of 512-byte blocks that are addressed by offsets from some fixed block. The offset used to address a block is called a device block number (DBN). As we shall see in section 5, this storage can be subdivided into partitions, allowing the storage on a device to be apportioned for policy reasons. At the device level, though, the storage manager deals only with a contiguous string of blocks; partition boundaries are not visible.

The device object is responsible for the transfer of data between secondary storage and memory. The device requires two tables in order to function. The first such structure is the media header. This table contains basic information about the medium and the device using it. This information includes the medium and device sysnames, the amount of available storage on the medium, and specifications for the device to which the medium is bound.

The other major structure is the index table. The index table describes the partitions that exist on this device. This will include information such as the location, extent, and type. The partition table is 16 entries long. Partitions are discussed in section 5.

The medium header and index table must be resistant to failures — in particular, device failures such as head crashes. If the index table is destroyed by a head crash, for instance, we lose access to the partitions on that medium. We therefore provide copies of the tables, placing the copies on different cylinders in order to minimize the risk from multi-sector failures. The alternate copies will be located in known positions based on some computable function. We do not anticipate problems as far as maintaining the consistency of the slave and master versions of the table is concerned, since the tables are infrequently modified and any such modifications are generally done during the system initialization.

In addition, the device objects will maintain a structure in memory called a flush table. The use of this table is discussed in section 7, but briefly, the flush table allows a device to associate an action sysname with a set of requests. This supports the commit operation performed on recoverable objects. Some devices may require the device object to provide bad sector recovery. Objects written for these devices will have to maintain a bad sector table on disk.

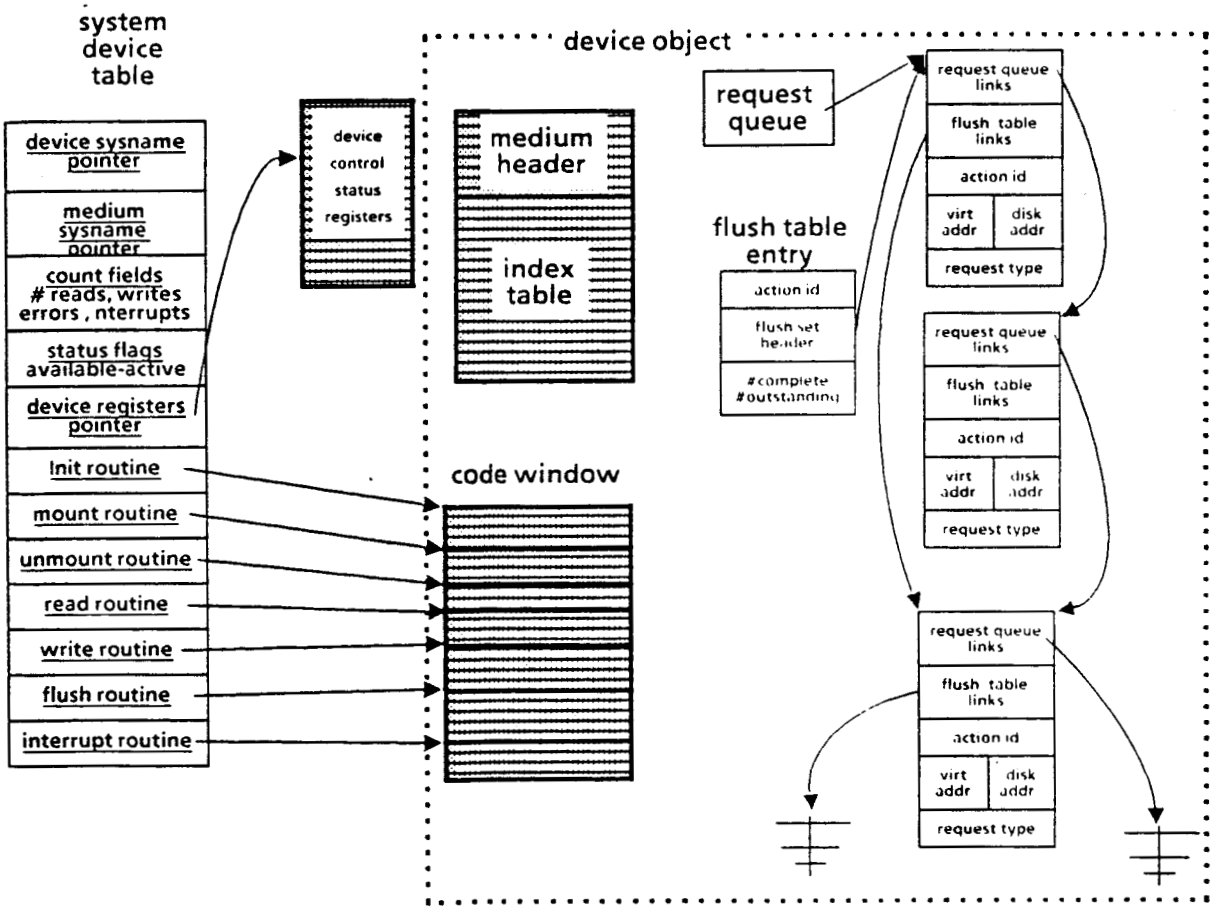



**Figure 3.** The system device table and other device object structures

The device object uses one other structure, the system device table. The system device table is not a part of the device object proper, but is actually the mechanism for managing the various instances of the device objects. This table lists all secondary storage devices that are active on the local machine. The device table entries contain pointers to device and medium sysnames, status variables for the device, device registers, and entry points into the operations for the

device object. Figure 3 shows a device object pointed to by one entry from the system device table.

### 4.3 Device Object Calls

The device object calls deal with the transfer of information to and from the device and with the relationship of the device to its medium. This allows for devices switching the physical medium used for storage in a uniform way. Device and media sysnames are generally needed by those calls setting up a binding between the medium and the device. Calls which perform i/o do not require a sysname. The proper device object calls are found through the system device tables.

*4.3.1 init(devname) return medname*
**Init** generates a sysname for the medium currently on the device specified by **devname.** This sysname is written in the medium header. Also written into the medium header is the device specific information that is required. An area for the medium index table is reserved. The return value is the medium. This is a format call; any existing structure on the medium is overwritten. No other formatting is done, however. Any partitions desired are created later by other calls. Redundant copies of the medium header and index table are created for reliability.

After the medium has been formatted, **init** mounts the device. See the description of **mount** for details.

*4.3.2 mount(devname, medname) returns integer*
This call binds the device called **devname** to the medium called **medname.** The sysname presented to the call is compared to that in the medium header. If the two match, the device and medium are bound. The medium index table and the medium map table are read from the disk. If the device requires it, a bad sector table is created from the device. The return value specifies the status of the call (success, failure).

*4.3.3 return_medium_cap(devname) returns medname*
This call returns the sysname of the medium that is bound to the device named **devname.** The return value is this sysname. If the device is unbound, a valid sysname might still be returned if a formatted medium is present in the device. In this case, the call can be seen as an operation to read a label.[3] This allows us to use media for which all currently existing copies of the sysname are deleted or unavailable.

*4.3.4 unmount(devname) returns integer*
**Unmount** breaks a device/medium binding. All partitions on the medium are de-activated. The return value is the status of the call.

*4.3.5 read(lbn, address) returns integer*
This call transfers the contents of a record located at disk address **lbn** to the page in memory at **address. Read** blocks the calling process on a semaphore until the request is complete and returns an integer indicating success or failure of the request.

*4.3.6 write(id, lbn, address, flag) returns integer*
This call transfers the contents of a page in memory at location **address** to the record located at address **lbn** on the device in question. **Id** is an identification used to associate this request with a set of requests being issued by an action. If **id** is an action sysname, then the device object looks the action id up in a flush table and if it is not there, creates an entry for the action and the request; if the action id is in the table, the request is added to that entry. If **id** is zero, then there is no action associated with this request. **Flag** is used to indicate whether this is a forced

3. This kind of operation might seem to present a security hole in the system, in that it allows the system to determine the name of an unknown medium and then mount it. However, note that this call can only be executed by kernel code or by a user call with special kernel capabilities, and these are assumed to be trustworthy.

write. If flag is non-zero, the device interrupts the normal scheduling of requests by placing this request at the head of the queue. The new request is performed immediately after the current request is completed. A forced write blocks the calling process on a semaphore until the request is complete. Non-forced writes do not normally block the caller.

*4.3.7 flush(id) returns integer*

**Flush** uses the flush table maintained by the device object to ensure that all write operations associated with the action identified by **id** are actually completed. The return value indicates the status of the call. A positive return value (the number of requests completed) indicates a successful flush. A zero or negative return value indicates that the action's sysname was not found in the flush table or that some error occurred while attempting to flush the specified requests.

*4.3.8 enter(partname, size) returns lbn*

**Enter** registers a partition on the device. This involves making an entry for the partition in the index table for the device, placing the partition sysname, **partname**, and the partition size, **size**, in the entry, and allocating storage on the medium for the partition. The starting logical block number for the partition is placed in the index table and is returned as the value of the call. A negative return value indicates that an exceptional condition occurred, such as not enough storage for the partition on the device. **Enter** is called as part of creation of a partition.

*4.3.9 remove(partname) returns integer*

This operation allows the caller to remove a partition from the device. **Partname** is the sysname for the partition. The entry for the partition is removed from the index table on the device and the storage for the partition is deallocated. The return value indicates success or an exceptional condition, such as a non-existent partition. **Remove** is called as part of the removal of a partition from the device.

*4.3.10 partitions(partarray) returns integer*

**Partitions** places the partition entries in the device's index table into the array parameter **partarray**. The major use of **partitions** is expected to be at system initialization, where it passes partition's sysnames to the boot code so that the partitions may be activated. The return value is either the number of partitions written into the **partarray** (a non-negative value) or a negative value indicating an exceptional condition, such as a bad index table.

## 5. The Partition Object

The partition object represents an intermediate level of abstraction of secondary storage. Partitions are consecutive blocks of secondary storage that reside completely on one device. Each partition is a logical object in that it manages the allocation of its own storage and maintains the structures used to do so. A Clouds partition does not enforce any logical organization of the data which resides on the partition, at least not in the sense of a Unix[4] partition. A Unix partition represents a separate file system and all the files on the partition have a hierarchical relationship. The objects residing in a Clouds partition may bear no relationship to each other. It is simply an administrative organization imposed by the partition system indicating how storage in a particular partition is managed.

The two important types of partitions are recoverable and non-recoverable. When a partition is made non-recoverable, it is a declaration that no recovery will be provided for object data stored on that partition and that recoverable objects should not be placed in it. There is no similar restriction for recoverable partitions; such partitions may contain a mix of recoverable and non-recoverable objects. Other partition types include those used for paging surfaces and those reserved for temporary items.

Partitions manage storage as device independent blocks of storage and these are the smallest units of allocation that partitions support. The blocks are addressed by a partition block number (pbn) which is an offset from the beginning of the partition. All partitions are a multiple of this block size.[5]

The partition has as its initial block a header containing partition specifications. The header repeats most of the information found in the medium index table entry for this partition, plus information about the partition's state. This structure is generally accessed only when the partition is activated or some change is made to the partition; at other times the information is in memory and is referenced there.

Another structure used by the partition object is the system partition table. Like the system device table, the SPT is not part of any one partition object instance, but is part of the underlying mechanism. The table contains entries for all partitions which reside on the local machine. Each entry in the table associates a partition sysname with the data structures and information for that partition. These structures and information include the starting block number for the partition, pointers to in-memory structures and buffers used by the partition object, and a pointer to the device object on which the partition resides. This last pointer is actually a pointer into the system device table. Figure 4 shows the complete relationship amongst these structures.

Another function of the partition object is to maintain the location of segments and make available this information upon request. One of the features supported by the Clouds kernel is the location independence of objects (and thus segments). We mean by this that an object may reside on any partition on any node in the Clouds system and may be moved to any other partition on any other node. This implies that each access to an object requires that a (potentially) system-wide search be initiated. The sysnames given to objects give no (definite) information as to the location of the objects. As can be imagined, such searches can be time-consuming. In particular, searches on the partitions at a node might require one or more disk access each. We discuss one method of lessening the impact of these searches shortly.

---

4. Unix is a trademark of AT&T Bell Laboratories
5. The preliminary implementation will undoubtedly make this size equal to the size of a main memory page frame.

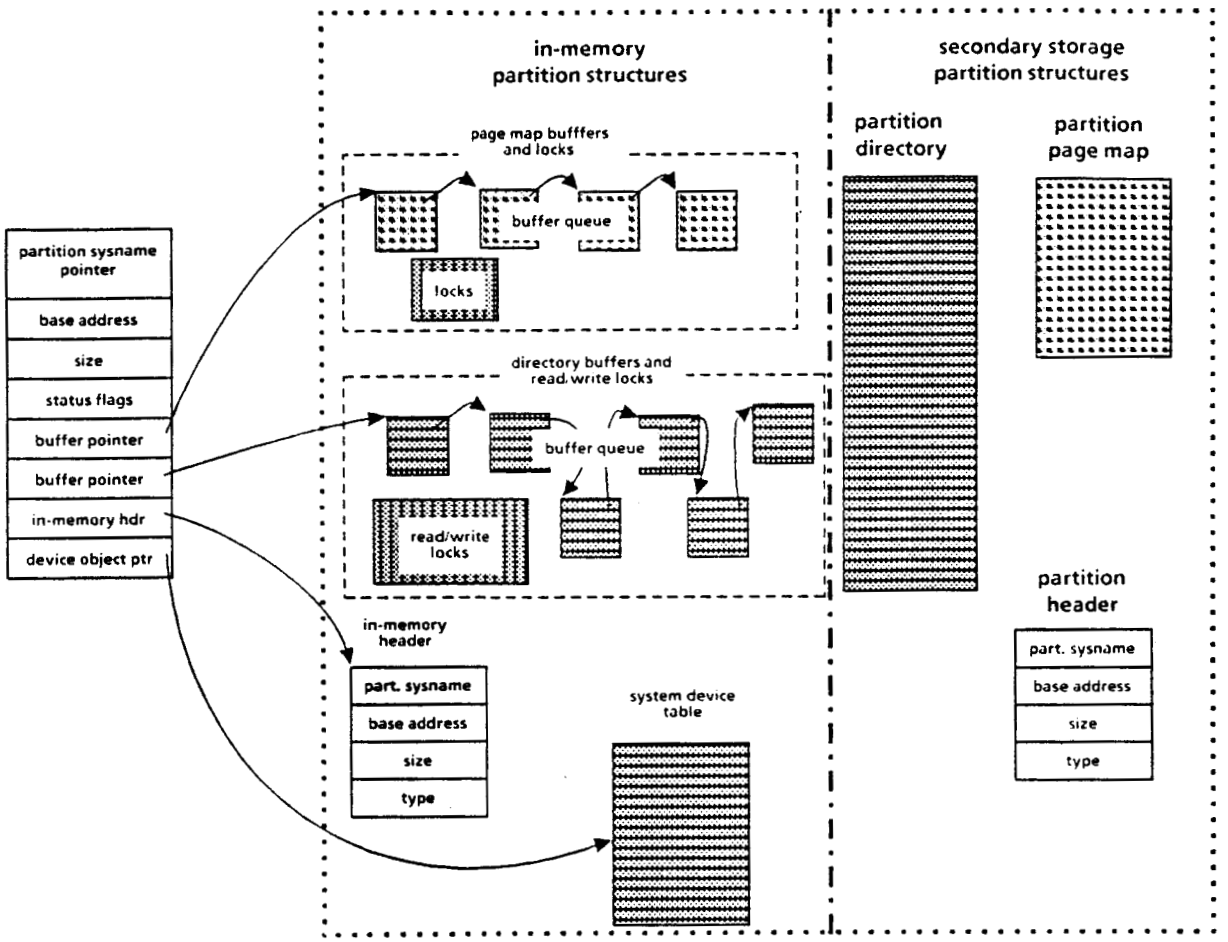



**Figure 4.** The system partition table and other partition object structures

## 5.1 Partition Data Structures

Two of the major functions provided by a partition object are the location of objects and the management of storage. To provide these functions the partition object maintains two structures: the partition directory and the partition page map. The partition directory is a large hash table which is composed of page-sized buckets. In our current implementation the bucket size is 512 bytes, allowing each bucket to hold sixteen entries, each entry consisting of sysname-pbn pairs. The sysname is the id for a segment and the pbn is the offset of the segment within the partition. The entries to the directory are hashed to the proper bucket on the sysname and then to the proper entry in the bucket by a secondary hash function, also on the sysname.

The page map is simply a bit map representing the storage allocation for the partition. This structure, along with the directory, contain most of the information that composes the partition state. As such, they are crucial to maintaining the reliability of the partition and the system as a whole, and some care must be taken in the modification and access of the partition directory and page map, as explained in section 7. Additionally, the storage manager must protect these structures from device failures. The basis for this protection is redundancy of the information. The partition directory and page map have duplicates at known locations in the partition. We are not overly concerned with the extra storage required; we calculate that even with duplicate structures we can keep the storage requirements for these two structures below one per cent of

the total storage. Combined with the protocols we follow for maintaining the reliability of segments and partitions, we should be able to minimize the access overhead caused by this redundancy.

The partition directory and page map may be too large to completely reside in memory and, in fact, we will not have them mapped entirely into virtual memory. Instead, we will maintain buffer areas for the two structures, bringing in new pages from secondary storage as needed, and using a least-recently-used discipline for replacement. We suspect that locality for the page map will be fairly good so that allocations of storage can be done from the memory buffers. However, we suspect that accesses to the partition directory will typically take one access to secondary storage. If our hashing functions are chosen properly we may be able to handle directory requests in (at most) one secondary storage access.

The partition object maintains another structure which it uses to avoid unnecessary secondary storage accesses altogether (or at least make such accesses rare). The structure in question is a Bloom filter [10] which we have called the *Maybe Table*. The Maybe Table is a probabilistic membership checker. It will indicate either that the object in question definitely does not reside on the partition being checked, or indicate that it possibly does. Thus, the Maybe Table gives a method of short-circuiting secondary storage accesses in cases where it gives a negative response. However, a positive response may still lead to unnecessary accesses to secondary storage. The key to success is to reduce the ratio of non-resident positive responses to all positive responses to as small a value as possible.[6]

As described in [10], a probabilistic membership checker is a hash table where collisions are allowed. There are two techniques described in that paper that present methods that could be used with Clouds object sysnames. In the first technique, the Maybe Table consists of a table of transformed entries. The transformation is a hashing function which takes a 48 bit sysname and produces a shorter Maybe Table entry. Several sysnames may hash to the same entry value. This entry value is then placed in the Maybe Table by the use of another hashing function; this time collisions are handled in a conventional manner. To query the Maybe Table, the sysname is once more transformed with the first hashing function, and the proper entry located using the second. If the retrieved entry matches the transformed sysname, a positive response is returned. Otherwise, the collision handling mechanism is invoked and another entry is tested. If a positive response has not been returned upon termination of this procedure, a negative response is returned.

A second scheme is to treat the Maybe Table as a bit-string and use $t$ different hashing functions, each of which returns an index into the bit-string. Placing a new entry in the Maybe Table requires setting the bit whose index is returned by each hashing function. The test for membership requires that all bits whose indices are returned by the hashing functions be set; any clear bit causes the return of a negative response. Figure 5 illustrates the use of these two techniques. In the example, the Maybe Tables are 18 bits in length. In each case, sysnames are represented by three bits in the Maybe Tables. In the first case, sysnames are represented straight-forwardly by three bit entries; in the second case, three bits are set for every sysname belonging to the table.

The benefit drawn from the use of a Bloom filter such as the Maybe Table is that it is a more compact representation of the universe in which membership is being tested. In the case of the Clouds kernel, this is the sysname population of a partition. This allows more of the table to be kept in virtual memory (perhaps all of it), and so queries on the Maybe Table can generally be

6. This is an area that is open to further research. We believe that the goal is achievable by careful selection of the (possibly more than one) filters used, and their manner of implementation. We hope to do some measurements and research on this once the system is working.

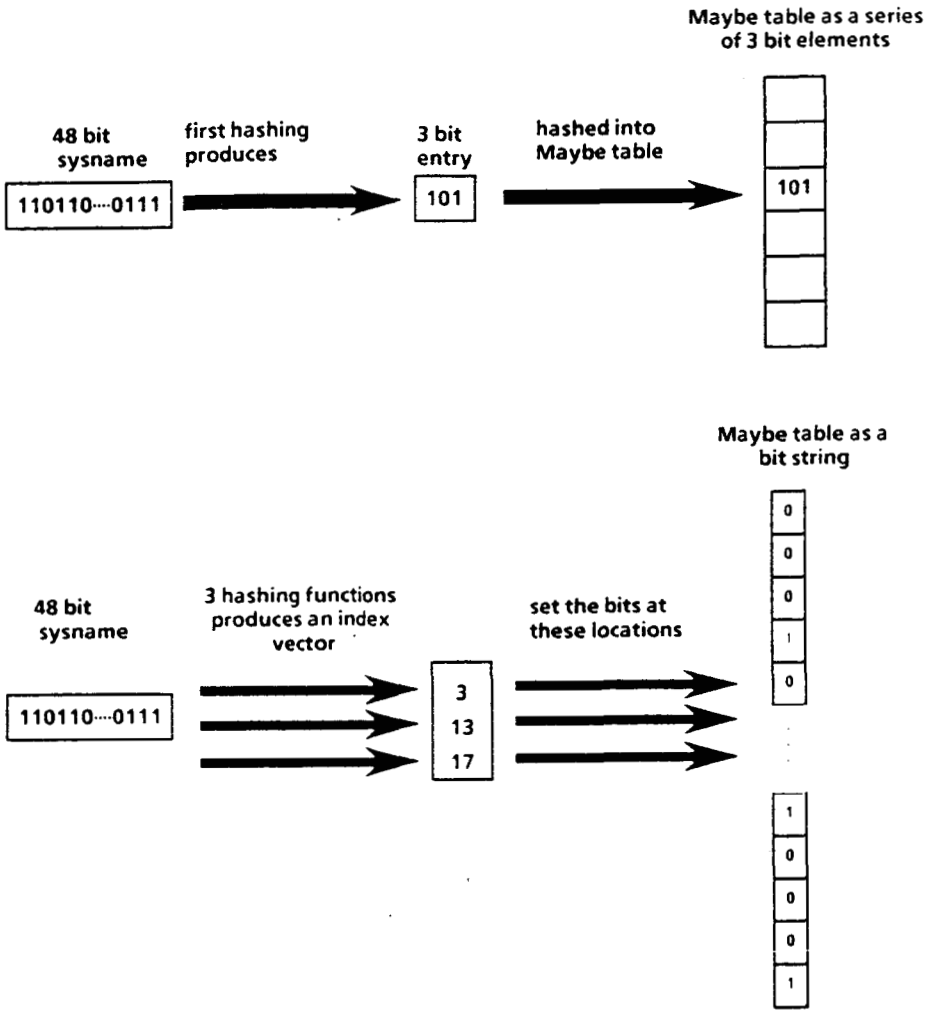



**Figure 5.** Two implementations of a Bloom filter

answered without going to secondary storage. If the response is negative, an unnecessary access to secondary storage is avoided, speeding the search for the proper segment. If the response from the query is positive, then an access to secondary storage is required, to either locate the segment or to ascertain that it is really not on this partition.

Maintaining the Maybe Table has several costs that must be considered. One, of course, is the initial creation cost. The storage manager will perform this initialization at system start-up for each partition and thus the time spent can be ignored. Another cost arises from the dynamic nature of the Clouds system. Objects are created on a partition, deleted from the partition, and moved to other partitions. Clearly, these changes must be reflected in the Maybe Table else the performance will be degraded. Creation of objects and the movement of objects onto a partition pose no problem: the sysname can simply be incorporated into the table via the methods described above. However, deletions of objects and movement of objects from a partition are more troublesome. An entry or set of bits in the Maybe Table cannot be cleared to remove a sysname's presence from the Maybe Table because several sysnames may be represented by the same entry or set of bits.

The simplest solution is to simply reconstruct the Maybe Table at intervals during the lifetime of the system. This reconstruction may be done asynchronously as a background task. The question of when the Maybe Table should be rebuilt is not yet answered. It would seem best to base the interval between reconstructions on activity of the partition, particularly the rate of deletions. This could be be done indirectly by recording the performance of the Maybe Table and reconstructing the table when the performance falls below a given threshold. Or the monitoring could be more direct, measuring the number of deletions and movements of objects from the partition. Both of these methods have advantages and disadvantages. The indirect method for example, seems to be desirable since it measures the attribute that we want to optimize (avoiding disk accesses). However, a burst of queries for a sysname not resident on this partition but which happens to hash to the same entry or set of bits could cause a severe drop in performance even though the table as a whole is behaving reasonably well.

We are currently incorporating a Maybe Table into the partition object as described in [10]. We wish to get the maximum performance from the Maybe Table with the minimum impact on virtual memory. Therefore, we may consider other implementations for the Maybe Table, depending on the performance obtained. It may be, for example, that we are able to take advantage of the nature of the sysname population to improve the performance of the table.

## 5.2 Calls on the Partition Object

The storage management system uses the following calls to manipulate the partition data. Most of the calls require at least one sysname as an input parameter, usually a sysname for the partition (the exception being **create_partition;** see below). Occasionally, sysnames for segments and devices may also be required.

### *5.2.1 P_create(devname, size, partatt) returns partname*

**P_create** reserves a sequence of records on a device to form the partition. **Size** is the size of the partition in bytes (this parameter is rounded by the call to the record size of the device) and **devname** is the sysname of the device on which the partition is to reside. A sysname for the new partition is generated and returned as the value of the call. The record location of the initial record of the new partition is stored, along with the size (in device records) and the partition sysname, in the media index table. The attributes of the partition, specified in the input parameter **partatt** are also stored in this new partition entry. **P_create** makes use of the **enter** call on the device object to perform its task. In particular, **P_create** must be able to request allocation of storage from the device.

### *5.2.2 P_destroy(devname, partname) returns integer*

This call takes the two sysnames given as input parameters and frees the chunk of storage used by the named partition. **partname** specifies the particular partition to be destroyed and **devname** specifies the device on which it resides. The integer return value indicates the status of the partition after the call (destroyed or not found on this device). The call removes the partition's entry in the **media index table** and releases the storage used by the partition. The device manipulations are performed with the device object call **remove. P_destroy** also makes calls on the device object to perform its task.

### *5.2.3 P_enter(partname, segname, pbn) returns integer*

**P_enter** places an entry in the partition directory for a segment. **Segname** and **partname** identify the segment and partition, respectively. The entry in the directory includes the segment sysname and the partition block number, **pbn.** The call also modifies the Maybe Table. The return value indicates success or an exceptional condition.

### *5.2.4 P_remove(partname, segname) returns integer*

This call removes the entry for a segment from the partition directory. **Segname** and **partname** identify the segment and partition, respectively.

*5.2.5 P_return(partname, segname, seginfo) returns integer*
**P_return** returns the segment header indicated by **segname** which resides on the partition specified by the input parameter **partname.** The header includes the sysname for the segment, the size of the segment (in partition records), the record address of the segment header, and whether the segment is recoverable. The segment header is placed in the parameter **seginfo,** which is a pointer to a block of storage reserved for the information. If the segment is present, the return value of the call is positive; otherwise the return value is negative. The call finds the information by searching the partition sysname map and examining the segment header found. The Maybe Table is first queried in an attempt to avoid unnecessary secondary storage accesses.

*5.2.6 P_get_{first,next}(partname, number, segarray) returns integer*
These two calls are similar to **P_return,** in that they return the attributes of a segment found on the partition specified by the input parameter **partname.** The segment is unspecified, however. **P_get_first** places the first **number** of segment sysnames appearing in the partition directory in the parameter **segarray. P_get_next** can then be used to retrieve the attributes of the **number** subsequent segments. The two calls share a static variable which holds the index of the next segment about which information will be returned by **P_get_next.** The variable is reset to zero after the last entry in the partition directory is accessed and is initially set to zero. which is an array large enough to hold the requested number of sysnames. The return value is either zero, indicating no sysnames could be found, or the number of sysnames actually returned by the call.

*5.2.7 P_available_space(partname) returns integer*
This call simply returns the number of free records on the partition indicated by **partname.** A negative value may be returned in exception conditions. The call does a bit count on the volatile record map. Because the volatile free map contains allocations and deallocations for uncommitted actions and because no synchronization is done on the record map, the value returned should be considered only an approximation of the "true" number of free records.

*5.2.8 P_{read,write}(partname, part_offset, address) returns integer*
**P_read** causes the transfer of the contents of a partition record, **part_offset** from the partition specified by **partname** to the physical page in memory indicated by **address.** P_write reverses the procedure, transferring the contents from the physical memory page to the partition record. The calls use their return values to signal exceptional conditions. The virtual memory system uses this call to handle page faults.

*5.2.9 P_getblk(partname) returns pbn*
**P_getblk** simply returns the partition block number of a free page on the partition. The volatile page map is updated to reflect the allocation. A negative value is returned if there is no partition storage remaining.

*5.2.10 P_returnblk(partname) returns integer*
This call deallocates the page at the partition block number passed through **pbn.** The volatile page map is updated. A negative value indicates a bad partition block number.

*5.2.11 P_restore(partname, pbn) returns integer*
The **P_restore** operation is called on system startup to examine the partition. If necessary, the operation will perform any repairs to the partition structures required to bring it back into a consistent state. The call will also cleanup any unfinished action processing. This sort of repair is done on a partition-by-partition basis, since not all partitions have the same attributes and therefore will not require the same processing. In particular, cleanup of action processing is not necessary on partitions not supporting recovery and partitions being used as paging surfaces. **P_restore** must determine attributes of the partition by examining the partition header and then proceed accordingly. The details of **P_restore's** operation are described in section 7, which is concerned with the reliability of the storage manager. **P_restore** also initializes structures used by the partition object, such as the Maybe Table.

# 6. The Segment Object

The segment object provides the final level of abstraction for secondary storage. With these objects, we are operating on blocks of storage allocated by the partitions. The abstraction provided by the segment object is that of a sequence of bytes (kernel segment type). The implementation is actually a tree of fixed length blocks of storage, as we shall see.

Segment objects provide a standard abstraction for the kernel to manipulate and process all Clouds objects. The object implementation provides mechanisms for mapping segment data in and out of virtual memory, creating and destroying segments, and modifying segments. The necessary algorithms for maintaining the reliability of the segment data exist at this level.

The segment object is unconcerned with the internal organization of the objects it is managing. The storage management system treats segments as uninterpreted bytes. Any interpretation is performed by other parts of the kernel, such as the object manager.

## 6.1 Segment Object Data Structures

Recall that a partition directory has a set of entries which contains the pbn for the segments residing on the partition. The partition block addressed by one of these entries contains a segment header that identifies the segment. The complete header is 512 bytes long and contains the segment (object) sysname, the object type sysname, a segment status field, a segment shadow pointer (the status field and pointer are used for recovery), and the size of the segment in bytes. The remainder of the header contains an array of pointers which lead to the segment data. These pointers address one of two sorts of blocks: index blocks, which are arrays of pointers to other blocks, and data blocks, which actually contain segment data. If, however, the storage required for segment data is less than that used for the array of pointers in the segment header, the segment data can be placed in the segment header itself. This would provide for the efficient processing of very small segments. Figure 6 shows the segment structure.

A segment is a tree whose depth depends on the amount of data in the segment. Hence, the smallest segment may have a depth of two (the header and the data blocks addressed by the header), but trees of arbitrary depth are supported. This also means that occasionally the segment will be restructured when its size is increased.

The interaction of the segment system and virtual memory is still being designed. It should be pointed out that much of the manipulations performed by the segment object will involve the segment's representation in virtual memory and the structures maintained by virtual memory itself. The segment system also makes some assumptions. One of these is that the location of the segment is known. That is, the action or process using the segment knows the partition on which the segment resides. Particularly, most segment calls do not require a partition sysname as a parameter.

## 6.2 Calls on the Segment Object

The following calls all require the sysname for the segment being manipulated. Any offsets are data record offsets, using the logical view of the segment.

*6.2.1 S_create(partname, segname, attr) returns integer*

**S_create** allocates storage for a segment and sets up the segment header and index records. The input parameters are the two sysnames for the partition and segment to be created,[7] and a structure holding information about the segment (its size, object type, recoverability). The storage for the segment can be allocated and structured on the basis of the size field of **attr.**

7. Note that this call does not return a new sysname for the segment. If that were the case, it would not be possible to move existing segments into a partition and still reference them by their old names.

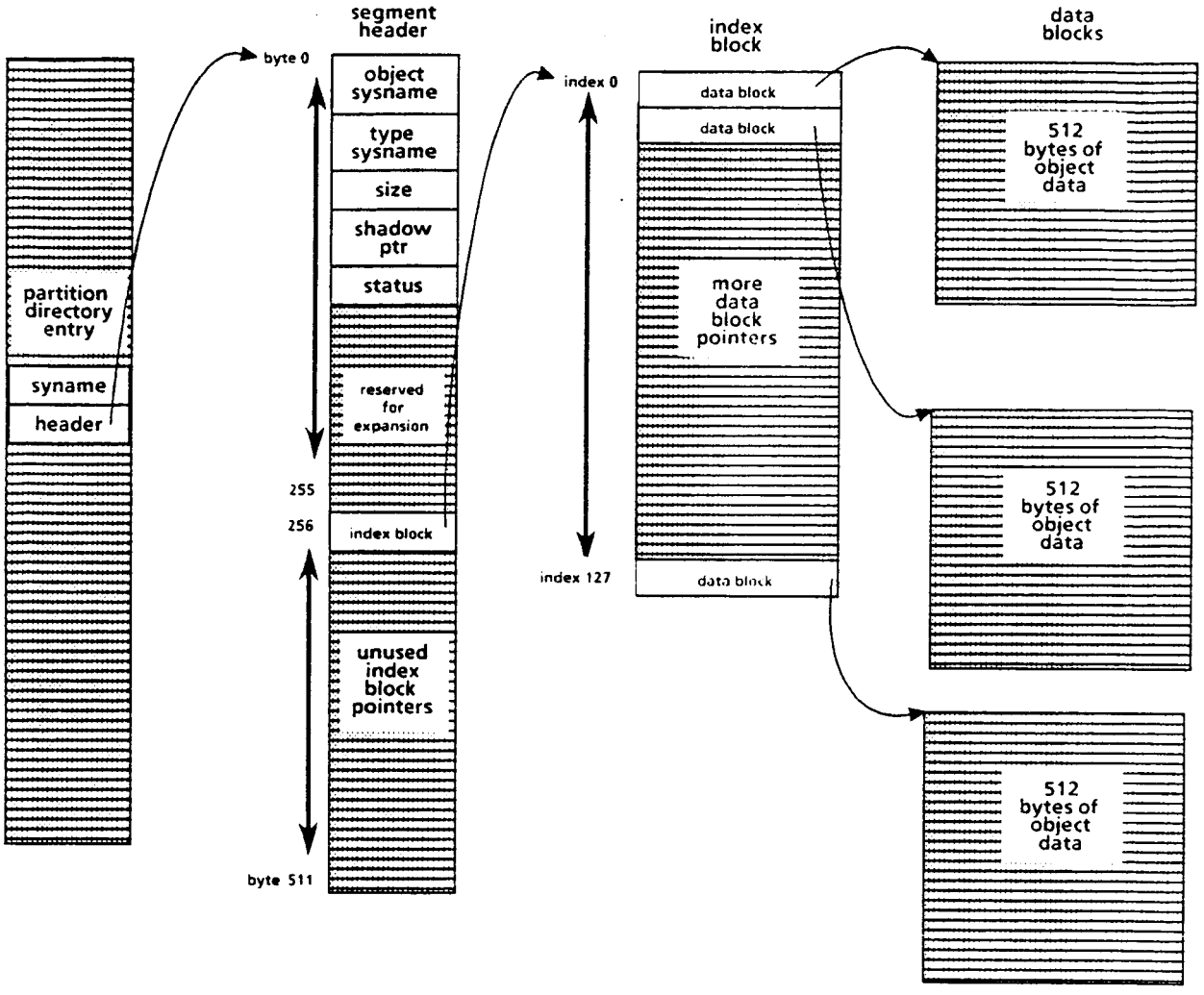



**Figure 6.** Clouds kernel segment structure

Data records are written in subsequent requests. The return value indicates the call status.

*6.2.2 S_destroy(partname, segname) returns integer*
This call deallocates storage for a segment. The sysname for the segment, **segname,** is removed from the partition directory.

*6.2.3 S_read(segname, offset, size, addr) returns integer*
The **S_read** call causes the transfer of **size** number of pages from storage to memory. **Segname** identifies both the memory and storage versions of the segment. The source of the pages is at location **offset** of the segment named by **segname. Addr** is the virtual memory address of the transfer destination. The return value indicates the status of the call.

*6.2.4 S_write(segname, offset, size, addr) returns integer*
**S_write** transfers data from memory to storage. **Addr** is the source of the transfer, in this case a virtual memory address. **Segname** is the sysname for the object (segment) whose data is to be transferred. Note that this identifies both the memory pages (source) and the secondary storage pages (destination) that must be transfered. **Size** number of pages, beginning at offset **offset** of the segment, are copied from virtual memory to the storage segment. The return value indicates the status of the call.

*6.2.5 S_precommit(aid, touchlist) returns integer*
**S_precommit** performs the segment level precommit protocol as described in section 5. **Touchlist** is a list of the objects which have been modified by the action. **Aid** is the sysname of the action making the precommit call. The call return value indicates the success or failure of the call.

*6.2.6 S_eoa(segname, flag) returns integer*
This operation performs the segment level commit or abort protocol as described in section 5, depending on the value of **flag.** The return value indicates the success or failure of the operation.

*6.2.7 S_chgsize(segname, delta) returns integer*
The call allocates or deallocates storage from the end of a segment. **Delta** is the number of records to allocate or deallocate (positive or negative value, respectively). The return value is the status of the call.

*6.2.8 S_status(segname) returns integer*
This call determines the state of a secondary storage segment by examining the status field of the segment header. The return value is this status (permanent, shadowed, precommitted).

## 7. Reliable Storage Management

In this section we look at the techniques used to ensure the reliability of the storage manager in the presence of machine failures and action aborts. All the techniques described below require the information and features provided by the use of atomic actions. This information includes the knowledge of when it is correct to make the effects of an operation permanent and what data has been modified. The storage manager provides a set of protocols that use this information to make the correct updates to secondary storage so as to leave the storage system in a consistent state. In order to understand these techniques and the motivation behind them, we need to understand how the Clouds kernel manages actions.

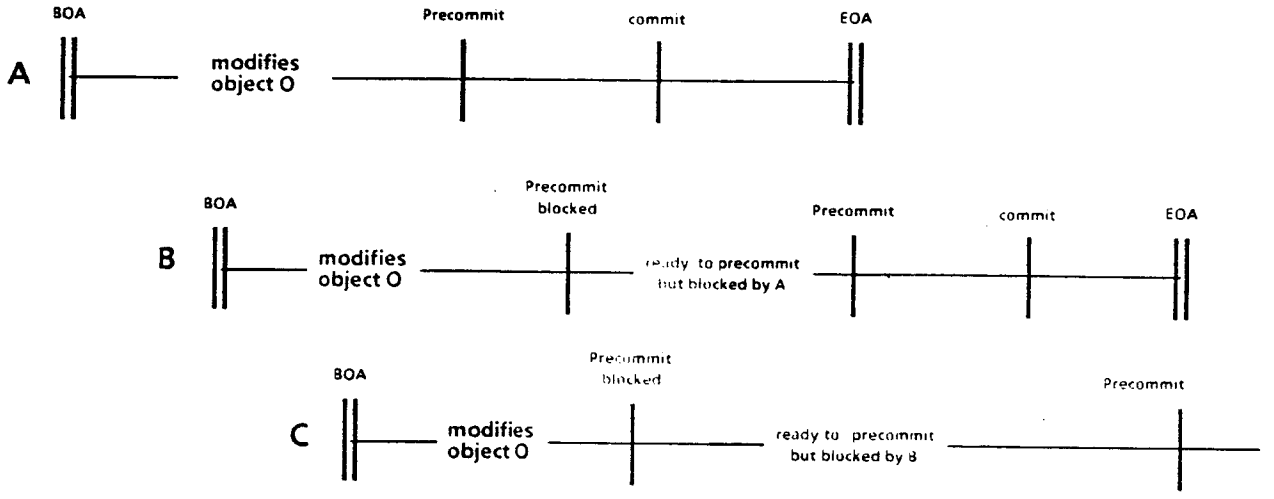



**Figure 7.** Actions block on competing commits

The Clouds system considers actions to be units of work. Many actions may be active in the same object, with each action updating object data. The only restriction enforced by the kernel[8] on the synchronization of actions which are operating concurrently on a single object is at action precommit. An action that precommits in an object blocks all other actions from precommitting in that object until the precommitting action is committed. Other actions still update and process the object's data; the only restriction is on the precommit procedure. Although this restriction may seem to create potential bottlenecks, the simplifications it provides in the processing of commits will keep the blocking intervals short enough so as to cause no problems. In particular, this restriction means that the storage manager must provide reliable updates for only one action per object per time period.

There are two levels at which the storage manager must supply this sort of reliability: at the partition level, and at the segment level. The partition has critical data which must be updated correctly to allow the storage manager to function correctly. As stated previously, this data includes the partition directory and the partition page map. At the segment level the storage manager is responsible for the consistent update of object data and the underlying structures that represent this data. We use two rather distinct approaches to providing the recovery for these two levels. In both cases the techniques provide pessimistic recovery; no changes are actually made to the "live" data until the responsible action commits.

8. The programmer may define other forms of synchronization within the implementation of the object based upon semantic knowledge and other design factors. The kernel does not preclude such choices.

## 7.1 Segment level recovery

Segment recovery is accomplished via a shadowing scheme[11]. That is, segments on which actions are operating will have shadow versions which the actions will actually see. We note that one of the goals of the recovery scheme is, aside from producing consistent results, to allow recovery of segments (and partition structures) with as little storage overhead as possible, and with as few storage accesses as possible. Shadowing, then, will be minimal, with only those parts of the segment actually modified being shadowed.

The shadowing scheme consists of a set of protocols that indicate what the storage manager must do for specified segment states and action events. We consider these states and events in the following paragraphs and develop the protocols that shadow segments. When an action is started, the storage manager is involved initially in the transfer of the data for the object being operated upon from storage to memory. Until precommit occurs, the only transfer of information is from device to system. All modifications to the action data are handled in memory by the action manager. On the action commit the storage manager starts transferring information back to storage. These transfers are the result of the action management system protocols for transfering action updates to the permanent state of the object.

### *7.1.1 The precommit protocol*

The precommit protocol ensures that updated pages of object data that an action has modified are recorded on non-volatile storage to prepare for the final commit of the action. The storage manager performs the shadowing and data transfers as follows:

P1 The storage manager determines how many pages are to be shadowed and allocates storage for shadow versions through calls to the virtual memory system and the partition object, respectively. The storage manager allocates shadow storage not only for modified data pages, but also for the segment header, plus any index pages that are required to reach a modified data page.

P2 The storage manager shadows the segment. The segment header is copied to the shadow segment header. The modified data pages are copied from memory to the shadow data pages. Modified versions of index pages are copied to shadow index pages. Some index pages must be modified and shadowed so that the shadows point to the shadow versions of data pages. The storage manager places a modified version of the segment header into the shadow segment header. Modifications made to the segment header data could include a change in the size, and changes to the array of pointers (some of these pointers may point to shadow pages, as with the index pages).

P3 The permanent segment header is modified so that the status flag indicates that the segment is being shadowed. A pointer is also set in the header which indicates the location of the shadow segment header.

One point to note about the above protocol is that there are a number of reads assumed to get the segment structure into memory. Also note that the number of pages that must be shadowed and the identification of which index pages must be shadowed can be determined by knowing the size of the segment and which data pages must be shadowed. The segment header is modified last to reduce the work necessary to restore the segment in the event the system crashes before the precommit is completed.[9]

Once the precommit completes, we are left with two versions of the segment. The two versions overlap in spots as illustrated in Figure 8, where blocks within the dashed box are part of the

9. A crash at any point before this final write will recover with the shadow pages still listed in the free space list and completely unreferenced, and thus they get scavenged automatically.

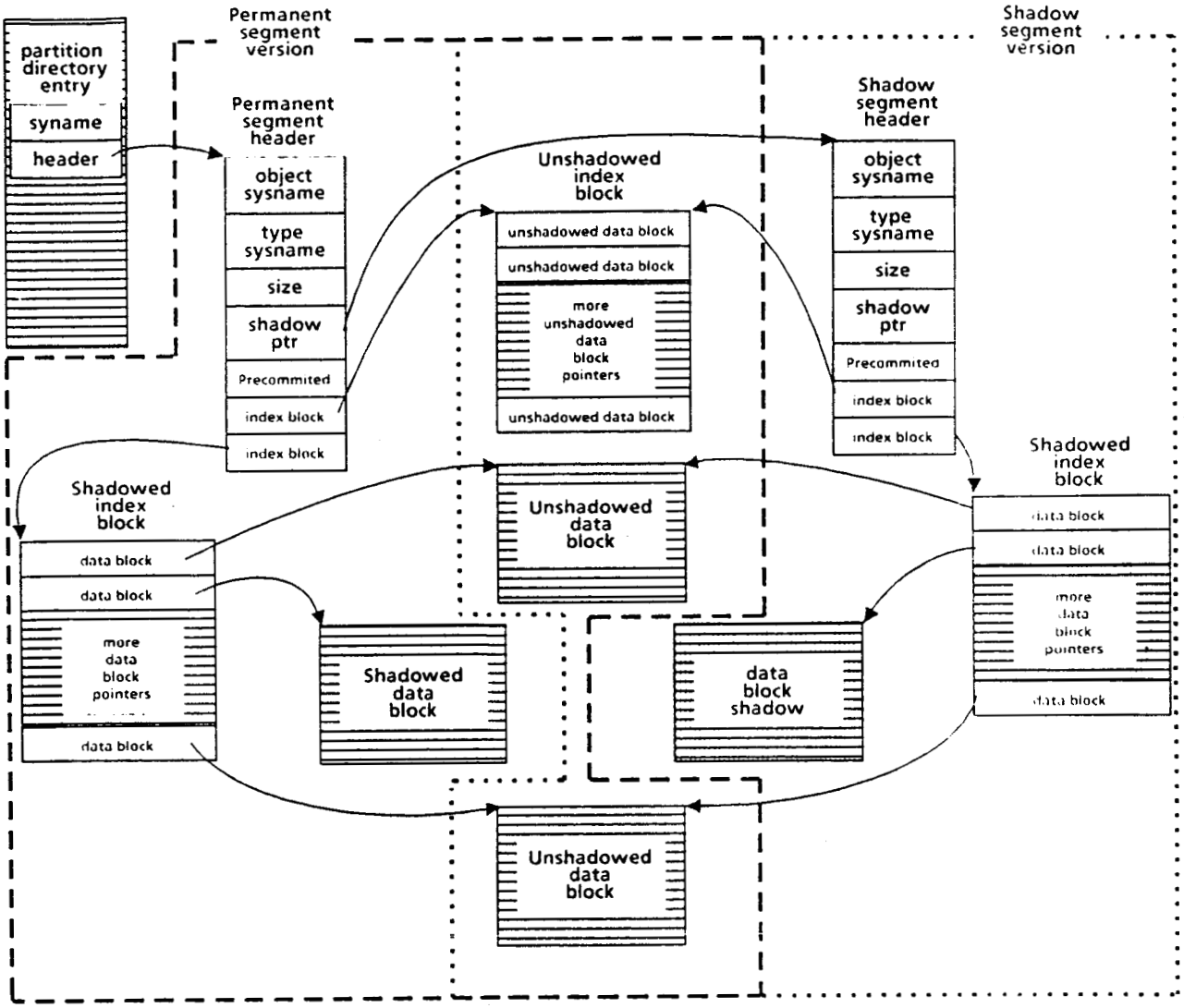



**Figure 8.** Precommitted segment

permanent version, while blocks inside the dotted box are part of the segment shadow. Read operations on unshadowed pages refer to permanent pages. The shadow version is visible only to the action which is performing the commit.

We must point out that the storage manager's precommit protocol is not the same as the action manager's precommit. After the storage manager has completed the shadowing, the action could still abort and the shadowed version would have to be removed. An example of such a situation is when the action spans several nodes and uses a two-phase commit protocol. Phase one is complete only when all nodes have completely shadowed any object data the action touched on their storage. If one node cannot do this, the action aborts.

*7.1.2 The commit protocol*

Once the segment is shadowed and the action decides that it can continue the commit, the storage manager performs its own commit protocol. The storage manager must switch the shadow version for the old permanent version of the segment. There is some bookkeeping for the partition as well. The protocol is as follows:

C1 Update the permanent page map on storage. This requires that all addresses for shadow records be allocated in the page map and all modified records of the segment including the segment header be deallocated in the page map.

C2 The partition directory is set so that it points to the new segment header for the segment.

C3 The shadow segment header is set so that it is now the permanent segment header, that is, it is marked as "permanent."

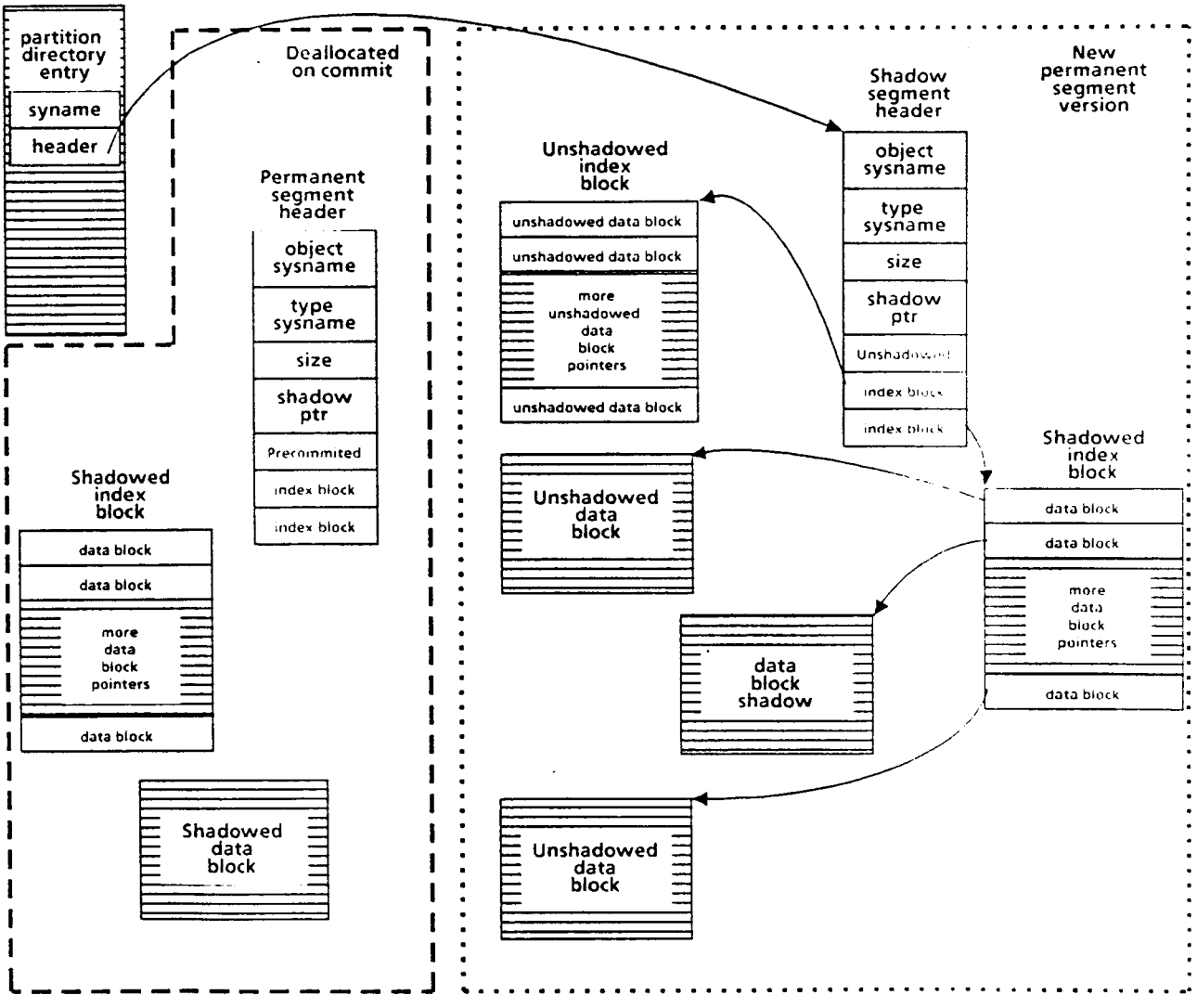



**Figure 9.** A committed segment

Once this protocol is complete, any references to the segment will refer to the new version of the segment. The new segment is a merging of old unmodified records and new records. Figure 9 shows a committed segment. The blocks in the dashed box were parts of the permanent segment being shadowed during precommit. These blocks are deallocated as part of the commit during step C1. During this phase of the protocol, the storage manager updates the permanent page map on secondary storage. Recall that Clouds uses pessimistic recovery and any effects of an action, including storage allocation to perform the commit, cannot become permanent until the action commits. Therefore, all allocations are performed on a volatile page map. We discuss this and other ideas in the section on partition level recovery.

*7.1.3 The abort protocol*

Actions can also abort for one reason or another and the storage manager requires a protocol for this event as well. The protocol simply rids the segment of any trace of the action's work as follows:

A1 The volatile page map is updated to remove allocations that the action has made to shadow the modified pages of the segment.

A2 The status flag of the permanent segment header is set to show that the segment is unshadowed and then the shadow pointer is set to null.

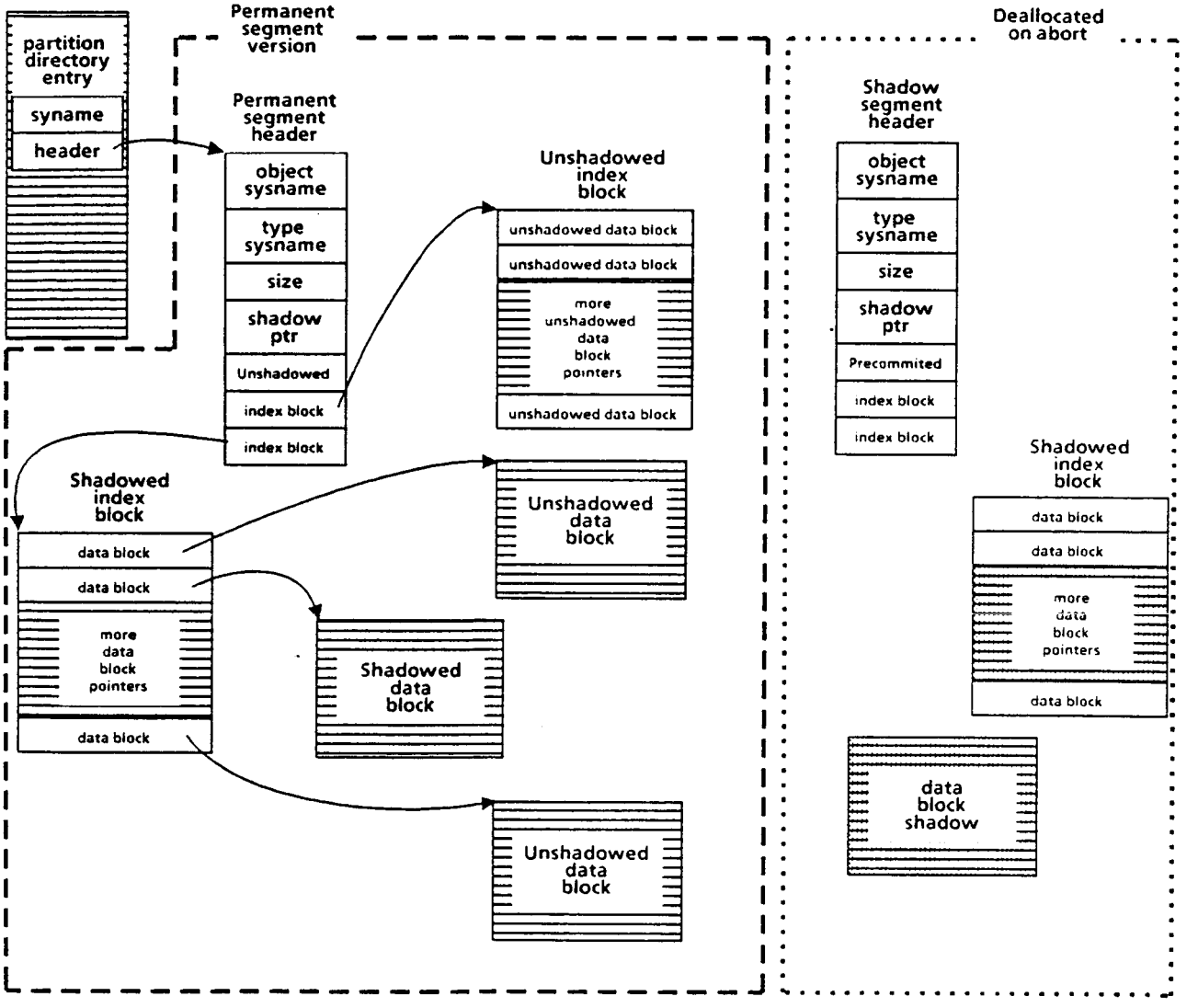



**Figure 10.** An aborted segment

The storage manager uses this protocol only when an action has started to commit and aborts in the middle. If the action aborts before attempting to commit, the storage manager is not involved at all. Figure 10 illustrates the results of the abort protocol. In this case, the blocks inside the dotted line are deallocated upon the abort, as these blocks are only shadows for the permanent segment.

*7.1.4 System failures*

One final event must be considered. That is how does the system recover from a machine crash? Specifically, we are concerned with restoring the segment and partition to a consistent state after the system is brought up again. The system may have had a number of actions in various states at the time of the crash and we want to insure the appearance of indivisibility of actions. Under the Clouds policy, any action that has not precommitted when a crash occurs is aborted when the system is restored. As we have already noted, actions which do not begin precommit before the system crashes do not concern the storage manager; these actions have no effect on system storage. For objects which completed precommit processing, we must determine whether their action's effects become permanent or are erased. This depends on the state of the action. The crash recovery protocol, then, is as follows:

CR1 A new volatile page map is created for the partition.

CR2 The storage manager determines which actions touched segments on this partition and determines the state of each such action. The storage manager polls a kernel database and examines the segments on its local storage to identify these segments.

CR3 If a segment was touched by an action that has completed phase one and should be committed, the storage manager performs the commit protocol on the segment, as above.

CR4 If the action which modified this segment was aborted by the action manager, the storage manager uses the abort protocol, as given above.

At the end of crash recovery, the partitions are in a consistent state; either the actions occurred or they did not. The database referred to in step CR2 is a kernel level database shared by the nodes in the system. The database exchanges information amongst the systems using a suite of algorithms developed in [1]. The information in the database represents an approximate state of the network. This database is copied from other nodes by the kernel when a node is added to the Clouds system. Among the information kept in the database is a list of actions, their status, and segments touched by the actions. Generally, the storage manager can find here the information needed for crash recovery. In some cases, though, a local action (one which does not leave the site on which it is born) may not appear in this list, even though its status at the crash time was complete and known. In cases such as these, the storage manager can find shadowed segments only by an exhaustive examination of the partitions.

Another issue is that of a system failure during an action write, so that only part of the write is actually completed. In the discussion thus far, we are assuming that we have atomic single record writes. The atomicity we are concerned with is failure atomicity, whereby the write either takes place or not. In practice, this means that we can detect an incomplete write (the system failed during a record write) and we are not overwriting the only copy of the data in question. If a device we are using does not support detection of incomplete writes, we can simulate the effect using the standard method of stable storage as described by Lampson and Sturgis in [12]. In [13] the question of when the atomic single record write assumption can be relaxed, if at all, and under what circumstances, is investigated.

## 7.2 Partition level recovery

In the last section we outlined the techniques used to provide reliability for the segments on storage. We now turn to the problem of maintaining the consistency of partition structures, particularly the page map and the segment directory. These structures were discussed to a small extent in the last section because they are involved in shadowing segments. We did not discuss how the structures themselves must be modified to maintain their consistency. Once again, let us consider the action environment provided by the kernel. Recall that a committing action blocks all other actions from committing in a segment it has modified. The partitions are objects, so that any action committing would block all other actions from committing in any object residing in that partition. For a one partition node, this permits only one action at a time to commit. We feel that this is too restrictive.

We allow any number of actions in a partition to commit simultaneously, excluding any segment conflicts. Given this, we do not feel that shadowing can be used to provide recoverability of the page map and directories. Maintaining the various shadow versions in itself would be complicated, but in addition we would need to propagate committed data to as yet uncommitted shadowed data. We therefore reject our segment level shadowing scheme as an approach for partition level recovery and we must develop another method for this task.

The partition directory does not have a volatile component. There are two copies of the directory residing on the partition (for the redundancy necessary to protect against media failures) and a committing action on a partition object must update both copies in a consistent manner to indicate that the new object version is to be used. Once again, we assume atomic single record writes, which will allow us to determine whether the copies are consistent, when the writes are performed in a determined order. An examination of both permanent copies and the header of the segment involved, if done in the proper order, will reveal any inconsistencies and the manner in which they should be resolved.

The partition page map has a volatile component which the storage manager uses to make non-committed storage allocations and which disappears after a system crash. Note that the volatile page map provides correct storage allocation information excluding system failures. Now recall that the commit protocol for storage management entails three steps, the second of which involves installing the action's storage allocations onto the permanent page map. We have two approaches we feel will provide consistent updating of the permanent page map. The first approach simply does away with the permanent page map of the partition, and maintains only the volatile version. As noted earlier, this provides correct storage allocation until a system failure occurs and the page map is lost. Clearly, we must be able to recover the page map after the system is restarted, and the obvious solution is an examination of the partition. Equally clearly, this will require quite extensive processing upon system startups.

The second approach to maintaining the partition page maps involves the use of intention lists and does require a permanent copy of the page map. With this approach, the storage manager during step one of the segment commit protocol does not write directly to the permanent page map, but instead writes an intention list of storage allocations (deallocations) to disk. Because the volatile page map reflects the correct storage allocation for a partition, the actual updating of the permanent page map from the intention list can be performed as background processing by the storage manager. If the system crashes before some updates are performed, they can always be done as part of the system startup processing. The steps required by this protocol are shown below:

1. The creation of the intention list begins at precommit. When the shadow is allocated, the storage manager places these pages on the allocation intention list. The pages to be replaced by the shadows are placed on a deallocation intention list.
2. When the signal is given to start the final commit, these lists are written to a list of pending allocations maintained by the partition.
3. At some later time, these lists are merged into the page map as part of normal partition bookkeeping.

The only restriction is that the updates from the intention list must be performed in the order in which the allocations and deallocations were committed.

Our initial implementation of the storage manager will use the first mechanism. We have two reasons for doing this. First, we are concerned more with the cost of commit processing than we are with system startup processing simply because we feel that system failures will be infrequent and because action processing is our model of computation. This approach both simplies the implementation and makes the commit process more efficient, since no extra disk writes are required to update a permanent page map.

Secondly, an extensive examination generally will be made of the partitions at system startup to clean up any unfinished action commits or aborts. The reconstruction of the page map is partially subsumed in this processing.

### 7.3 Device support for recovery

The above protocols have several implicit assumptions on which they rely to operate correctly, two of which concern the device object. We have already mentioned the assumption that devices can perform atomic single record writes. The other assumption concerns the transfer of data from system to storage. The protocols assume that upon completion of a call to any of the "write" operations the data intended for transfer to storage has, in fact, been transferred. Under conventional systems, this is not necessarily the case, since requests for writes to storage may be buffered. Data may or may not actually be transferred before the system crashes. If the data were not actually transferred, there is no way to recover the segment or partition when the system is restarted.

At the device level, then, the storage manager requires some way in which to ensure the timely completion of data transfers. We wish to accomplish this without adversely affecting the other processing on the system. Also, the action causing the writes to storage must be informed of the completion of the writes in order to continue its commit processing.

There is a great deal of latitude with the timing of when the action writes are forced to the device. One discipline is to have a synchronous write operation that immediately forces the device to schedule requests issued by the operation. By this we mean that any requests currently being processed are completed and then normal scheduling is pre-empted. Synchronous write requests are then carried out in order of receipt. Thus, action writes are forced to the device early in the sequence of action commit processing. The drawback is that requests for synchronous writes appear in bursts at precommit and commit. Any scheduling that the device does for efficiency of the device's operation is disrupted.

Another approach is to allow the device to schedule the requests subject to its own constraints and simply inform the storage manager when the requests are completed. This allows the devices to schedule requests efficiently, but can delay action commit processing. However, the storage manager does know when the completion of the precommit and commit protocols can be safely signalled.

A compromise approach initially allows precommit and commit to be enqueued as usual and handled as normal requests. It is only when completion of the commit or precommit is imminent that the write must be forced to storage. To accomplish this, requests must be identifiable by the storage manager so that the manager can signal which requests must have priority. The manager can simply place the action id of the committing action in a field of the request when requesting a write to storage.

When the storage manager determines it is necessary, it can make a call on the device object to reorder its queue of requests, giving priority to this action's requests. This technique may prove useful if a significant amount of time can elapse before the storage manager must complete the precommit and commit procedures. In cases where the action has touched a number of objects on several systems this may indeed be the case. In such situations, the devices can operate efficiently (and possibly reduce the number of pending precommit and commit requests, reducing the disruption when it becomes necessary to force them to storage), and the action is not delayed, since it is not ready to complete its commit. To accomplish this as stated, the storage manager must be able to identify when requests must be forced to storage. This will be based on the results of any two phase commit that is performed and the storage manager will rely on the action management system to signal when final commit is to be performed.

Each device object maintains a flush table (as discussed in section 4) to control the forcing of action writes. When the list of requests for the action entry in the flush table is empty, the storage manager can inform the action that the commit processing can continue.

## 7.4 Summary

Support for reliability and recovery is integrated throughout the storage manager from the lowest level to the highest. The segment system, via the use of segment objects, provides for recovery of client object data recovery through the use of shadowing of modified data and the discipline of the shadowing provided by the protocols discussed above. The data that the storage manager uses to manage Clouds objects is made recoverable by the partition objects. At this level, our primary concern is how to maintain the data across system failures, and we present a few approaches for doing this. At the device level, support is provided to ensure that data is written when necessary, allowing action processing to be performed correctly at a higher level.

## 8. Conclusions

The motivation behind the Clouds project is the belief that systems in general and distributed systems in particular should provide reliable data management and reliable computation. This report documents part of our efforts towards that goal, namely the storage manager for the Clouds kernel. The Clouds storage manager, in addition to providing the traditional services of storage management, also provides support for the object-action methodology presented by the Clouds kernel.

We have presented an overview of the storage manager for the Clouds kernel. The storage manager is presented as a collection of objects, each of which provides an abstract view of the secondary storage. At the lowest level, secondary storage is viewed through the device object, and the physical storage medium is viewed as a sequence of pages (in the current implementation, a page is 512 bytes) with very little structure, other than the device header and index table. One step higher in our hierarchy is the partition object, which manages a portion of the raw storage provided by the device object. Once again storage is viewed as a sequence of pages, but that storage has a more defined structure. Each partition maintains a directory and a page map, so that each partition is responsible for managing its storage and for providing a location service for the next level of abstraction, the segment object. The segment object provides a view of storage that is a sequence of bytes and each segment object generally corresponds to some other kernel or user object. The storage manager views segments as a tree-like structure of pages.

We have described the data structures associated with each object and presented the operations with which the data structures can be manipulated. We have also tried to convey the relationships amongst the three objects and to show how they interact with each other and the rest of the kernel.

The research that we are conducting is primarily involved with how the storage manager provides the recoverability of the storage it manages and thus supports the reliability of the Clouds kernel. To that end the storage manager uses a set of protocols to ensure that object data is updated in a consistent manner and that even through system failures, enough information survives to maintain the consistency of the object. We show how these protocols are used to support the action/object programming paradigm of the Clouds system.

Each level of storage object discussed provides some support for recoverability. The device objects maintain flush tables which allow the storage manager to ensure that action writes are completed before a commit is finalized. The partition object maintains a consistent view of allocated storage and insures the correct updating of the partition directory. The segment object provides recovery of object data through the set of protocols described.

*REFERENCES*


1. Allchin, Jim, *An Architecture for Reliable Decentralized Systems,* Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1983.
2. McKendry, Martin, *Ordering Actions for Visibility,* Technical Report GIT-ICS-84/05, Georgia Institute of Technology, Atlanta, Georgia, 1984.
3. Allchin, Jim and Martin McKendry, *Object-Based Synchronization and Recovery,* Technical Report GIT-ICS-82/15, Georgia Institute of Technology, Atlanta, Georgia, 1982.
4. Allchin, Jim and Martin McKendry, "Synchronization and Recovery of Actions," *Proceedings of the Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Montreal, 1983.
5. Spafford, Eugene and Martin McKendry, *Kernel Structures for Clouds,* Technical Report GIT-ICS-84/09, Georgia Institute of Technology, Atlanta, Georgia, 1984.
6. Spafford, Eugene, *Kernel Structures for a Distributed Operating System,* Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, in preparation.
7. Jones, A. K., "The Object Model: A Conceptual Tool for Structuring Software," *Operating Systems: An Advanced Course,* Springer-Verlag, New York, pp. 7-16, 1979.
8. Wilkes, C. Thomas, *Preliminary Aeolus Reference Manual,* Technical Report GIT-ICS-85/07, Georgia Institute of Technology, Atlanta, Georgia, 1985.
9. LeBlanc, Richard J. and C. Thomas Wilkes, "Systems Programming with Objects and Actions," *Proceedings of the Fifth International Conference on Distributed Computing,* Denver, 1984.
10. Bloom, B. H., "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM,* No. 13, Vol. 7, pp. 422-426, July 1970.
11. Gray, J. N., "Notes on Data Base Operating Systems," *Operating Systems: An Advanced Course,* Ed. by R. Bayer, R. M. Graham, and G. Seegmuller, Springer-Verlag, Berlin, 393-481, 1979.
12. Lampson, B. W. and H. E. Sturgis, *Crash Recovery in a Distributed Storage System,* unpublished paper, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California, 1979.
13. Pitts, David V., *Storage Management for a Reliable Decentralized Operating System,* Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, in preparation.

# Kernel Structures for Clouds *

Technical Report
GIT-ICS-84/09

March 1984
Last Revision: May 14, 1984

Eugene H. Spafford

Martin S. McKendry

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

* This research is funded in part by NASA grant NAG-1-430
and by NSF grant DCR-8316590

## 1. Introduction

In the past few years, a great deal of research has been focused on the potential benefits of distributed systems. In particular, a distributed system offers the potential of a fault-tolerant computing environment. A distributed system also suggests increased computing power through the combination and application of resources. The presence of multiple machines, however, raises many questions relating to communication, consistency, reliability, configuration, and user interfaces, to name just a few. These questions are difficult to address, and that is perhaps the reason why so few attempts have been made to construct actual distributed systems. Interesting recent work in this area includes the *Eden* project at the University of Washington (e.g., [Alme83]), the *Argus* project at MIT (e.g., [Lisk83] and [Weih83]), the *Accent* system at CMU ([Rash81]), and the *ISIS* project at Cornell ([Birm84]).

The *Clouds* project is an approach to the construction and application of a distributed system that is intended to address these questions. We support the "room full of computers" view of distribution. In this view, the user sees a single resource, despite physical distinctions. In our research approach, this is achieved by constructing a highly-transparent multicomputer operating system with low-level support for maintaining consistent data items. A *multicomputer* or *computer cluster* is a system of many computers joined into one large system. The system's distribution is *transparent* to users and to most operating system components in the sense that the user is not aware of the nature or number of machines which compose the multicomputer. The user's data and processes may be distributed throughout the multicomputer system, or they all may be located on one processor -- there is no observable difference to the user, nor is there any need for the user to be aware of the configuration. We support this transparency during *upward configuration* -- the addition of more machines, and during *downward reconfiguration* -- the removal or failure of machines.

*Clouds* supports abstract data objects at a very low level. These objects are used to build the operating system and applications. Some of these objects may be made *recoverable* (operations on those objects may be undone or reversed in the event of failure or error). *Atomic transactions* or *actions* are used by both the operating system and user applications to maintain consistency and recoverability of data and operations. The design makes use of actions and objects to provide reliable operating system services, such as job schedulers, and thus provide a fault-tolerant system.

The principles and motivations behind the *Clouds* project have been described in more depth in several documents ([McKe83], [McKe84], [Allc83a]). The authors assume that the reader is already acquainted with the *Clouds* project and is somewhat familiar with the goals outlined in those documents. This paper is intended to be an introduction to the internal structures of the *Clouds* kernel. We will be constructing an experimental *Clouds* system during the next few years using dedicated minicomputers and personal computers. Further description of the *Clouds* kernel will be done as this experimental system continues to be designed and constructed ([Spaf84], [Spaf85], [Pitt85]).

## 2. Basic Assumptions

### 2.1 Terminology and Logical Structure

The term *computer* is used in this paper to mean a physically-discrete computer. Each computer supports an instance of the *Clouds sub-kernel*. The sub-kernels implement the *Clouds* virtual machine on each computer. The copies of the sub-kernels communicate with each other and together

form the *kernel* for the system. The operating system itself is implemented above the kernel, and applications are programmed above the operating system.

Figure 2.1 illustrates this logical hierarchy of levels.

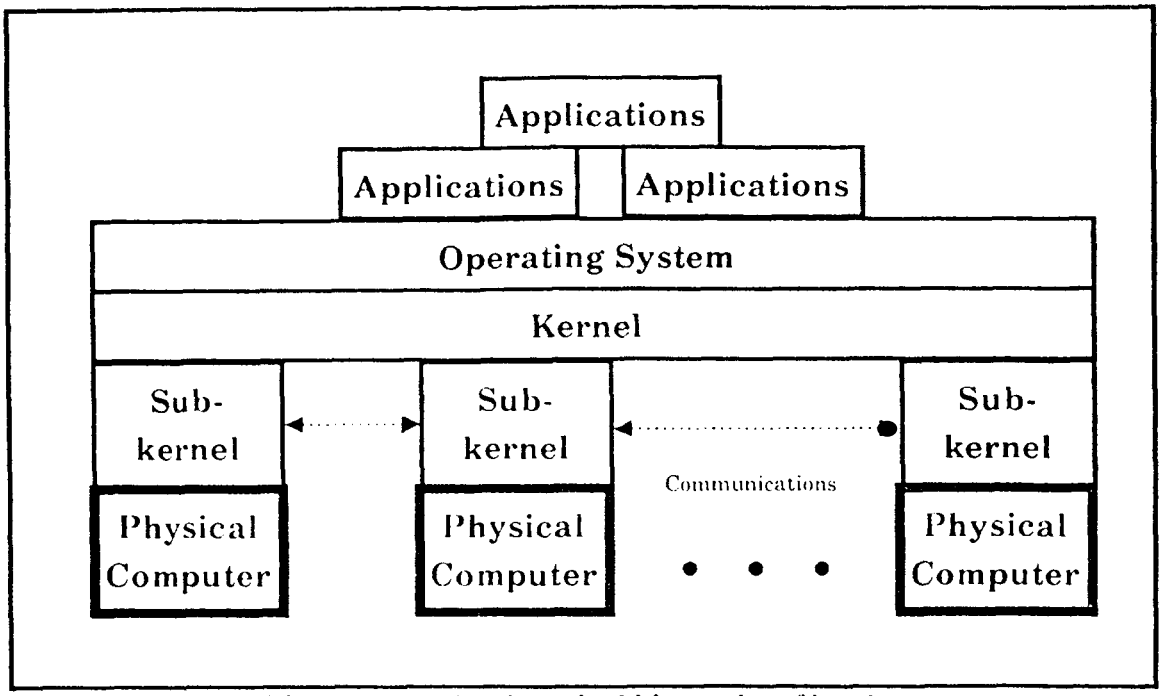



**Figure 2.1 -- Clouds logical hierarchy of levels**

### 2.2 Hardware Structure

The *Clouds* multicomputer is composed of a number of minicomputers connected by one or more communications paths, and accessed by "intelligent" terminals. The minicomputers are likely to be in close physical proximity, while the terminals may be somewhat more distant, but will still be within 1-2 km. Although isolation of a single machine is possible (a *trivial partition*), we anticipate that the probability of a general partition (disruption of communication so as to form functioning, but isolated groups of processors) will be small.

Our first prototype system will consist of three or four Vax 11/750 processors connected together by a fault tolerant 70Mb/sec bus. These systems will also be connected by a 10Mb/sec Ethernet, and possibly through dual-ported disks. A number of IBM PC microcomputers will also be connected to the Ethernet and will serve as intelligent terminals. Figure 2.2 illustrates the connections.

### 2.3 Access rights, names, and capabilitites

The system references objects with unique identifiers. Each item is referenced uniquely via a 32 bit quantity known as a *sysname*. The sysnames are unique in time and space -- non-identical items have different names. Sysnames are composed of a node ID and a sequence number, which together form the *birthmark*. The sequence number is composed of two fields -- the *subsequence number*, and the *crash count*. The subsequence field is incremented for each new name request. The crash count is incremented each time a node is restarted, and each time that the subsequence field overflows. The

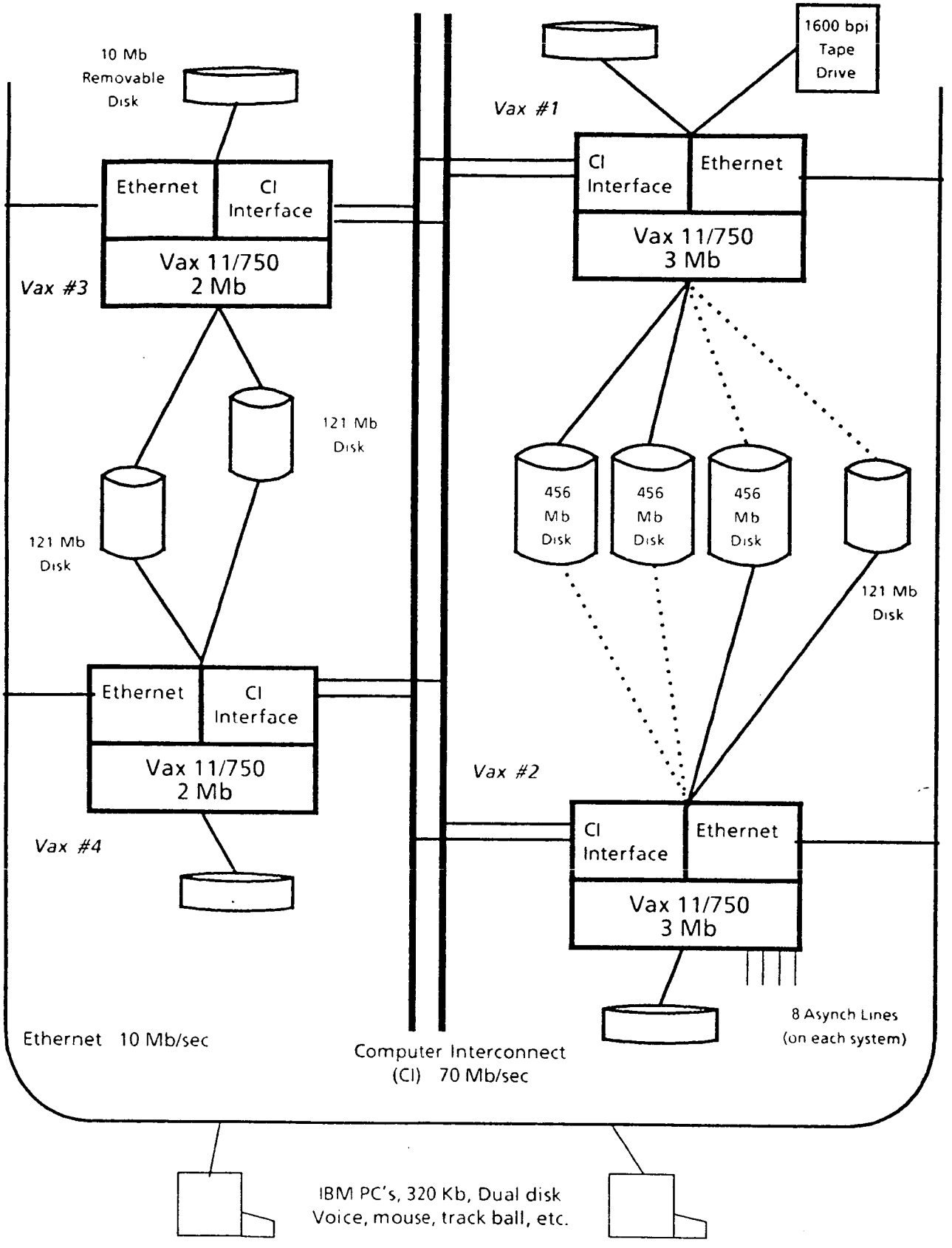



Figure 2.2 -- Prototype Configuration

sysname also contains a field which helps determine the type of the item referenced by the sysname -- a system procedure, a user-defined object (referred to as a *client object)*, a process, and so on.

Objects are referenced via object capabilities. An object capability is a 64 bit value consisting of the sysname of the object instance being referenced, and a 32 bit capability mask defining the access rights to the object. Each bit set in the mask indicates an operation that can be executed by the holder of the capability (those operations being present in the object). (Refer to figure 2.3.) Items being referenced by the system have *implied* access rights for certain kernel operations. These *implied* rights always allow the kernel to invoke the operations, but the ability to invoke those operations cannot be passed to user processes.

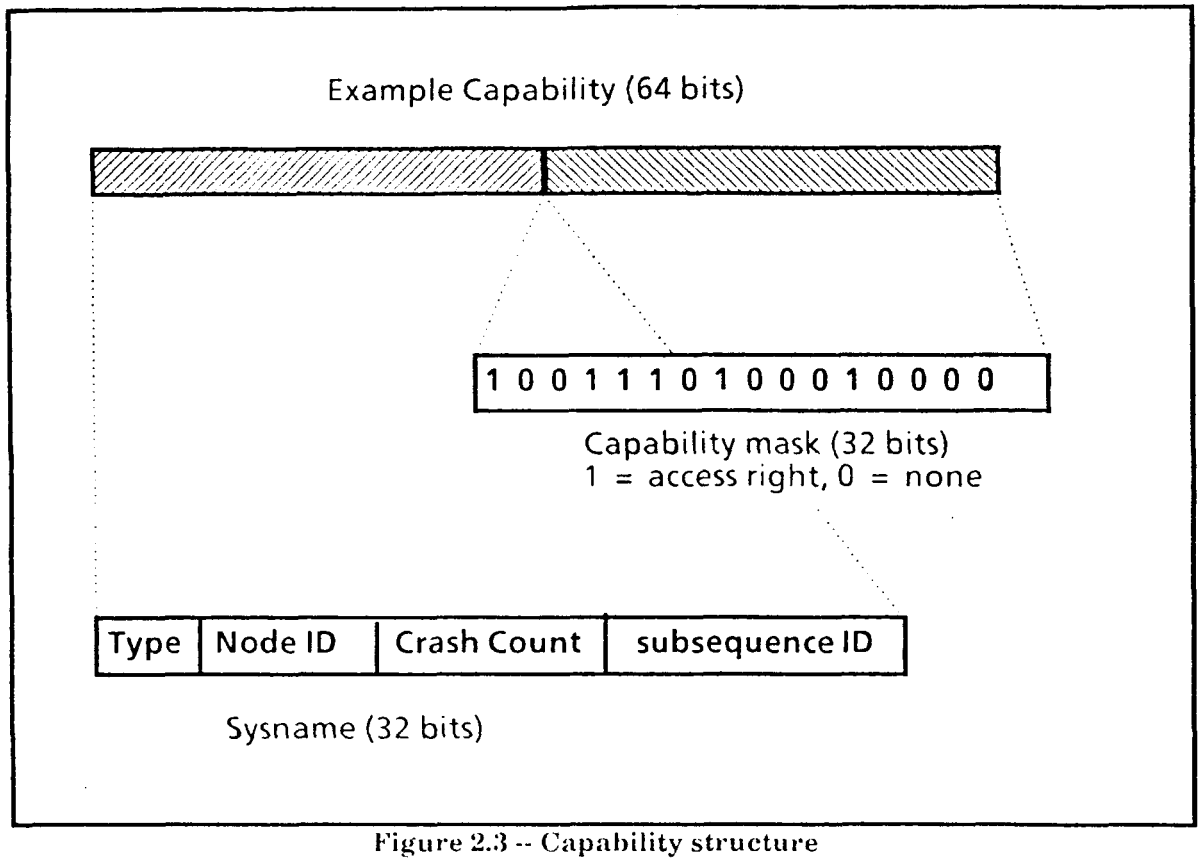



**Figure 2.3 -- Capability structure**

The existence of objects implies other capabilities. Any reference to an object also implies a reference to the type manager for the object (described in the next section), and to the system object manager. The kernel has an implied capability consisting of the sysname and all access rights. This implied capability may be used to invoke certain operations (e.g., abort, initialize), and it may be used when managing the segment in which an object resides. These implied capabilities and references are all used internally by the kernel and their existence is not seen by application software.

## 3. Objects

Objects are the fundamental data abstraction in Clouds. The rationale behind their use, and specifically their use in Clouds, has been described in other papers (e.g., [Allc83a], [Allc83b],

[McKe83], [Jone79]). Objects in Clouds are *passive* unlike Ejects in Eden [Almc83]; there are no processes resident inside the object. This section describes the structure of objects and the manner in which they are created, deleted, and in which operations are invoked.

### 3.1 Structure (types and managers)

Objects can be viewed as consisting of two parts: an object type manager and an object instance (the "type" and "instance," respectively). The type manager consists of procedure code which is allowed to manipulate the object during creation and deletion of the instance, a template of the uninitialized instance, and certain other bits of information used by the kernel in object management. The type manager operations are invoked to create and delete instances of the object type, move instances of the object from one location to another, modify existing instances, and other related operations.

The type managers are objects in their own right; their associated type manager is part of the kernel and is known as the *object manager*. There is an object manager in each sub-kernel which communicates with all of the other object managers in the system. Figure 3.1 illustrates the logical relationships amongst objects instances, object types, and object managers.

Object instances consist of data comprising the object, procedure code which operates on the data when operations on the object instance are invoked, access and modification information, synchronization variables (if appropriate), and other information related to the object instance. As an optimization, the procedure code for object instances can be stored in one single location (e.g., in the type manager) and shared by all of the active instances. The object can be thought of as composed of the code (which may include action-oriented operations such as *commit* and *abort*), permanent data, and volatile data (such as heaps or stacks) which disappears when the object is not in use (see figure 3.2).

Each instance and each type is stored as a segment (see section 6). When an operation is invoked on an object instance, the kernel maps the code for the type manager and the instance into the virtual address space of the invoking process. The object managers and segment system are responsible for finding objects when presented with the capability, and with mapping those objects into virtual memory space. This will be discussed in more depth in section 3.4.

### 3.2 Object Creation

To create a new object, the user first describes the object type using an appropriate applications programming language. This forms a template of the data structures which compose the data portion of the object. The user also codes functions which operate on this data. These functions import and export arguments by value only; reference parameters are simulated by passing capabilities by value.

The procedural part of the object definition may contain special routines for synchronization of access to the object, support of atomic actions, and initialization of new instances of the object. These routines may be derived from a standard system library, or the user may program them specifically. Every type manager contains functions for creating a new object instance, deleting an old object instance, and for initializing a new object instance. Each recoverable object must also contain functions to implement beginning of action (BOA), abort, precommit, and commit.

Once the object implementation is written, it is compiled by the appropriate system compiler(s). The result will be a file of code and data which is passed to the system object manager via a call through the kernel interface. The object manager will change the type of the file to "type manager" and will return an object capability to the new type manager.

To create an instance of an object, the user invokes the "create" operation on the type manager. The calling process also specifies a storage partition (cf) where the permanent version of the object

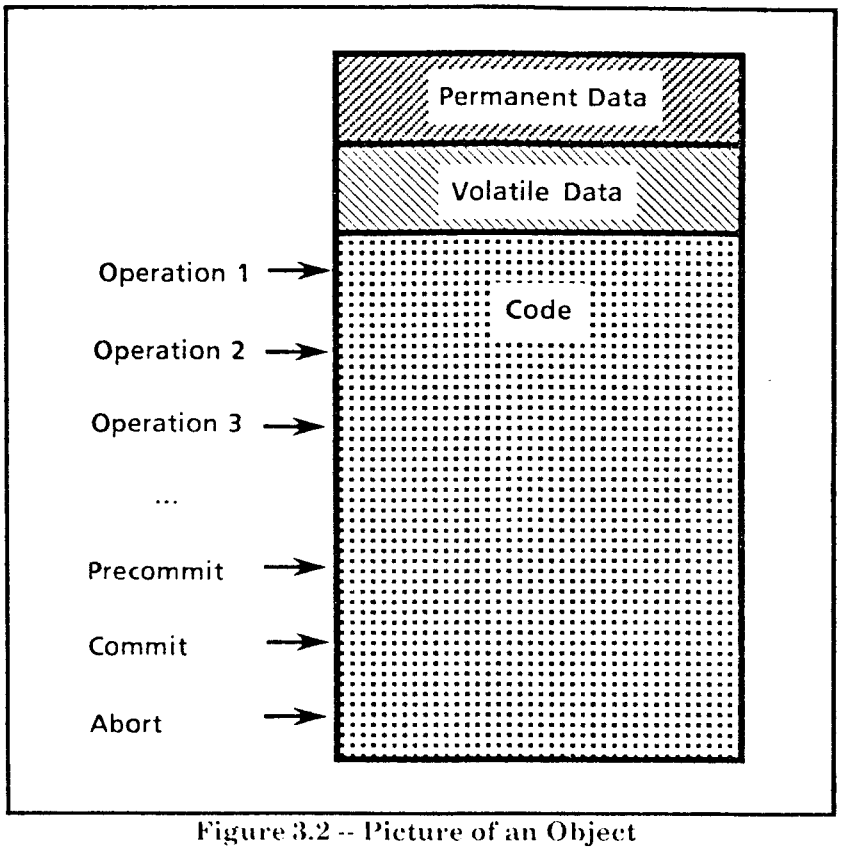



**Figure 3.2 -- Picture of an Object**

instance is to reside. The type manager uses this partition capability to create a new segment of type "client object." The sysname returned by the segment system is used by the type manager to create the capability returned to the user. The type manager also initializes the object capability mask so that all defined operations are enabled (e.g., if there are only 10 operations defined on the object, the remaining 22 bits in the mask are not set).

Once the type manager has created a segment, it uses its data template and "init" function to initialize the new object instance. This usually involves setting some data values to initial states, initializing the heap for the object, resetting and defining synchronization variables, and creating a skeleton VAM (Virtual Address Map -- see section 5) which will be used to map the object instance into a process's address space when needed.

When everything has been updated, the type manager returns the newly created capability to the caller. Future references to the object instance will all be through this capability. The type manager also increments an internal instance count which may later be used when deleting the type manager (deleting the type manager while there are outstanding instances results in those instances becoming "orphans" which cannot be used); the instance count is always an advisory value and no guarantee is made about its accuracy.

### 3.3 Deleting Objects

To delete an object instance, the user invokes the "delete" operation contained in the type manager for the object. This is an operation on the capability to the object instance. Each object contains a pointer to the type manager for the object instance, and this pointer is used to find the type manager and

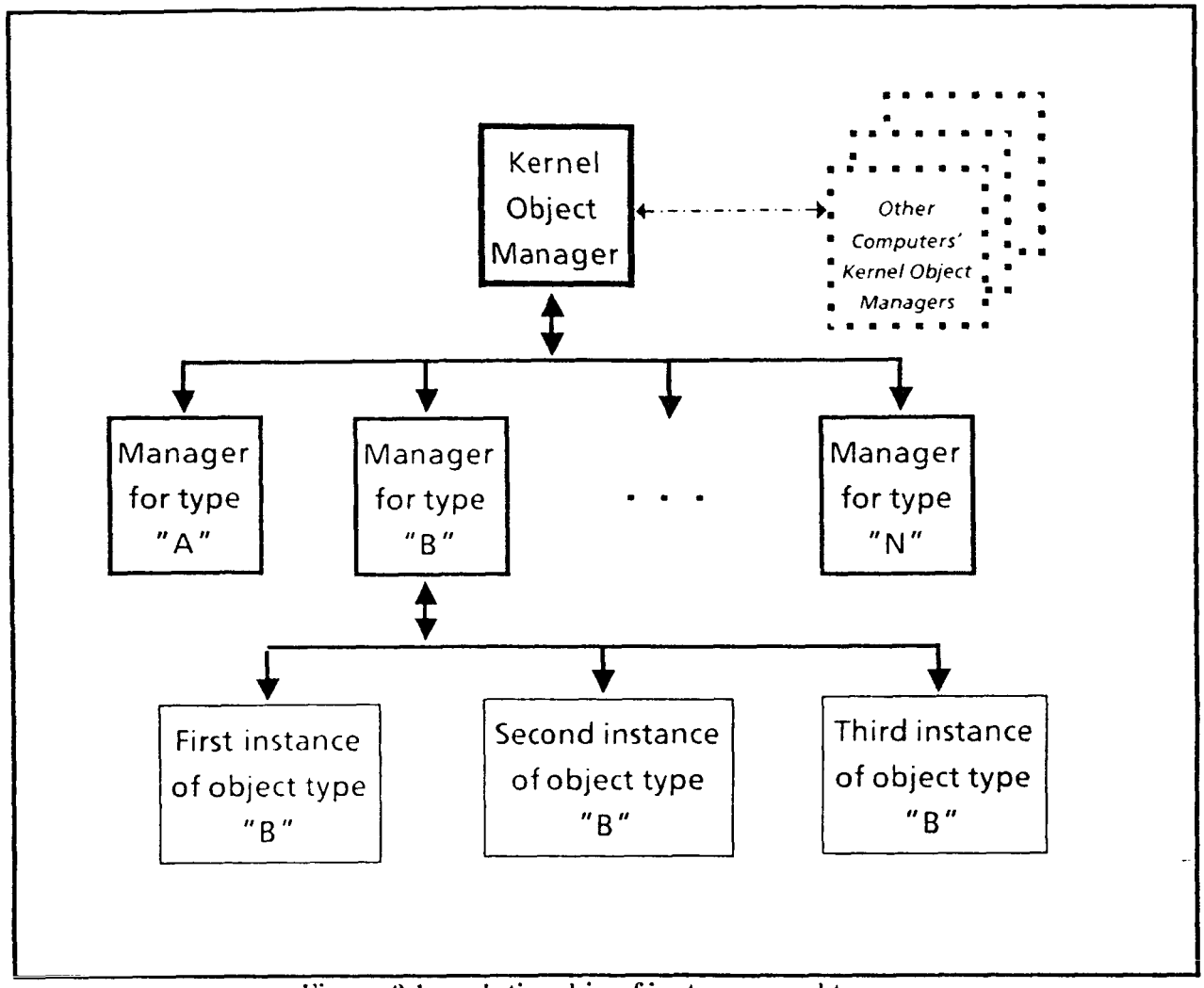



**Figure 3.1 -- relationship of instances and types**

invoke the delete operation; this is an example of the use of an "implied" capability. To delete an object instance the user must have the "delete" right, however.

Type managers are treated exactly like objects if the "delete" operation is performed on them. The system object manager acts as the type manager for all user-defined type managers, and it is responsible for performing the necessary delete operations. Note that the object manager has the option of checking the reference count inside the type manager and issuing some warning to the user if it is non-zero. That is, it is possible to advise the user if that type manager might still have outstanding instances in existence.

### 3.4 Invoking operations on instances

An operation is invoked on an instance by making a call on the kernel with the capability to the object instance, the operation number, and (optionally) a list of parameters to the operation. The kernel object manager will first verify that the operation number is in range and that the capability mask

contains the required access rights. If the attempted access is not valid, the process or action attempting the operation is aborted (to prevent security violations and help contain errors).

Next, the segment system is presented with the object capability and requested to map the instance into memory. If the instance name is currently known to the segment system, then a mapping is already known. Otherwise, the segment system searches the directory of every local active partition. If the instance is found, then the mapping is performed and the segment is now "known" to the segment manager; future references to the segment will be to the segment on the local partition. This search mechanism is described in more detail in section 8 and in [Pitt85].

If the segment is not found locally, the segment system invokes a search module which attempts to locate a copy of the segment elsewhere in the system. If unsuccessful, then the current action is aborted since there is no way of deciding whether the desired object is locked, never existed, or simply cannot be found due to some failure in the system. If successful, the reference causes the object to be made "active" on the system where it is found. From this point on, references to the instance will continue to be mapped through the search module. Future references will be mapped to a system where the instance is active.

Once the segment containing the object instance is made active, the segment containing the object is read to find the name of the type manager for the instance. The type manager is then mapped into memory in exactly the same manner as the instance was mapped. The object instance may contain a "hint" which can be used to speed this mapping: a hint can indicate the site where the type manager was found when the object instance last referenced it.

After the instance has been mapped into the process's virtual address space, the operation is invoked with the appropriate parameters. Additionally, a capability to the object control block for the object is stored in the "current object" field of the process control block for use by the kernel, if necessary.

### 3.5 Cloning Objects

For enhanced availability and speed of access, it may be practical to have redundant copies of type managers located at different spots around the system. Thus, if a machine goes down and it makes a type manager inaccessible, it may be possible to map future references to a copy of the type manager located on a still-active machine; these copies are referred to as *clones*.

Currently, only immutable objects may be cloned. This means that only type managers may be cloned, since they contain unchanging code and data templates. The object manager has a *copy* operation which can be invoked to clone a type manager.

References to cloned type managers still occur through the standard capability mapping mechanism. All clones will have the exact same name and are completely interchangeable. Any reference can be mapped to any available type manager.

When a type manager is cloned, its reference count field is marked as "not valid" both in the original and in the clone. When a delete operation is performed on the type manager, one of the clones will be deleted. The application code may choose to warn the user that the type manager is a cloned object and there may be other copies in existence. The code may further note that there may be undeleted object instances which could be rendered useless by removal of the last clone.

# 4. Processes and Actions

## 4.1 Processes

The basic instrument of activity is the process. A Clouds process is similar to processes used in other systems. Each process represents an identifiable sequence of operations. Each process is represented by a unique *process control block* or *PCB*. The PCB for a process is used to store the contents of the machine registers when the process is not active. The PCB also holds pointers to the process's stacks, pointers to the structures currently defining the virtual memory space of the process, and a pointer to the object control block of the object that the process is currently accessing, if any.

A process is created when the kernel process manager receives a request for a new process. Each request specifies an activity that the newly-created process is to begin, and a pointer to an activity-specific block of parameters for the process to use. The process manager allocates space in its internal store for a new PCB and a new set of memory maps (see section 5 for more details). The PCB and memory maps are initialized, and the process stacks are also initialized. The system scheduler object is invoked with a capability to the process and the process is added to the ready list for subsequent dispatch and activation. A capability for the process is also returned to the activity which requested the creation of the process. That capability can be used to halt or kill the process, change its priority, or modify other operating characteristics of the process.

When a process has completed its assigned tasks it returns to the kernel process manager and its PCB and memory maps are reclaimed for use in building new processes. The process may also be halted and reclaimed by the process manager upon request. Such a request must contain a capability for the process and have the necessary access rights.

There is only one process running on a machine at any one time. The other processes in the system at that machine are either linked into the *ready list* or they are linked into the *wait list* associated with some synchronization item. The ready list is a list of processes awaiting a turn at the processor. The ready list is maintained in a scheduler object according to the scheduler's queuing algorithms. The process dispatching mechanism for the virtual machine invokes operations on this scheduler module to obtain the next ready process or to enqueue a process (make it ready). The scheduler object is not, strictly speaking, a part of the kernel. There may be one scheduler for many or all of the machines in a Clouds system, or there might be might be one per machine each using a different queuing discipline on their ready lists. This enables the system to change scheduler modules and disciplines without stopping so as to adjust to changing configurations and workloads.

The kernel supports traditional counting semaphores, single and multi-mode locks, event tickets, and timed events as means of synchronization. The process manager can create new instances of each of these items upon request. Each synchronization item is associated with a wait list that is a list of processes blocked on that variable. When a process blocks on one of these items it is removed from the ready list and linked into the end of the wait list associated with the item. A pointer to the synchronization variable is placed into the PCB of the process for later reference. When a process is unblocked, it is unlinked from the wait list and added to the ready list via an invocation of the scheduler.

## 4.2 Actions

Actions are sequences of operations which occur *atomically*. That is, to an outside observer, the operations performed by an action occur all at once or not at all. Actions in Clouds have been discussed extensively in [Alle83]. Briefly, in Clouds, actions are related to processes in that the operations associated with an action must be performed by one or more processes. The kernel object

manager is responsible for maintaining information about actions. It keeps a record of all objects visited by each action during the lifespan of the action. It handles the commit and abort protocols associated with actions. It is also responsible for ensuring that only processes operating in the context of actions are allowed to touch recoverable objects.

When the object manager receives a request for a new action it first obtains a new worker process from the process manager. It enters the sysname of the process into an *active action descriptor*, and assigns an *action name* to the process. That action name is copied into the PCB of the process, and is used to form a new capability to the action. This capability is then passed back to the requester. The requester is not given the capability for the process; all requests to the process manager are made on the action and through the object manager.

While an action is active, the object manager notes every recoverable object touched by the action and enters the sysnames of those objects into the action descriptor. Should the action abort itself or be aborted by some other process holding a capability to the action, the object manager will (eventually) invoke the "abort" operation on each recoverable object listed in the action descriptor. When the action attempts to precommit or commit, the object manager will invoke the corresponding operations on each of the objects listed in the action descriptor. Once an action aborts or commits, its action descriptor is deallocated and the process manager is called to reclaim any processes associated with the action; by definition, an action can perform no more operations after it has aborted or committed.

The synchronization of actions and multiple processes acting on behalf of the same action is left entirely to each object accessed and to the object manager. The kernel provides no direct support for deadlock handling. These can be resolved through the use of timeouts -- each request to create an action is accompanied by a value specifying a maximum completion time. If the action does not complete within that time interval then it is aborted by the object manager.

Clouds also supports *subactions*. These act like actions in virtually every respect except that their effects are not permanent until all of their top-level ancestor actions commit. An action is not allowed to commit until all of its subactions have completed by aborting or committing. When an action aborts, all of its subactions are also aborted. The algorithms defining the interaction of actions and subactions are described in [Allc83], as are the algorithms for dealing with various kinds of network and communications failures.

# 5. Virtual Memory

## 5.1 Mapping Virtual Memory

The virtual memory system maintains the binding of physical memory locations to references made by executing processes. In particular, we are interested in the mapping of locations inside *Clouds* objects in such a manner that we can invoke the operations on objects and modify object instances. Additionally, we wish to support shared objects, an embedded kernel, and efficient management of storage coupled with correct operation of the action/object mechanisms. These constraints have made a major impact on the form of the underlying virtual memory structure. The resultant structure bears some surface similarities to the virtual memory structure of systems such as HYDRA [Wulf74] and Accent [Rash81].

Before describing the structures used to support the virtual memory system, we present the address space as seen by a typical process, and describe the utilization of that space. The virtual address space can be thought of as composed of three portions -- object space, per-process space, and system kernel space (see figure 5.1). The object space contains the code and data associated with the currently active object; this may include executable code from the object type, data items and heaps associated with

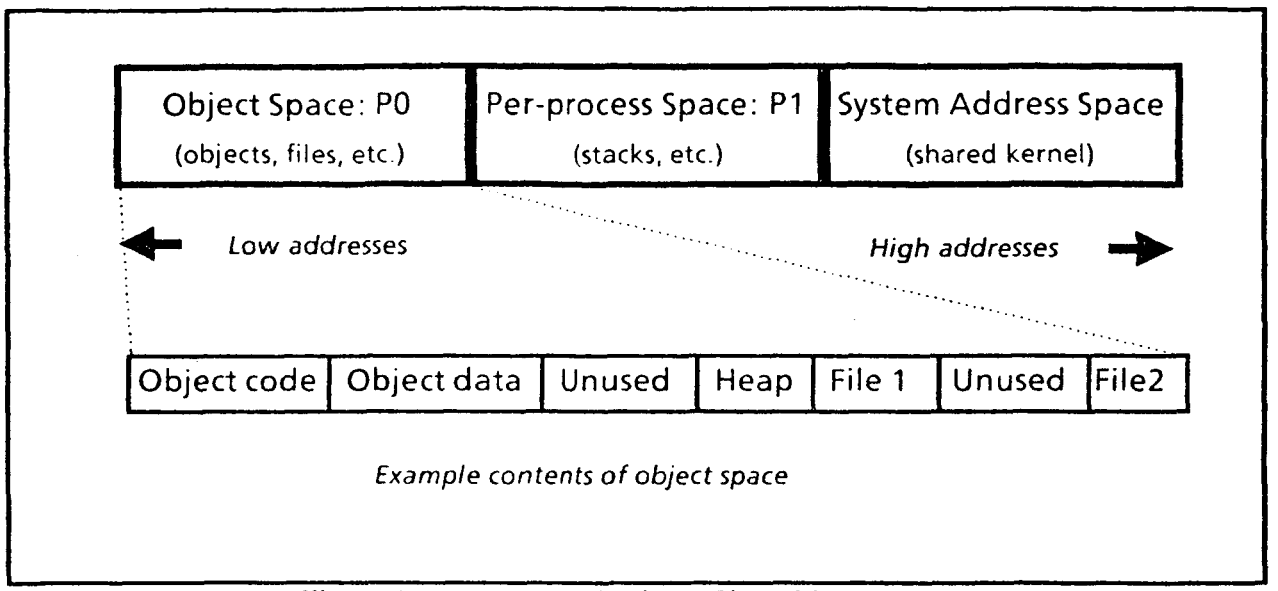



**Figure 5.1 -- A process's view of its address space**

the instance of the object, and one or more "windows" into files being referenced by the active object. On the Vax, this space corresponds to the P0 page map.

The per-process space contains items that survive object invocations and returns. That is, it contains items which are not specifically associated with any active objects but are instead associated with the active process. This space contains all of the user stacks and static data not associated with any object. On the Vax, this space corresponds to the P1 page map.

The system kernel space is the same for every process. Shared items such as the virtual memory tables and process control blocks reside in the system space. The system space is also where most of the code which implements the kernel resides. On the Vax, this corresponds to the area mapped by the system page map.

Each of these address spaces is mapped in a similar manner. Refer to figure 5.2 for the following discussion of common features.

Contiguous, related addresses in a process' address space are mapped together as one "chunk." For example, all the executable code in an object type could be mapped as one chunk, as could an entire user stack, or the entire contents of a file. Each "chunk" is represented as an entry in a *Virtual Address Map* or *VAM*. There is one VAM describing system space, one unique VAM for each process' per-process space, and one VAM for each active object. These VAMs are referenced, respectively, by a pointer in the system control block, by a pointer in each PCB (process control block), and by a pointer in each OCB (object control block).

Each VAM has a header which points to a hardware page map. This pointer is loaded into the current process' page map register to effect the mapping indicated. That is, the page map table (PMT), contains the hardware defined entries necessary to define the virtual memory space in the physical page frames available.

The VAM header also contains a count (up to 8 in the initial implementation) of valid VAM entries following the header. Each VAM entry defines a "chunk" of the virtual address space. A VAM entry

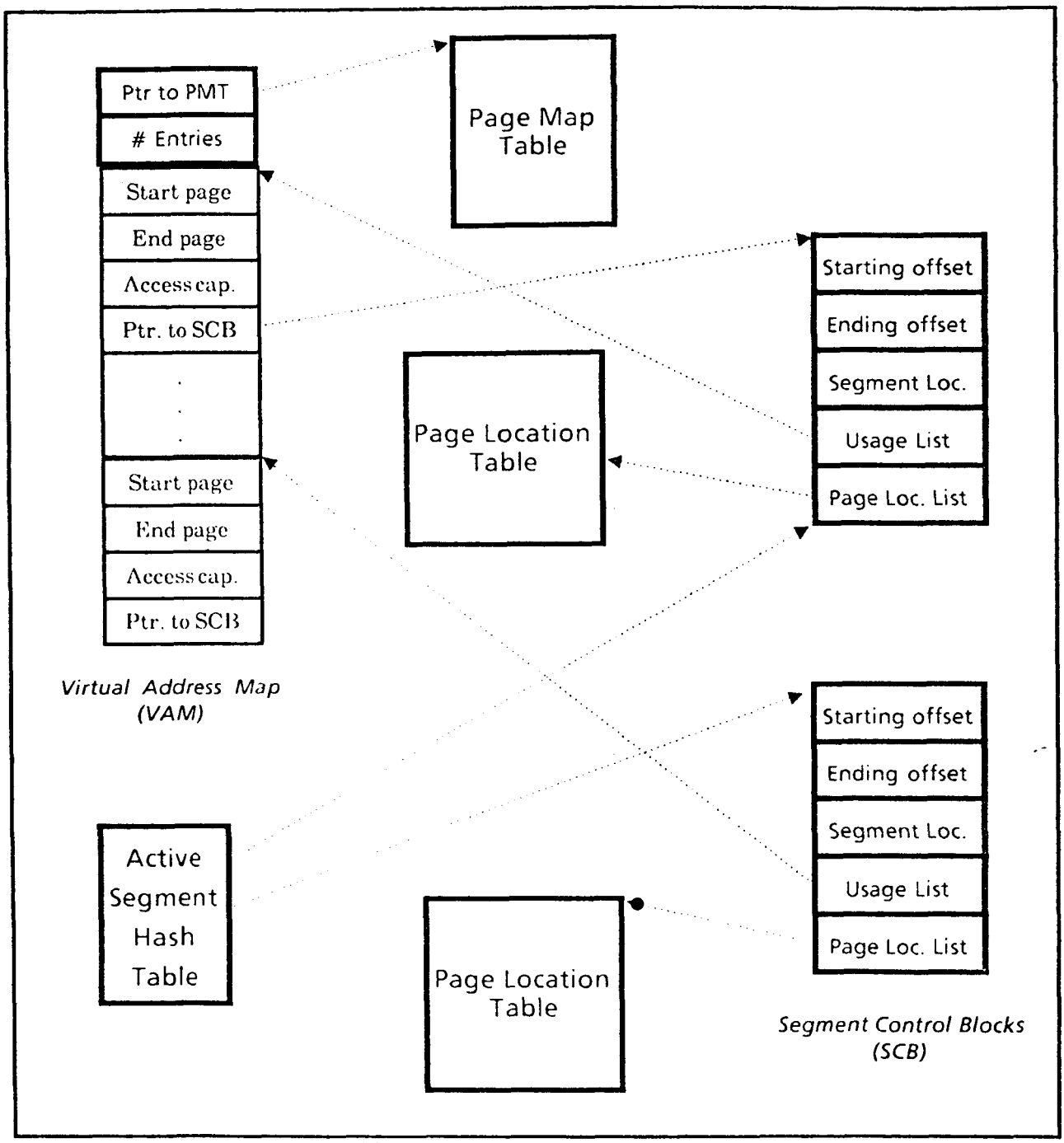



**Figure 5.2 -- Common Features**

contains the starting and ending page numbers of the process address space described by the entry. It also contains a pointer to a *segment control block* or *SCB*, and the capabilities that this process/object has in relation to that segment (e.g., read-only, read/write, delete, etc.).

Each SCB defines a currently active segment (segments are described more fully in the next section). The SCB indicates the portion of the real segment which is mapped. For instance, if the process is

accessing only a few pages of a multi-megabyte file, there is no need to map the whole file segment into the address space. Instead, the SCB would contain the beginning and ending page numbers of the section that was being accessed. (The VAM entry would indicate the addresses within the virtual address space where that segment would be mapped.)

The SCB also contains pointers to all VAM entries referencing the segment (the usage list), and a pointer to a *page location list* or *PLL*. There is one entry in the PLL for each page described by the SCB. Each PLL entry indicates the status of that segment page (one of: resident & locked, resident & unlocked, being brought in (in-transit), being removed (out-transit), on the pre-page list, non-resident and in the source partition image, non resident and on the paging partition, or not yet defined). The PLL entry also contains a location which is used in combination with the status to find non-resident pages and remove resident pages. The frame number of resident pages is also indicated in this list.

The relationship of these structures to one another may be made clearer by presenting a brief scenerio of handling a page fault. Suppose an active process gets a page fault in its P0 (object) space. The page fault handling code locates the appropriate VAM by using the pointer located in the OCB of the currently active object (as noted above, a pointer to the current OCB is in the PCB of the current process). Next, the fault code compares the page number of the fault with the ranges presented in each VAM entry. When the appropriate entry is found, its pointer is used to locate the SCB which describes the missing page. The offset from the starting page number in the VAM entry determines the relative offset in the segment which is needed.

The pointer to the page location list in the SCB is used next. The fault code uses the relative offset to index into the PLL and obtain an entry corresponding to the missing page. Further action is determined by the current status of the missing page:

the page is resident. This implies that the page was being brought in at the time of the fault and has since arrived, or else the page is also in use by some other process and the presence of the page has not yet been indicated in this process's page map table. In either case, the frame number is taken from the PLL and inserted into the proper place in the PMT pointed to by the VAM.

the page is undefined. This implies unintialized data space, such as in a heap or stack. An empty frame is filled with zeros (for error containment and security), and its frame number is placed into the appropriate places within the PLL and the PMT.

the page is non-resident. Based on the location information in the PLL, and the segment information present in the SCB, a request is made to read the missing page into an empty page frame. The status of the page in the PLL is changed to "in-transit" and the process waits until the page arrives.

the page is pre-paged. This implies that the page was added to the pre-page list as a candidate for removal. The page is removed from the pre-page list and its status is changed back to "resident." The PMT is updated appropriately.

the page is in-transit. This implies that some other process has already requested the page. The process waits until it arrives; the requesting process will perform the mapping and this process will awaken to find the page resident.

the page is out-transit. This implies that the page is currently being written out to secondary storage to free the frame in which it was residing. Nothing can be done about reading the page back in until a stabilized image is present on the secondary storage. Therefore, the process waits until the transfer out of memory is complete. When the process is awakened, it will mark the page

as resident and not modified since the resident version corresponds to the version in secondary storage, and then continue, or it will bring the page back in.

In each case, locks are used to ensure consistent results during concurrent accesses.

### 5.2 Managing Physical Page Frames

In order to efficiently provide empty page frames to satisfy page faults, it is necessary to keep track of the state and use of each page frame. This is accomplished through the use of the *physical page table* or *PPT*. The PPT is organized as an array with the index of each element corresponding to a physical page frame; PPT entry 5 corresponds to frame 5, and so on.

Each entry in the PPT contains information about the status of the page frame, and links to other page frames in the same state. This is accomplished by putting forward and backward link fields in each PPT entry; each link corresponds to the index of the next (or last) PPT entry in a doubly-linked chain of similar entries. There are four such chains threaded through the PPT: the active frame list, the pre-page frame list, the free frame list, and the locked frame list.

The active frame list contains entries referring to page frames which are currently occupied and in use. As pages are brought in in response to page faults, they are added to the end of this list. Each entry in this field also has a pointer to an SCB describing the page, and an offset field which can be used to locate the page's entry in the PLL associated with the SCB.

The free frame list is simply a list of currently available page frames. Requests for empty page frames are satisfied with the entries in this list. We assume that the list is never empty; if necessary, we will suspend all other processes and run a page reclaimation process to keep the number of free pages above a minimum threshold.

The locked frame list contains entries corresponding to pages which cannot be thrown out of memory. This includes pages involving active device I/O, pages containing critical code or data (like the PPT!), and pages which are being kept from paging due to performance considerations.

The pre-page list contains entries which are candidates for removal from memory , thus freeing those frames. The page reclamation process will remove entries from the head of the active frame list and add them to the tail of the pre-page list, while at the same time tracing down all PMTs which reference this page and marking it as nonresident. Pages are removed from the head of this list and added to the free list after their contents have been written out to secondary storage, if necessary. Pages referenced before they reach the head of the pre-page list get reactivated and moved back onto the active frame list. This whole mechanism implements a form of FINUFO (First In, Not Used, First Out) paging algorithm. (Note: if the Vax hardware supported a "referenced" bit in its page tables, this could be avoided!)

## 6. Secondary Storage, Partitions and the Segment System

### 6.1 General Structure and Terms

In order to present a consistent, uniform means for the kernel to reference items requiring storage, every item on the system is viewed as having a second type known as *segment*. Segments are basically untyped units of storage which can be read, written, copied, deleted, and moved by kernel code. Paging, whether of a file, a process space, or an object, is always accomplished with the same set

of segment operations. Copying of items from one place to another is always accomplished with segment operations, and so on.

A segment is *active* if it has been recently accessed or was present in memory before being referenced. The SCB for the segment indicates which processes are referencing it, and which partition driver (see below) needs to be invoked for operations on the segment. These SCBs are allocated and initialized whenever a segment is made active (i.e., referenced and not currently active). The *Active Segment Hash Table* hashes segment capabilities into pointers to the corresponding SCBs.

Each segment representing a permanent item has a version resident on some *partition*. A partition corresponds to some block of space available on a secondary storage device. In general, a partition could exist on a disk, on a tape, or in a block of special memory. Usually, a partition will be blocks of space on disk device(s).

A segment on a partition is composed of a *segment header* and a data area. The segment header contains information about the segment, such as the defined type of the segment (e.g., *client object* or *file*), a capability to the type instance for this segment, time of creation, and other such information. The data space is the actual contents of the segment.

Partitions are composed of a partition header which describes the partition (record size, extent, etc.), a free list whose format is partition-dependent, a directory, directory entries, and data records containing the segments. This structure is more fully described below.

Segments are *recoverable, non-recoverable, or temporary (volatile)*. Recoverable segments represent items which may be accessed only by actions, while non-recoverable and temporary segments may be accessed by actions or processes. Operations on objects of type *segment* are generally not available to application processes but are always available to kernel code. Any process with the correct capabilities may read any object, whether it is represented as a non-recoverable or recoverable segment. Temporary segments "disappear" on machine crashes and are used for paging space and volatile data structures.

Because of the constraints necessary to ensure the recoverability of objects and to support atomic actions accessing those objects, partitions are not allowed to cross device boundaries. That is, each partition must be fully contained within one device. It is possible to locate the directory and free list for a partition on a device different from the directory entries and data, but this adds a great deal of complexity and delay in the operation of the partition.

Volatile and recoverable segments can be mixed freely within a partition, but some partitions will not be capable of supporting recoverable segements and will be so marked in the partition header. For instance, partitions on a tape are not able to support recoverable objects. In general, a partition supporting recoverable entries must reside on a device which:

1) allows random reads and writes;

2) does not perform internal buffering of writes except by application choice;

3) provides atomic single-record writes.

Condition 3, above, may be relaxed by use of replicated writes to other devices; duplication of writes to implement stable storage is a standard method when dealing with potentially unstable devices and critical applications [Lamp81].

Partitions are added to the system by *mounting*. The operation to mount a partition involves mapping a partition driver to a device driver and scheduler (which may already be active with another partition on the same device). This relationship is shown in figure 6.1. The device driver

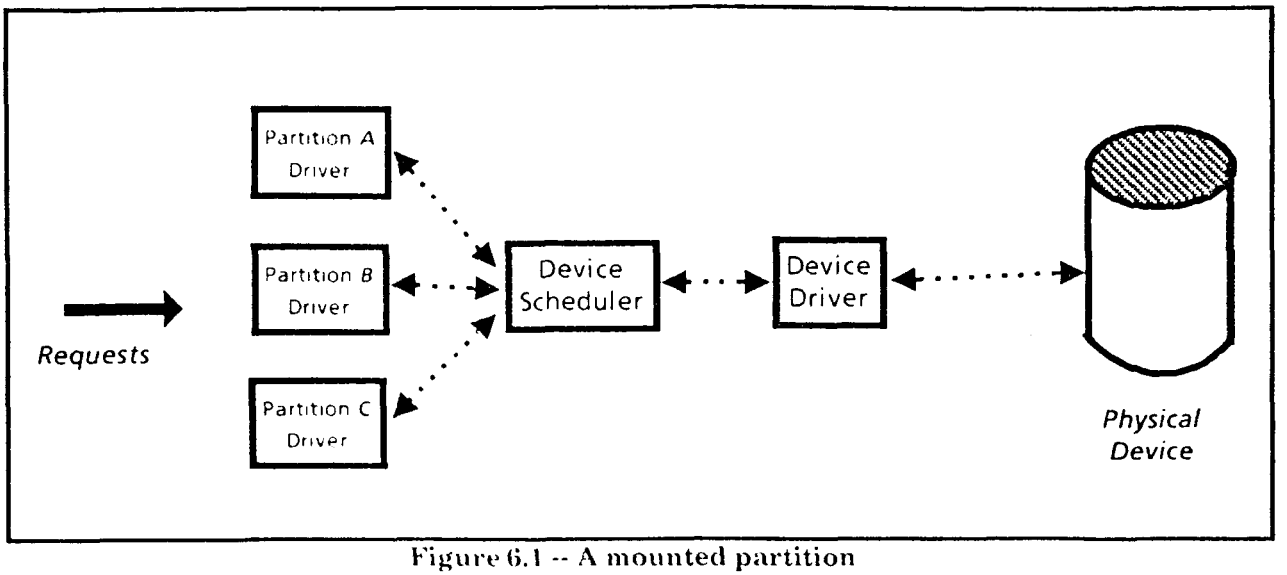



**Figure 6.1 -- A mounted partition**

module is actually a process which performs operations on the device according to requests provided by the device scheduler module. The driver code is written to take full advantage of individual device characteristics and requirements. This may include implementation of scatter/gather operations, automatic retries of erroneous reads, and so on.

The device scheduler module is designed to take requests from partition drivers and then provide them to the device driver in some orderly, efficient manner. The scheduler module may also contain space to provide buffered reads and writes, and manage these buffers.

Each partition driver accepts requests to read or write a record of a segment in the partition. These requests are then mapped into the appropriate operations on the free list and directory structure, and reformatted into request packets to the device scheduler. The partition driver maps the segment capability into an absolute address in the device based on the segment offset, directory entry, and partition header. Requests to partition drivers may also get modified due to shadowing and recovery considerations. Some requests such as commits and deletes may require that the free list and directory be read or written as well.

Every partition supports operations to *create* a segment, *delete* a segment, *alter* a segment (change its type or maximum allowable size), *read* a page from a segment, *write* a page to a segment, and *truncate* (shorten the working size) of the segment. Each segment is created with a maximum allowable size beyond which it is not allowed to grow. Operations to create a segment or alter a segment require an appropriate capability to the partition as well as the segment involved.

### 6.2 Recoverable Segments

In addition to objects, partitions need to support recoverability for directory structures and the free lists used in the management of the partition itself. The Clouds storage management organization achieves this goal of recoverability by embedding portions of the free list and directory management in the management of the recoverable segments. This system also allows the shadowing and paging of recoverable objects.

Consider the partition logical organization as shown in figure 6.2. The partition header is always

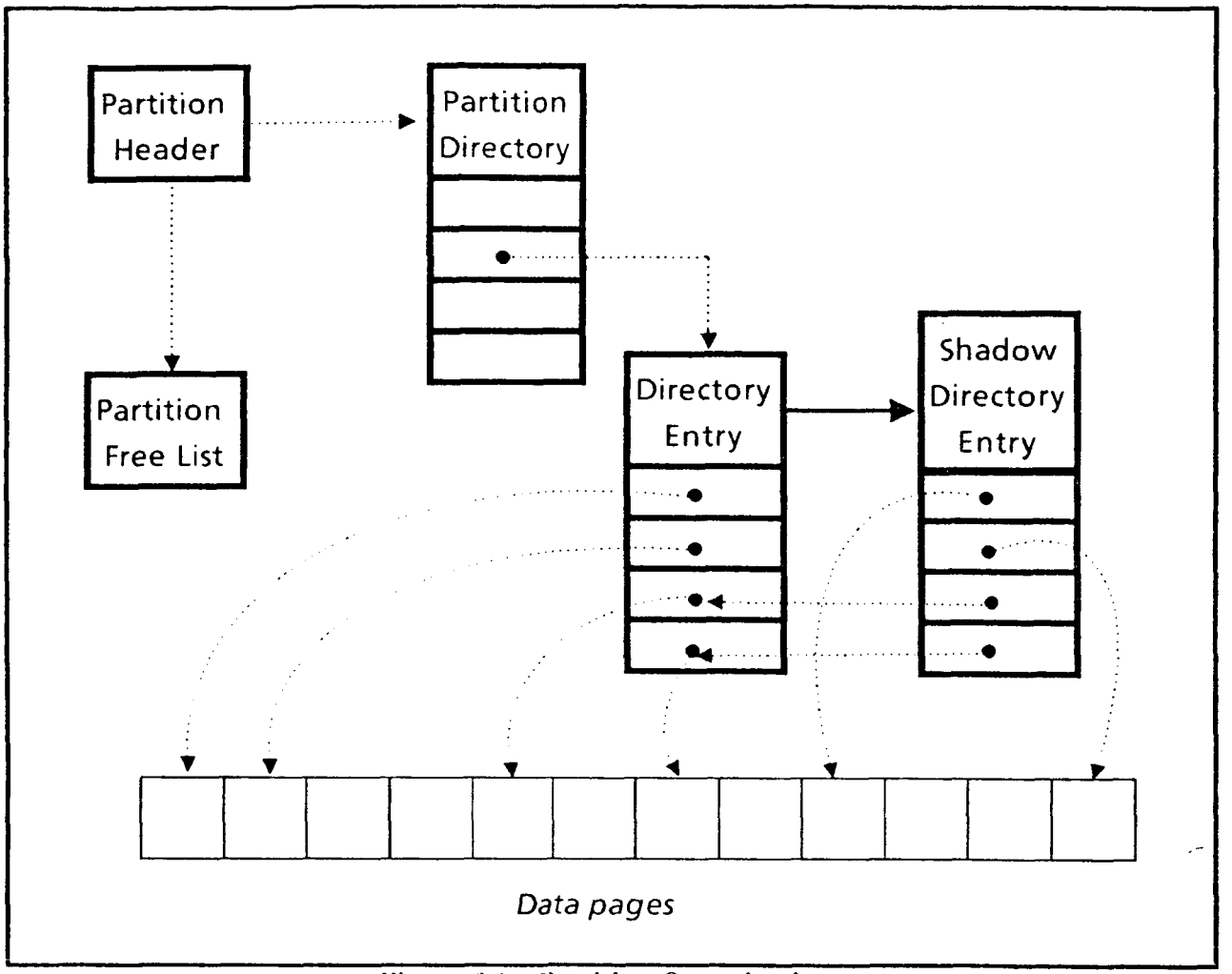



**Figure 6.2 -- Partition Organization**

found in record zero of the partition. The header contains information about where to find the free list and directory for the partition, and whether the partition supports recoverable segments. The exact form of the free list is not really important as long as it is possible to map it into memory; a bit map is probably the most reasonable form for a disk partition.

The partition directory consists of segment names (sysnames) and the record number within the partition which contains the directory entry for that segment. The partition directory may be many records long, and could be organized as some form of sorted tree rather than as a simple list. To simplify the discussion, we assume here that the directory is simply a linear list of name/address pairs.

Each directory entry consists of name and type information, and a list of the records which make up the segment. This list could also be many records long. Each entry in the list consists of a flag field and a record address which points to the corresponding data record within the segment. In addition to this information, the header contains a field which could contain a pointer to a shadow version of the segment.

When the partition is mounted, the code for the partition driver checks the directory entries to discover uncommitted and precommitted entries (as described shortly). If any such entries are found, their processing is completed. Then a volatile, in-memory copy of the free list is made. The partition directory may also be processed to provide a list of names of objects present on this partition.

When a read reference is made to a recoverable segment in this partition, it is only necessary to use the directory entry to locate the appropriate data page and read it into memory. If the reference is a write reference, then it is necessary to shadow the segment until the referencing action either commits or aborts. Note that in the following description the operations undertaken are all done internal to the partition driver and the calling action never sees anything other than a consistent view of the object.

When the first write to a recoverable segment is done by an action, the partition driver makes a copy of the directory entry on the partition. Each data record pointer in this shadow entry is initially the same as the corresponding pointer in the permanent version. Subsequent writes will be done to newly allocated records containing copies of the corresponding permanent data record. The shadow directory entry is changed to point to these new shadow data pages. After the shadow is created, the permanent version directory entry is changed to include a pointer to the shadow and a status flag indicating that it is being shadowed. If a crash occurs anytime between this point and the time the shadow is committed, the code which checks the directory during partition mount will discover the shadow and remove the pointer.

The records allocated for the shadow segment are all taken from the volatile free list held in memory. The actual free list on the partition is not updated except during the actual commit of the shadow, so any failure simply loses the volatile free list and the records occupied by the shadow remain marked as free in the permanent free list on the partition.

If an abort occurs it is necessary to remove the pointer in the directory entry for the permanent version of the object and mark all of the records of the shadow as "free" in the volatile free list. When a precommit occurs, the status flag in the permanent version is updated from "being shadowed" to "precommitted shadow present." If a crash occurs before a commit or abort, the code which mounts the partition will discover this precommitted segment and then determine whether to complete the commit or abort based on information from other machines in the net.

When a commit occurs, the status of the shadow is changed to "permanent" and its shadow pointer is set to point to the old permanent version (this can be done in a single record write). Next, the partition directory is written to point to the new permanent version. Next, the permanent free list is updated to indicate that the records used in the new permanent version are unavailable and that the records used by the old permanent version are free; the volatile version in memory is likewise updated. Lastly, the pointer in the directory entry for the new permanent version is set to null. If a crash occurs at any point in this processing the commit can be continued from the point of failure when the partition is next mounted.

This scheme assumes that single record writes to the partition are atomic (i.e., uncorruptable during a crash). It also assumes that crashes occur infrequently and that some extra processing when a partition is mounted will not be a difficulty. Using this scheme, aborts and precommits require a single record write. Creation of the shadow requires two record writes in addition to the writes to the data portion of the segment. Commits require approximately three record writes beyond those normally needed to update the free list. Some of these writes may be buffered without affecting the resulting consistency. Thus, this provides an efficient means of supporting recoverable segments and partition structures. Similar algorithms exist for creating and deleting recoverable segments, and for operating on non-recoverable partitions. All of these algorithms are discussed in more depth in [Spaf84] and [Spaf85].

# 7. Network Communications and RPC

## 7.1 Network communications

Each machine in the Clouds multicomputer may be connected to one or more other machines in the system by any number and kind of communication channel. The nature of the connections is also immaterial, although it should be obvious that a minimum number of connections of sufficient bandwidth will enhance ultimate performance. The prototype system will have the individual processors connected by a common high-speed backbone and by an Ethernet. Multiple Ethernets and asynchronous communications lines may also be accommodated.

Communication between individual subkernels is handled by a replicated *communications manager*, one per subkernel. Each subkernel makes communications requests through the single interface presented by its copy of the communications manager. Based on the destination and nature of the request, the communications manager chooses the communications channel(s) over which to send the message. The communications manager determines a transmission channel based on message size, message priority, current communications configuration and error counts, and load information. A fixed-size header is prepended to the data portion of the message, and the communication is transmitted over the appropriate medium.

Note that all applications, including the RPC mechanism, define their own protocol for message traffic. The communications manager provides facilities for acknowledging receipt of messages, and for receiving (possibly prioritized) replies to messages. Other than the fixed header block prepended to each message, the communications manager makes no interpretation of the data portion of any message. This allows the operating system and applications software to make direct use of the communications system in whatever manner best suits them.

There is no guarantee that messages are delivered by this communications system to the appropriate software on a remote node. Acknowledgements must be done at a higher level, if desired; the communications driver only supports hardware-level acknowledgements. There are also no guarantees of delivery order or assurances against duplication during transmission. The only assumptions being made in the design of the Clouds communications system are:

- -- that it is extremely unlikely that the network will be partitioned into isolated segments;
- -- that if a message arrives at a machine, it is possible to determine if it arrived uncorrupted;
- -- messages arrive at their destination in small, finite time or else are lost forever;
- -- the overall probability of corrupted or lost messages is small.

Incoming messages are delivered to the communications manager by individual device drivers and signalled via device interrupts. The communications manager determines the nature of the incoming message and passes it to the appropriate object or process within the subkernel for further action. The communications manager does no protocol checking whatsoever other than determining that the message arrived intact (usually indicated by a hardware status code) and that the message was actually destined for this subkernel. It may use the header information present in each message to update its own internal tables.

The communications manager is made aware of communications channels via an *activate* operation similar to a partition *mount* operation. The communications manager is invoked with a capability to the physical device driver for the channel, and a set of parameters which describe the speed and possible connectivity of that channel. This information is used to help the manager determine when to use that channel for communications. As messages go out and come in over that channel, that information is updated within the internal tables of the communications manager. The first message sent out over an activated channel is a "I'm here, who's there?" message. This is to inform other

systems of the availability of communications over this path, and the responses ellicited are used to identify potential destinations for future messages.

### 7.2 Remote Procedure Calls

One of the operations available on the communications manager is that of the remote procedure call on an object. Normally, this operation is invoked by the object manager when an invocation is attempted on an object which is not present on this machine. The object manager therefore reformats the attempt into an RPC operation on the communications manager. A capability to the object, an operation number and a pointer to a parameter block (usually just the current stack frame of the caller) are the arguments to the call on the communications manager. The communications manager constructs a message consisting of the capability and operation number, and a copy of the parameters to the object (remember that all parameters in object invocations are passed by value). This message is then sent out over the appropriate communications channel(s) and the calling process is blocked until a reply is received. A unique identifier is associated with the message to identify the response.

When the RPC message is received at the site where the object currently resides, the communications manager requests a *cohort* process from the process manager. The cohort is dispatched with a copy of the object capability and operation number, and with a pointer to the copy of the parameter block which comprised the remainder of the message. If the request was on behalf of an action, the cohort assumes the action identity of the calling action via a call to the object manager. Next, the cohort copies the parameter block to its stack and invokes the object as if the call had originated locally with the cohort. When the invocation returns to the cohort, it calls the communications manager to formulate a reply to the original request. The reply is constructed with the values and error codes returned by the invocation, if any, and with the unique identifier provided with the incoming RPC message. This reply message is then sent out and the cohort is reclaimed by the process manager.

When the reply message is received at the original machine, the communications manager alters the state of the blocked requesting process to indicate the location of the reply message, and then the process is unblocked. It retrieves the reply information, alters its stack and registers as indicated by the reply, and then returns. The whole process of call and return looks exactly like a local (although possibly slow) invocation.

## 8. Object Searching and Invocation

One of the most important features of objects in Clouds is that every invocation embodies an implicit search. No assumptions are made about the location of objects. In fact, it is entirely possible that objects may move from machine to machine between invocations. The implicit search also allows more dynamic use of cloned objects and alternate communications pathways.

When an object is invoked, the object manager searches the Active Object Table for an entry matching the name of the object. If such an entry is not found, then a new entry is allocated and added with a set of default values. If the name is found in the table, then the entry contains a pointer to one of two things: an OCB (Object Control Block) or a search module. If the pointer is to an OCB, then the object is present on this machine, and is either currently active or was just recently active. In either case, the OCB contains all of the necessary information to bring the object into memory and map it into the virtual address space of the requesting process.

If the entry for the object is a pointer to the default (initial) search module, then all of the local partitions are searched to see if the object is present locally. If it is found on a local partition, then an OCB is allocated and initialized and the invocation proceeds as above. If the object is not found

locally, then the pointer is altered to point to the network search module and that module is then activated.

The network search module uses available information about the configuration of the network and the available communication paths to seek the object on some set of remote machines. This search operation may actually be done as a "search and perform" operation to save time, with the parameters for the RPC included with the search message. If a remote site receives such a search message, it attempts to find the object locally. If the object is found, it is made active locally and the operation is performed as a normal RPC.

If the network search module fails to locate the object on any remote machine it returns a simple "not found" indicator to the requesting process, aborting it if it is an action. It is not possible to determine at this point whether the object does not exist, exists on a processor which is currently not communicating with this machine, or whether the object is currently not available due to action visibility constraints.

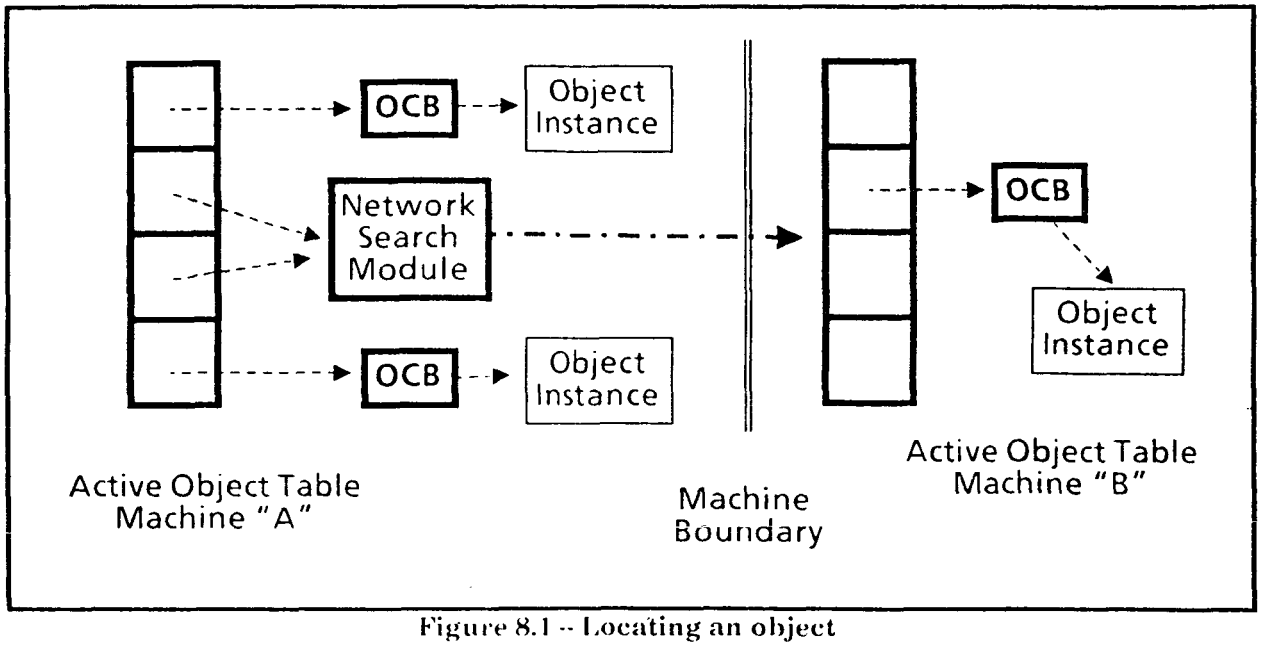



**Figure 8.1 -- Locating an object**

The search strategy may well be optimized somewhat through the use of hints. Once an object has been found and made active it is possible to include a hint with the entry in the active object table on the calling machine. Future references to the object can be tried first on the hinted-at machine since that is the last known location of the object. It may also be possible to derive hints in other ways; this area will be the subject of further research by the Clouds group. [Pitts85]

## 9. Kernel Interface

Conceptually, the kernel interface extends across machine boundaries and exists on each processor within the Clouds system. In actuality, the interface is replicated on each machine. In our prototype, the interface consists of service calls which change the processor mode to the kernel state and then examine the arguments for validity. Kernel calls (including object invocation) use a protected per-

user stack to hold return state information. Calls by code within the kernel to other parts of the kernel are done directly and avoid the overhead of a system trap.

In general, each kernel operation requires one or more capability parameters specifying what is to be done. The kernel interface maps those capabilities to simpler operations to be performed by specific subkernels. Most kernel operations are done by actions dispatched by the interface. This allows better error containment and prevents kernel operations from being only partially completed. The actions so spawned operate independently of the requesting process or action. This mechanism helps simplify the design of the distributed aspects of the kernel, especially when dealing with kernel services being performed on remote systems which may possibly fail in the midst of the operation.

As an example, consider a request to the kernel to move a file from one partition to another. The request to the kernel would include a capability to the file and a capability to the destination partition. The request does not require the specification of any specific machine names or locations. The kernel will locate the file and parition based on the provided capabilities. The movement of the file will occur as part of an action, with the copy being done with segment operations to read and write pages of the file. Should the communications channel or one of the processors fail during the transfer, the action will be aborted and the partially transferred file will be erased from the destination partition. If the destination partition does not have enough space for the file, the action will abort and the space will be freed. Other errors act in a similar manner with appropriate error codes being returned to the caller, if possible.

This method of structuring the interface also allows the system to be expanded to other processors which may employ a different underlying architecture. As long as it is possible to create an action somewhere in the Clouds system then the calls through the kernel interface can be attempted at some location; the operation of the kernel interface is independent of the location of the requester since all location information is contained within the capabilitites passed as arguments.

## 10. Conclusion

This paper has presented an overview of the internal structure of the Clouds kernel. This presentation has also given an indication of how these structures will interact in the prototype Clouds implementation. Many specific details have yet to be determined and await experimentation with an actual working system.

## 11. References

[Allc83a] Allchin, J. E., "An Architecture for Reliable Decentralized Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also released as technical report GIT-ICS-83/23)

[Allc83b] Allchin, J. E., and M. S. McKendry, "Synchronization and Recovery of Actions," Proceedings of the 2nd *ACM SIGACT-SIGOPS* Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983

[Almc83] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," University of Washington Department of Computer Science, Technical Report 83-10-05, October 1983

[Birm84] Birman, K., *et.al.*, "Implementing Fault-Tolerant Distributed Objects," Cornell University Computer Science technical Report 84-594, March 1984

[Jone79] Jones, A. K., "The Object Model: A Conceptual Tool for Structuring Software," Operating Systems: An Advanced Course, Springer-Verlag, NY, 1979, pp. 7-16

[Lamp81] Lampson, B. W., "Atomic Transactions," Distributed Systems: Architecture and Implementation, Springer-Verlag, NY, 1981, pp. 246-265

[Lisk83] Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS*, Vol. 5, No. 3, July 1983

[McKe83] McKendry, M. S., J. E. Allchin, and W. C. Thibault, "Architecture for a Global Operating System," IEEE Infocom, April 1983

[McKe84] McKendry, M. S., "Clouds: A Fault-Tolerant Distributed Operating System," IEEE Distributed Processing Technical Committee Newsletter, 1984

[Pitt85] Pitts, D. V., "Naming and Searching in a Distributed Operating System," PhD Thesis, School of Information and Computer Science, Georgia Institute of Technology, *in progress -- 1985*

[Rash81] Rashid, R. F., and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," Proceedings of the 8th Symposium on Operating Systems Principles (ACM SIGOPS), December 1981

[Spaf84] Spafford, E. H., "A Consistent File System for Clouds," Technical Report, School of Information and Computer Science, Georgia Institute of Technology, *in progress -- 1984*

[Spaf85] Spafford, E. H., "Kernel Structures for a Distributed Operating System," PhD Thesis, School of Information and Computer Science, Georgia Institute of Technology, *in progress -- 1985*

[Weih83] Weihl, W. and B. Liskov, "Specification and Implementation of Reilient, Atomic Data Types," Symposium on Programming Language Issues in Software Systems, June 1983

[Wulf74] Wulf, W., *et. al.*, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, Vol 17, No. 6, June 1974