PROJECT ADMINISTRATION DATA SHEET

[X] ORIGINAL [ ] REVISION NO. _____

Project No. G-36-636 (R-6062-0A0) GTRC ~~GIT~~ XXXX DATE 10 / 31 / 85

Project Director: ~~R. La Blanc~~ R. J. MARTIN K LeBlanc School ~~ISE~~ ICS

Sponsor: Maryland Procurement Office

Ft. Geor. Meade, MD 20755-6000

Type Agreement: Contract MDA904-86-C-5002

Award Period: From 10/01/85 To ~~09/30/86~~ 6/11/86 (Performance) 10/30/86 (Reports)

Sponsor Amount:	This Change	Total to Date
Estimated: $		$
Funded: $	83,388	$ 83,388

Cost Sharing Amount: $ N/A Cost Sharing No: N/A

Title: Fault Tolerant Distributed Computing

ADMINISTRATIVE DATA OCA Contact Ralph Grede X 4820

1) Sponsor Technical Contact:

L. G. Tarbell R-53
Maryland Procurement Office
9800 Savage Rd.
Ft. Geo. Meade, MD 20755-6000
(301) 859-6695

2) Sponsor Admin/Contractual Matters:

Ms. Ann Witt L-433
Maryland Procurement Office
9800 Savage Rd.
Ft. Geo. Meade, MD 20755-6000
(301) 859-6943

Defense Priority Rating: DO:A7 Military Security Classification: Unclassified

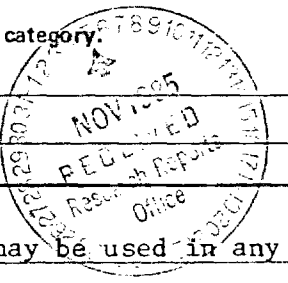
(or) Company/Industrial Proprietary: N/A

RESTRICTIONS

See Attached Government Supplemental Information Sheet for Additional Requirements.

Travel: Foreign travel must have prior approval – Contact OCA in each case. Domestic travel requires sponsor approval where total will exceed greater of $500 or 125% of approved proposal budget category.

Equipment: Title vests with Sponsor - No equipment is proposed.



COMMENTS:

When deemed necessary, the hours of effort in any classification may be used in any other direct labor classification. See Sec. B - Labor Classification.

COPIES TO: SPONSOR'S I. D. NO. 02.123.001.86.002

Project Director	Procurement GTRI Supply Services	GTRC
Research Administrative Network	Research Security Services	Library
Research Property Management	Reports Coordinator (OCA)	Project File
Accounting	Research Communications (2)	Other A. Jones

FORM OCA 65 2S5

GEORGIA INSTITUTE OF TECHNOLOGY OFFICE OF CONTRACT ADMINISTRATION

SPONSORED PROJECT TERMINATION/CLOSEOUT SHEET

2/18

Date 2/3/87

Project No. G-36-636 School/~~Lab~~ XXX ICS

Includes Subproject No.(s) N/A

Project Director(s) R. Le Blanc GTRC /~~GIT~~ XXXX

Sponsor Maryland Procurement Office, Ft. Geor. Meade, MD 20755-6000

Title Fault Tolerant Distributed Computing

Effective Completion Date: 9/30/86 (Performance) 10/30/86 (Reports)

Grant/Contract Closeout Actions Remaining:

- [ ] None
- [X] Final Invoice or Final Fiscal Report
- [X] Closing Documents
- [X] Final Report of Inventions Questionnaire To P. I.
- [X] Govt. Property Inventory & Related Certificate
- [ ] Classified Material Certificate
- [ ] Other _____

Continues Project No. _____ Continued by Project No. _____

COPIES TO:

Project Director
Research Administrative Network
Research Property Management
Accounting
Procurement/GTRI Supply Services
Research Security Services
~~Reports Coordinator (OCA)~~ XXXXXXXXXX
~~Legal Services~~ XXXXXXXXXX

Library
GTRC
~~Research Communications (RCX)~~ XXXXXXXXXX
Project File
Other Ina Lashley
Angela Jones
Russ Embry

FORM OCA 69 :85

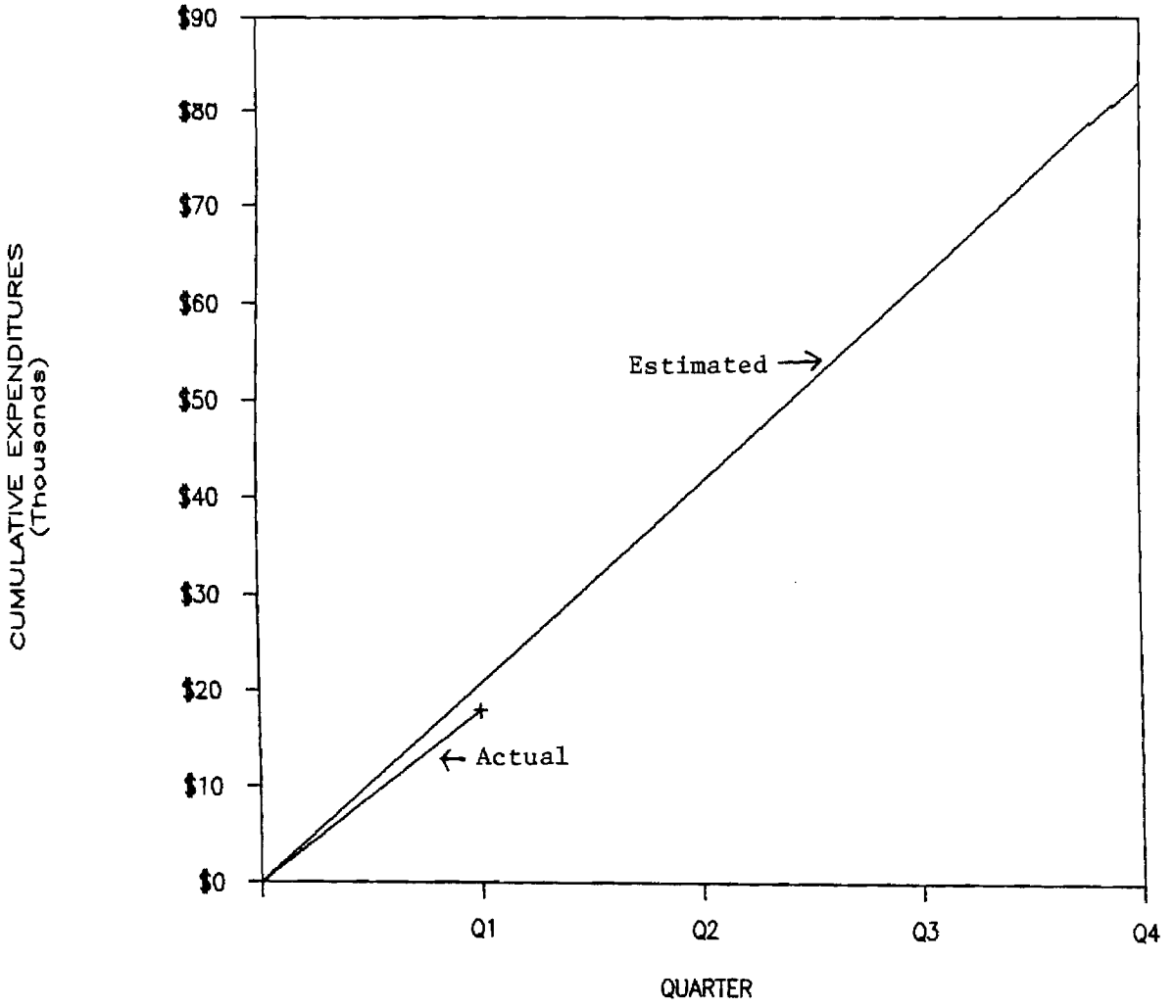
Date Prepared February 10, 1986
Contract No. MDA 904-86-C-5002
Contractor Georgia Tech Research Corp.

Summary/Work Package Title
Fault-Tolerant Distributed Computing
Report Month 10/1/85 - 12/31/85

FUNDS EXPENDITURE REPORT

Column A ORIGINAL PROPOSAL			Column B Latest Accepted Revised Proposal	Column C Reporting Quarter Expenditures	Column D Cumulative Expenditures to Date Total Man Hours	 Dollar Value	 Pct. Dollar Value	Column E Cost to Complete Estimate	Column F Latest Cost Estimate
1. Direct Labor									
Type	Number Of Hours	Hourly Rate	Dollar Total						
PI	525	$26.98	$14,164.50	$795.91	29.5	$795.91	5.6%	$13,368.59	$14,164.50
GRA	1950	$12.38	$24,142.85	$8,183.81	661	$8,183.81	33.9%	$15,959.04	$24,142.85
Clerical	260	$8.33	$2,166.67	$325.00	39	$325.00	15.0%	$1,841.67	$2,166.67
Total Direct Labor			$40,474.02	$9,304.72		$9,304.72	23.0%	$31,169.30	$40,474.02
Burden @ 21.0% (Excluding GRA Labor)			$3,429.54	$235.39		$235.39	6.9%	$3,194.15	$3,429.54
Total Direct Labor and Burden			$43,903.56	$9,540.11		$9,540.11	21.7%	$34,363.45	$43,903.56
2. TRAVEL EXPENSE			$3,200.00	$633.81		$633.81	19.8%	$2,566.19	$3,200.00
3. GENERAL & ADMINISTRATIVE EXPENSE			$1,500.00	$300.00		$300.00	20.0%	$1,200.00	$1,500.00
4. COMPUTING CHARGES			$2,300.00	$575.00		$575.00	25.0%	$1,725.00	$2,300.00
TOTAL DIRECT COSTS			$50,903.56	$11,048.92		$11,048.92	21.7%	$39,854.64	$50,903.56
5. INDIRECT COSTS @ 63.5			$32,323.76	$7,016.06		$7,016.06	21.7%	$25,307.70	$32,323.76
TOTAL CONTRACT PRICE			$83,227.33						$83,227.33
TOTAL COMMITTMENTS AND EXPENDITURES				$18,064.98		$18,064.98	21.7%		

FUNDS EXPENDITURE GRAPH



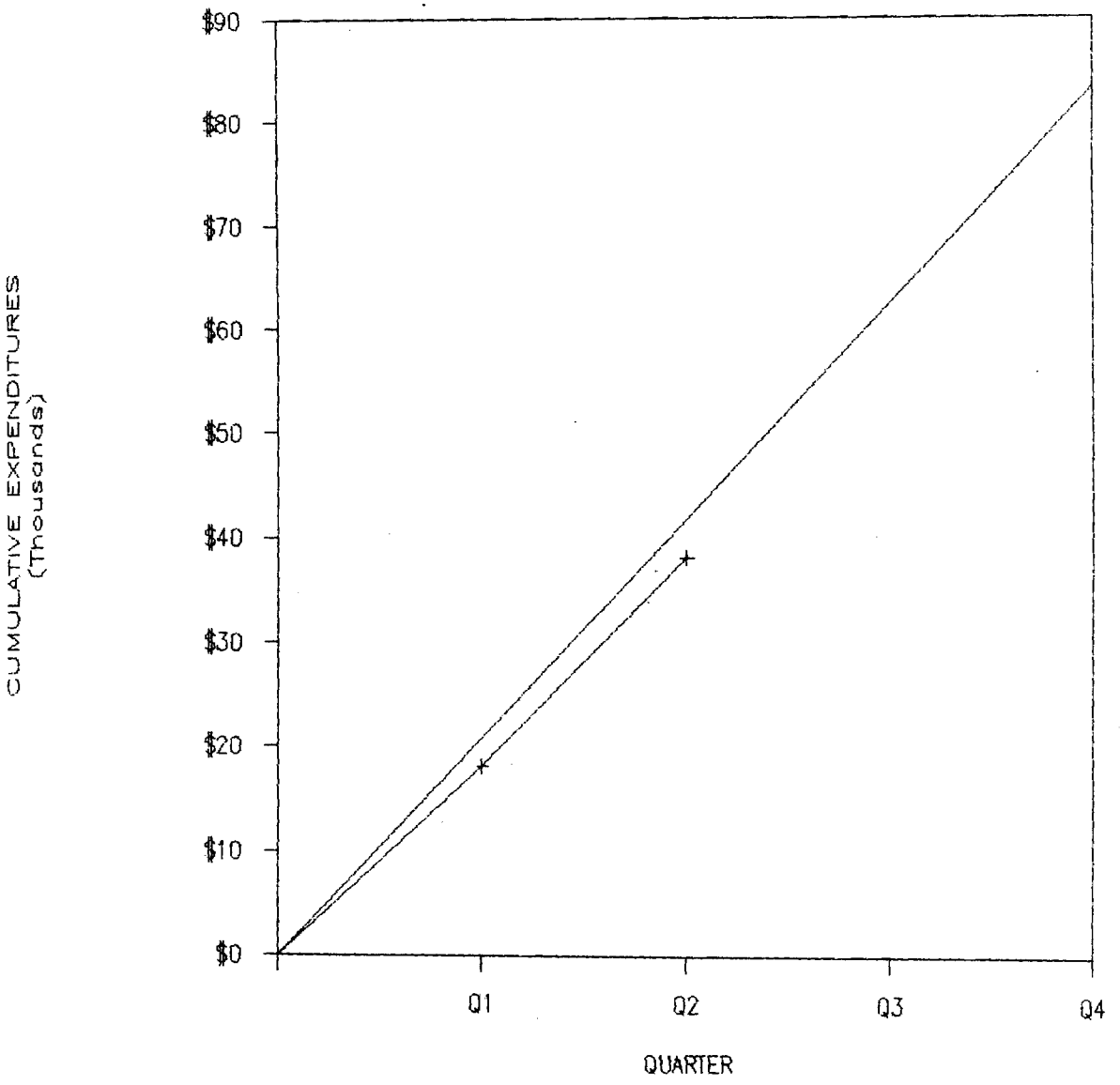
Date Prepared April 18, 1986
Contract No. MDA904-86-C-5002
Contractor Georgia Tech REsearch Corp.

Summary/Work Package Title
Fault Tolerant Distributed Computing
Report Month 1/1/86 - 3/31/86

FUNDS EXPENDITURE REPORT

Column A ORIGINAL PROPOSAL			Column B Latest Accepted Revised Proposal	Column C Reporting Quarter Expenditures	Column D Cumulative Expenditures to Date Total Man Hours	Dollar Value	Pct. Dollar Value	Column E Cost to Complete Estimate	Column F Latest Cost Estimate
1. Direct Labor									
Type	Number Of Hours	Hourly Rate	Dollar Total						
PI	525	$26.98	$14,164.50	$1,200.61	74	$1,996.52	14.1%	$12,167.98	$14,164.50
GRA	1950	$12.38	$24,142.85	$7,577.14	1273	$15,760.95	65.3%	$8,381.90	$24,142.85
Clerical	260	$8.33	$2,166.67	$1,116.67	173	$1,441.67	66.5%	$725.00	$2,166.67
Total Direct Labor			$40,474.02	$9,894.42		$19,199.14	47.4%	$21,274.88	$40,474.02
Burden @ 21.0% (Excluding GRA Labor)			$3,429.54	$486.63		$722.02	21.1%	$2,707.53	$3,429.54
Total Direct Labor and Burden			$43,903.56	$10,381.05		$19,921.15	45.4%	$23,982.41	$43,903.56
2. TRAVEL EXPENSE			$3,200.00	$0.00		$633.81	19.8%	$2,566.19	$3,200.00
3. GENERAL & ADMINISTRATIVE EXPENSE			$1,500.00	$1,200.00		$1,500.00	100.0%	$0.00	$1,500.00
4. COMPUTING CHARGES			$2,300.00	$575.00		$1,150.00	50.0%	$1,150.00	$2,300.00
TOTAL DIRECT COSTS			$50,903.56	$12,156.05		$23,204.96	45.6%	$27,698.60	$50,903.56
5. INDIRECT COSTS @ 63.5			$32,323.76	$7,719.09		$14,735.15	45.6%	$17,588.61	$32,323.76
TOTAL CONTRACT PRICE			$83,227.33						$83,227.33
TOTAL COMMITTMENTS AND EXPENDITURES				$19,875.13		$37,940.12	45.6%		

FUNDS EXPENDITURE GRAPH



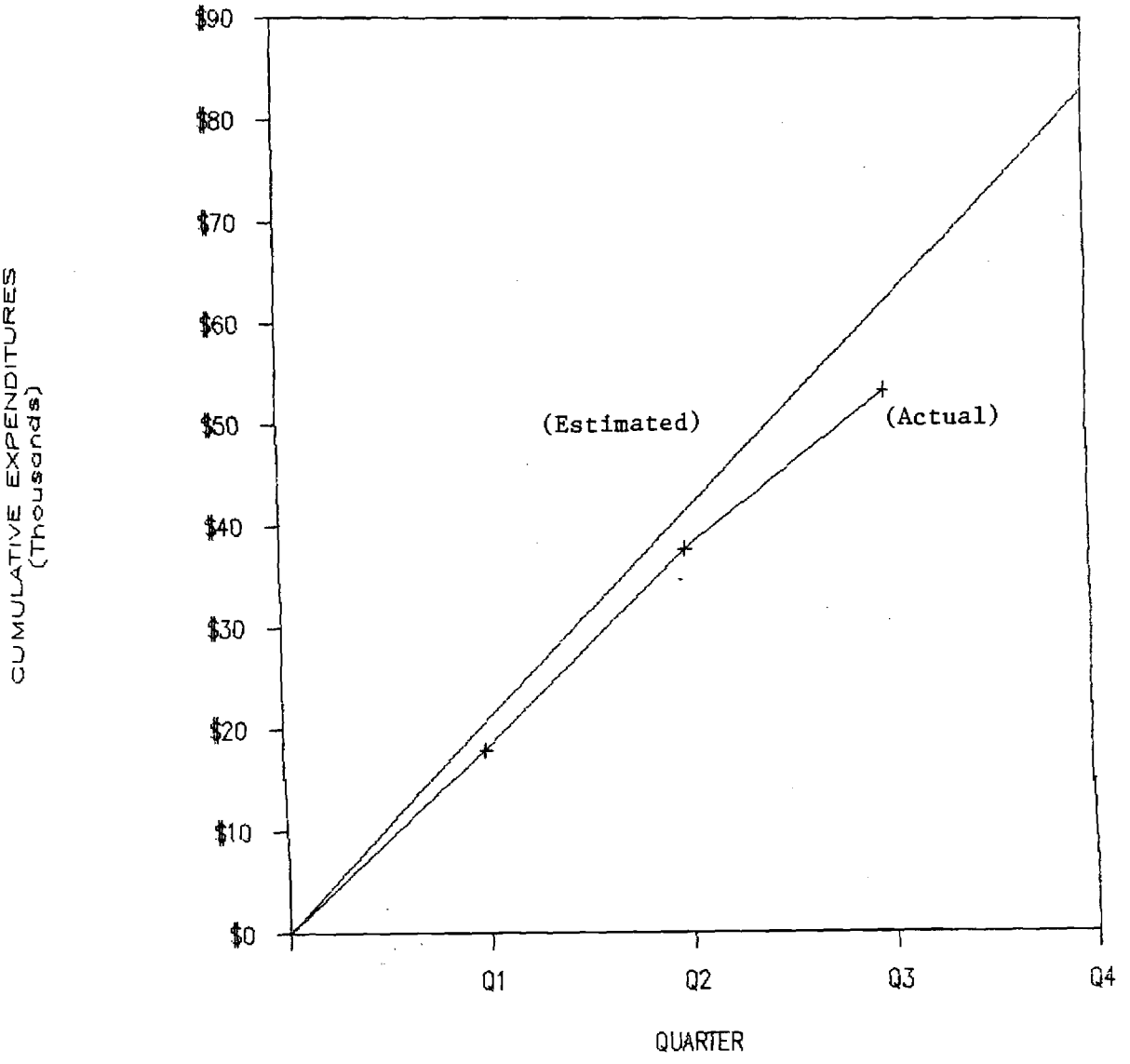
Date Prepared July 21, 1986
Contract No. MDA904-86-C-5002
Contractor Georgia Tech Research Corp.

Summary/Work Package Title
Fault Tolerant Distributed Computir
Report Month 4/1/86 - 6/30/86

FUNDS EXPENDITURE REPORT

Column A ORIGINAL PROPOSAL				Column B Latest Accepted Revised Proposal	Column C Reporting Quarter Expenditures	Column D Cumulative Expenditures to Date Total Man Hours	 Dollar Value	 Pct. Dollar Value	Column E Cost to Complete Estimate	Column F Latest Cost Estimate
1. Direct Labor										
Type	Number Of Hours	Hourly Rate	Dollar Total							
PI	525	$26.98	$14,164.50		$2,576.59	169.5	$4,573.11	32.3%	$9,591.39	$14,164.50
GRA	1950	$12.38	$24,142.85		$3,033.33	1518	$18,794.28	77.8%	$5,348.57	$24,142.85
Clerical	260	$8.33	$2,166.67		$425.00	224	$1,866.67	86.2%	$300.00	$2,166.67
Total Direct Labor			$40,474.02		$6,034.92		$25,234.06	62.3%	$15,239.96	$40,474.02
Burden @ 21.0% (Excluding GRA Labor)			$3,429.54		$630.33		$1,352.35	39.4%	$2,077.19	$3,429.54
Total Direct Labor and Burden			$43,903.56		$6,665.26		$26,586.41	60.6%	$17,317.15	$43,903.56
2. TRAVEL EXPENSE			$3,200.00		$2,452.70		$3,086.51	96.5%	$113.49	$3,200.00
3. GENERAL & ADMINISTRATIVE EXPENSE			$1,500.00		$0.00		$1,500.00	100.0%	$0.00	$1,500.00
4. COMPUTING CHARGES			$2,300.00		$575.00		$1,725.00	75.0%	$575.00	$2,300.00
TOTAL DIRECT COSTS			$50,903.56		$9,692.96		$32,897.92	64.6%	$18,005.64	$50,903.56
5. INDIRECT COSTS @ 63.5			$32,323.76		$6,155.03		$20,890.18	64.6%	$11,433.58	$32,323.76
TOTAL CONTRACT PRICE			$83,227.33							$83,227.33
TOTAL COMMITTMENTS AND EXPENDITURES					$15,847.98		$53,788.10	64.6%		

FUNDS EXPENDITURE GRAPH

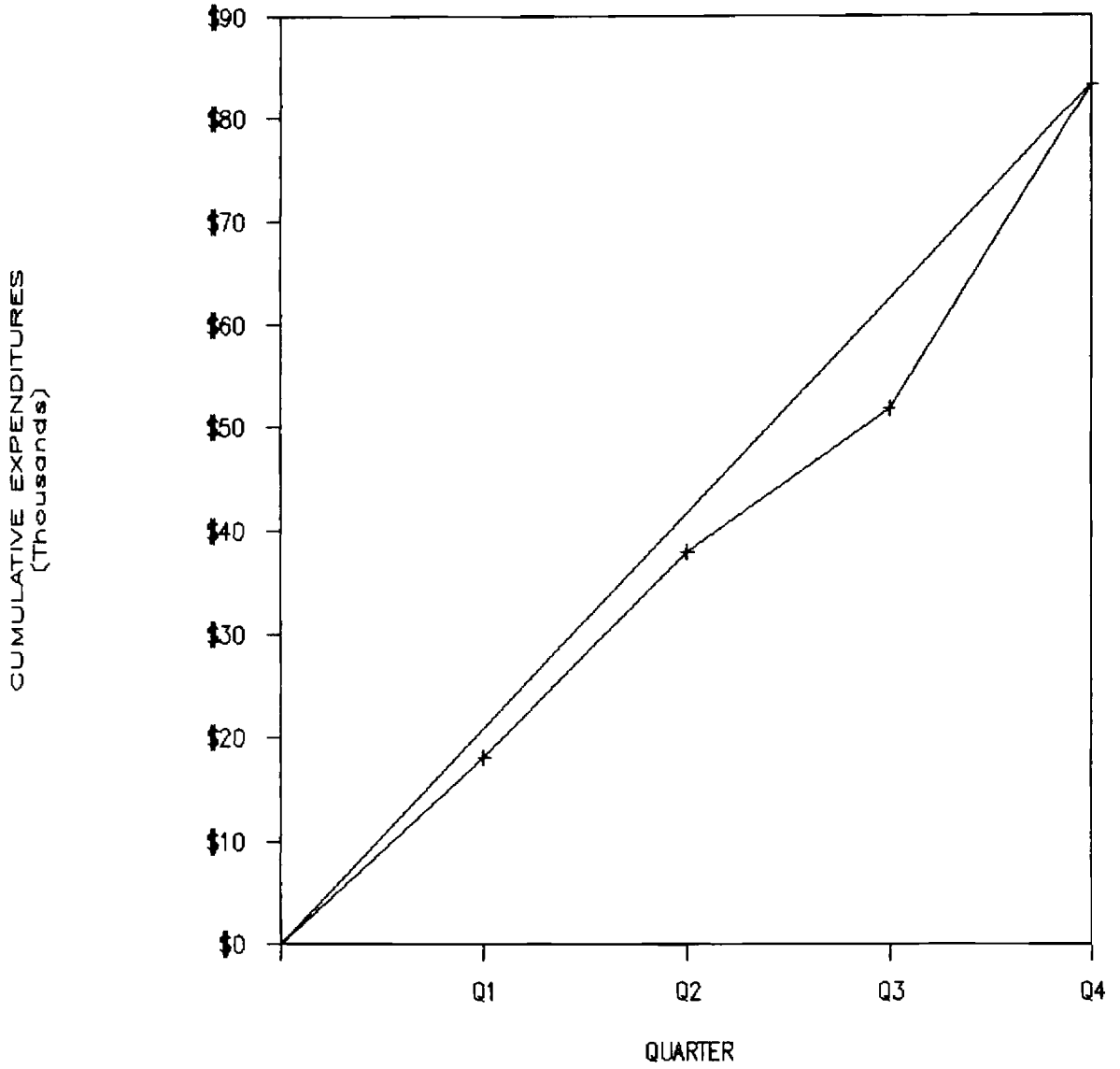


Data Prepared November 26, 1986
Contract No. MDA 904-86-C-5002
Contractor Georgia Tech Research Corp.

Summary/Work Package Title
Fault-Tolerant Distributed Computing
Report Month 7/1/-86 - 9/30/86

Column A			Column B	Column C	Column D		
ORIGINAL PROPOSAL			Latest Accepted Revised Proposal	Reporting Quarter Expenditures	Cumulative Expenditures to Date: Total Man Hours	Dollar Value	Pct. Dollar Value
1. Direct Labor							
Type	Number Of Hours	Hourly Rate	Dollar Total	Dollar Total			
PI	525	$26.98	$14,164.50	$14,164.50	$7,905.14	462.5	$12,478.25	88.1%
GRA	1950	$12.38	$24,142.85	$25,164.85	$8,431.43	2198	$27,213.33	108.1%
Clerical	270	$8.33	$2,247.65	$2,247.65	$408.33	268	$2,233.32	99.4%
Total Direct Labor			$40,555.00	$41,577.00	$16,744.90		$41,924.90	100.8%
Burden @ 21.0% (23.5% after 1 JUL 86) (Excluding GRA Labor)			$3,446.55	$3,446.55	$1,961.98		$3,305.58	95.9%
Total Direct Labor and Burden			$44,001.55	$45,023.55	$18,706.88		$45,230.48	100.5%
2. TRAVEL EXPENSE			$3,200.00	$2,178.00	$0.00		$2,130.43	97.8%
3. GENERAL & ADMINISTRATIVE EXPENSE			$1,500.00	$1,500.00	$503.80		$2,003.80	133.6%
4. COMPUTING CHARGES			$2,300.00	$2,300.00	$0.00		$1,583.80	68.9%
TOTAL DIRECT COSTS			$51,001.55	$51,001.55	$19,210.68		$50,948.51	99.9%
5. INDIRECT COSTS @ 63.5			$32,385.99	$32,385.99	$12,198.78		$32,352.31	99.9%
TOTAL CONTRACT PRICE			$83,387.54	$83,387.54				
TOTAL COMMITTMENTS AND EXPENDITURES					$31,409.46		$83,300.82	99.9%

FUNDS EXPENDITURE GRAPH



QUARTERLY PROGRESS REPORT
FAULT TOLERANT DISTRIBUTED COMPUTING
CONTRACT #MDA 904-86-C-5002
REPORTING PERIOD: 1 JAN 86 - 31 MAR 86

1. Project Status

During the past quarter, work has continued on each of the three project tasks. These efforts are closely related to other work in progress within the Clouds Project, our major research effort in the area of reliable distributed computing.

Under the Language Support for Robust Distributed Programs task, work continues in two major areas: the integration of the Aeolus compiler with the Clouds kernel services and the use of the Aeolus language system as a testbed for studying the problems of programming in action-object systems.

Under the Storage Management for an Action-Based Operating System task, the focus of our work has been on implementation, testing and integration with the virtual memory management mechanisms of the Clouds kernel.

Under the Operating System Support for Reliable Distributed Computing task, our efforts are directed toward specification and functional design of the operating system services which will be implemented on top of the object and action management mechanisms provided by the Clouds kernel. Our current focus is on a fault-tolerant job scheduler and an applications-level distributed database system.

The work on the tasks of this project is proceeding on schedule. Working in combination with other efforts in progress within the Clouds project, we are now in the process of debugging our initial prototype system.

2. Language Support for Robust Distributed Programs

As described in the last report, work in the systems programming language effort continues in two major areas: the design and implementation of the Aeolus language itself, as well as the use of the language as a testbed for the study of programming methodologies to achieve resilience and availability in action/object systems such as Clouds.

2.1 Language Design and Implementation

A new and substantially complete version of the reference manual for Aeolus[Wilk85] has been distributed locally for comment during the preceding quarter. This manual contains not only the description of the complete language (which, as we have mentioned in previous reports, has been considerably revised over the past year), but also provides definitions of the interfaces of Aeolus with its runtime libraries as well as with the object and action management features of the Clouds kernel. The most important portions of this manual have been summarized, along with commentary on the rationale underlying the major new features of the language, in a recent paper submitted to the IEEE Computer Society 1986 International Conference on Programming Languages.[Wilk86] (This paper is attached as Appendix A.)

The implementation of the Aeolus compiler has advanced considerably during the preceding quarter, taking into account the revisions mandated by the new version of the reference manual. The kernel routines for remote procedure call and primitive object management have recently been tested; runtime support for the interface with these kernel routines has been defined, and the testing of Aeolus code making use of the Clouds object management facilities awaits the implementation (now in progress) of a transport/linking mechanism to move compiled objects from our compiler development environment under Unix to machines running the Clouds kernel. As mentioned in our last report, use of the Clouds action management facilities from Aeolus code awaits the implementation of the action management portion of the kernel from Kenley's design[Kenl86] and pseudo-code implementation; this work is now in progress.

2.2 Programming Methodologies for Action/Object Systems

Our current work on the study of programming methodologies appropriate to distributed systems was described in our last report. This work involves the study of various methods of achieving resilient, available objects through the use of replication. Similar work[Birm85, Birm85a]

has recently been reported by researchers on the ISIS system at Cornell; however, that work (unlike ours) does not consider the problems introduced by network partitions, assuming rather that all failures are of the so-called *fail-stop* variety. In our work, we take into account the problems involved in reconciling the states of replicated objects which have run in independent partitions during a network failure. Thus, we may achieve higher availability in situations in which temporary violations to consistency are tolerable. Our work, as well as recent work[Dasg86] by other researchers in the Clouds project, has also suggested some of the functionality which will be required of the fault-tolerant job scheduler for the support of availability in Clouds. It is in the job scheduler that we envision most, if not all, of the knowledge about individual machines in the system will be concentrated, such as whether a certain machine is available or what the current loads are on the individual machines. Thus, the job scheduler is the natural portion of the system to support functionality such as the creation of distributed replicants of an object class, the selection of the most appropriate individual replicant from a class of such replicants to perform work requested of the class, or the support of *forward progress* (that is, moving work started on an object running on a system which subsequently failed to another system on which another replicant of the object exists). We anticipate that our work in the coming months will provide a firmer design for the interface needed with the job scheduler.

3. Storage Management for an Action-Based Operating System

The major components of the storage management system have been implemented and tested. The major effort this last quarter has been the completion of the partition and segment systems. Work has also continued on the device driver for the RA81 disk. We have demonstrated the support of page fault handling and object operation invocations[Spaf86] this quarter. More information on the features described in this report will be contained in a forthcoming paper.[Pitt86]

The primary work on the partition system was the implementation of the partition mounting routines. These routines examine the storage devices attached to the system in order to locate partitions. Then the routines make those partitions available to the system. The mounting of partitions includes the creation of the in-memory structures for the partitions and the addition of these structures to the system tables. Also, the mounting routines examine all segments which reside on the partition, if any, in order to initialize the allocation maps and to check the integrity of the segments. At this time, however, there is no support for the repair of the damaged segments; such segments would simply be flagged to the system. Data recovery is attempted during these checks for segments involved in action events. That is, the storage manager will try to complete commits that were in progress at the time of the crash and perform aborts on those segments on which actions were operating, but had not yet started to commit. The implementation of these routines were deferred until the routines that support action events were in a stable state.

The work in the segment system has been in two areas (with much overlap between them): virtual memory support and action event support. In addition, the routines for creating and removing segments were tested. The collection of the routines described below form the high level interface of the storage management system to the rest of the kernel.

Virtual memory support is provided in two set of operations. First there are the read and write segment block routines. These operations provide for the transfer of page-sized blocks of data between the physical memory on the VAXen and disk storage. These routines provide the page-in and page-out facilities for the virtual memory page-fault mechanism. Mappings between the virtual memory pages of a segment and the disk images of the segment are maintained by these routines. We have also placed some support for the action action management system in these routines, as the write routine will maintain several versions of recoverable pages on disk.

Support for providing mappings of segment data into virtual memory is found in the map window routines and the segment activate routines. A segment on disk is activated by creating an entry for it in the active segment table and initializing the entry with some basic data about the segment such as its size and type. After the segment is activated, portions of the segment can be mapped into virtual memory by creating the appropriate window descriptors. Each window descriptor describes the location, size, and characteristics of the segment portion (window) that is being mapped. The active segment table entry maintains the physical page mappings and disk mappings for the segment.

The segment system supports both segments which have permanent disk images (objects) and volatile segments (heap storage for object operation calls), which do not. Both types are handled by the same mechanisms, although volatile segments are never activated by the segment activate routine; they are created on the fly by the map window routine. Routines also exist for the removal of windows and the modification of windows.

Recovery of segment data is provided by two routines: a precommit routine and an end-of-action routine. The end-of-action routine is responsible for performing both commits and aborts depending upon the flag passed to the routine. The precommit is responsible for phase one support of the two phase commit protocol. This routine is called by the action management system[Kenl86] to examine recoverable segments touched by an action and to create shadow versions of the segments which were modified by the action. Precommit basically examines the page tables that map each segment to discover which pages are modified and then forms a minimal shadow on disk for that segment. The information used to create the shadow is stored in a descriptor in the active segment table entry for the segment. The end-of-action takes the information in the descriptor and uses the information to either commit the changes (the shadow becomes the new permanent version) or to abort the changes (the shadow is ignored and return to free storage). Both routines require special cases for objects which were created or deleted by the action. The code is present for the end-of-action routine, but the corresponding code must still be implemented in the precommit routine.

The storage currently supports a working device driver for RL02 removable pack device, which provides conventional i/o services plus support for flushing action requests to disk. Partition support is complete, providing partition creation and activation operations, i/o operations, directory lookup facilities, and a storage allocation mechanism. Currently, object and paging partitions are supported. Segment objects support is almost complete. Segments can be created and destroyed. Virtual memory mapping of segments is complete and integrated with mapping of objects. Work still needs to be done in aging segments from active status. Segment page fault handling on local segments is complete. Page fault handling for remote segments remains to be completed. Support for multiple virtual memory versions for action management is provided. The operations supporting the commit of action are as described above.

4. Operating System Support for Reliable Distributed Computing

The design of the Clouds kernel and the action management systems have been completed. The design work is progressing on some enhancements and application of the Clouds system.

The two notable design projects are the fault tolerance enhancer and the distributed database system. A probe based monitoring system is being developed that will tie into the reconfiguration system and also incorporate duplicated actions to allow the Clouds system to have forward progress in case of failures and allow intelligent, automatic system health maintenance.

4.1 Probes, Monitoring and Fault Tolerance

The basic fault tolerance mechanism supported by *Clouds* is the action paradigm implemented by the action management system. The action paradigm ensures consistency of the computing environment in the face of failures. It is a backward recovery scheme. A failed

action causes an implicit rollback, and the action may not be able to execute until the fault has been rectified. This degree of fault tolerance can be improved by the usage of better techniques that allow the action to continue using alternate paths of execution.

The key to improved fault tolerance lies in the implementation of a mechanism for the system to monitor itself. The monitoring can be at several levels, discussed later, but the basic components of the monitoring system are *probes*.

Probes in *Clouds* are a form of emergency status enquiries, that can be sent from a process to an object or to another process. When a probe is sent to an object, the probe causes the invocation of a *probe-procedure* defined by default in the object. The probe procedure returns to the caller a status report of the object. This includes the status of the synchronization mechanisms, the actions currently executing in the object and other relevant information.

The system monitoring subsystem consists of a process (daemon) that runs at each site (monitor). The monitor has a list of components that it needs to keep track of. The list is has a static part and a dynamic part. The static part contains capabilities to various critical system components (network drivers, disk drivers, schedulers, action management system and so on). The dynamic part consists of capabilities to user defined objects and actions that the user expressly records with the monitor, for tasks that require a high degree of fault tolerance.

The monitor at one site has a logical backup, that is a monitor at another site. The various monitors act as primaries for the site it runs on and doubles as a backup for a remote monitor. This allows the distributed system to detect site failures and network partitions.

The monitor periodically probes all the components in its list. The status of these components are stored in a fully replicated database. This database has the same structure and properties as the database used to locate *Clouds* objects, i.e. it is highly available, but may not be consistent at all sites, or may contain out of date data. The inconsistency of the database does not cause major disruptions in service. The data in the database are used by various system services and the reconfiguration system.

4.2 Object Based Distributed Database

A distributed database system, under design as of present is using the object paradigm built into the Clouds system, to build a sophisticated, flexible distributed database that supports consistency, availability, failure tolerance and replication.

One of the notable differences in structure between conventional database systems and a system supported by *Clouds* is the storage mechanism. Instead of files, we have a more powerful construct namely objects. In the following sections we describe how to implement a database system, using the object paradigm. Subsequently we discuss approaches to implement concurrency control and transaction commit for the database objects and transactions under the *Clouds* environment. We also provide insights into the effective management of the distributed database and how to provide support for data replication (the *Clouds* kernel does *not* support replication).

Virtually any kind of database system can be supported in the object based architecture. However to avoid getting into all the design approaches for various data modeling paradigms, we choose to discuss the most popular database model, the relational database model. The approaches for implementing other models would be different, but can be derived from the basic ideas in our design.

The basic building blocks in a relational database are relations and the relational operators that access the relations. At a slightly lower level are the access mechanisms used for fast access to individual or groups of tuples in the relational tables using key searching, indexing or hashing techniques.

The obvious way to implement an object-based relational database system is to use a relation per object scheme. An object holds all the data of the relation and contains the access

mechanisms to access the data. Thus the object defines operators that do key lookups, projections, tuple insertions, tuple deletions range queries and other such operations on the objects. A good feature of this approach is that the object can be encapsulated and be independent of any systemwide definition of structure or storage mechanisms. The internal structure of the object, that is the data organization (binary tree, B-tree, table unsorted), is not visible to the database system from outside and thus different relations can be organized in different ways and yet look functionally identical. The organization of each object could be tailored to the method that suites the data contained and the size of the object. The scheme of building a relational database by using relational objects, fragment handlers, access objects and multi-operator objects using *Clouds* has an interesting and important payoff. Distribution, concurrency control, recovery and transaction commit is automatic.

All the objects used by our system uses the *Clouds* default synchronization and recovery services. This implies, all accesses to data in any object uses the 2-phase locking protocol. The locking granularity is an object. For this reason we chose to use the fragmentation scheme. As described above, the handlers, shared objects and relational operators, do not update local permanent data and hence are never exclusively locked, permitting uninhibited concurrent access.

Clouds keeps track of each object touched by every transaction. The updates on these objects are made on shadow versions, and the permanent versions are not updated. When a transaction commits, all the object it touched are committed, that is the shadows are written to permanent storage and all locks are released. The commit uses a 2-phase commit strategy to ensure that site failures and network partitions do not give rise to inconsistent versions.

If a transaction aborts all updates are also cleaned up, by discarding the shadow versions. It is interesting to note that *Clouds* keeps only one shadow version for every object in use, that is, if several transactions are in the process of updating the same object, they would work on the same in-core version of the object. In this case if one transaction commits, it will cause the uncommitted updates of another transaction to be flushed to permanent storage. This scheme causes errors if the objects are recoverable but not synchronized. In our design, this situation cannot arise, as we use synchronized objects, and more than one transaction cannot update the same object concurrently.

The distributed database also support fragmentation of relations for efficient access as well as finer lock granularities. Replication is supported for higher availability. Fault tolerance is supported by the *Clouds* system and the monitoring scheme outlined above. The design is conceptually simple and yet quite general and powerful.

REFERENCES

[Birm85] Birman, K. P., and others. "An Overview of the ISIS Project." *DISTRIBUTED PROCESSING TECHNICAL COMMITTEE NEWSLETTER* (IEEE Computer Society) 7, no. 2 (October 1985). (Special issue on Reliable Distributed Systems.)

[Birm85a] Birman, K. P. "Replication and Fault-Tolerance in the ISIS System." *PROCEEDINGS OF THE TENTH ACM SYMPOSIUM ON OPERATING SYSTEM PRINCIPLES* (ACM SIGOPS), Orcas Island, Washington (December 1985).

[Dasg86] Dasgupta, P., and M. Morsi. "An Object-Based Distributed Database System Supported on the Clouds Operating System." TECHNICAL REPORT GIT-ICS-86/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[Kenl86] Kenley, G. G. "An Action Management System for a Distributed Operating System." M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01.)

[Pitt86] Pitts, D. V. "Storage Management for a Reliable Decentralized Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (In progress.)

[Spaf86] Spafford, E. H. "Kernel Structures for a Distributed Operating System." PHD DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (In progress.)

[Wilk85] Wilkes, C. T. "Preliminary Aeolus Reference Manual." TECHNICAL REPORT GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986.)

[Wilk86] Wilkes, C. T., and R. J. LeBlanc. "Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System." TECHNICAL REPORT GIT-ICS-86/12, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Submitted to the 1986 International Conference on Programming Languages.)

Rationale for the Design of *Aeolus*: A Systems Programming Language for an Action/Object System

Technical Report GIT-ICS-86/12

C. Thomas Wilkes
Richard J. LeBlanc

The Clouds Project

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280
(404) 894-3152

ABSTRACT

The goal of the *Clouds* project at Georgia Tech is the implementation of a fault-tolerant distributed operating system based on the notions of *objects*, *actions*, and *processes*, which will provide an environment for the construction of reliable applications. The *Aeolus* programming language developed from the need for an implementation language for those portions of the Clouds system above the kernel level. Aeolus has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the Clouds notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the Clouds system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as Clouds.

In this paper, the features provided by the language for the support of readability and maintainability in systems programming are described briefly, as is the rationale underlying their design. Considerably more detail is devoted to features provided for support of object and action programming. Finally, an example making use of advanced features for action programming is presented, and the current status of the language and its use in the Clouds project is described.

April 17, 1986

DRAFT—Submitted to the IEEE Computer Society 1986 International Conference on Computer Languages.

Rationale for the Design of *Aeolus*: A Systems Programming Language for an Action/Object System

Technical Report GIT-ICS-86/12

C. Thomas Wilkes
Richard J. LeBlanc

The Clouds Project

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280
(404) 894-3152

1. Introduction

The goal of the *Clouds* project at Georgia Tech[Allc82, Allc83, Allc83a] is the implementation of a fault-tolerant distributed operating system based on the notions of *objects*, *actions*, and *processes*, which will provide an environment for the construction of reliable applications. The *Aeolus*[1] programming language developed from the need for an implementation language for those portions of the Clouds system above the kernel level. Aeolus has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the Clouds notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the Clouds system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as Clouds.[LeBl85, Wilk86]

The intended users of Aeolus are systems programmers working on servers for the Clouds system. Clouds provides powerful features for the efficient support of resilient objects where the semantics of the object are taken into account; it is assumed that the intended users have the necessary skills to make use of these features. Thus, although support for the automatic recovery and synchronization features of Clouds is available, we have avoided providing very-high-level features for programming resilient objects in the language, with the intention of evolving designs for such features out of our experience with programming in Aeolus. These features will then be incorporated into an applications language for the Clouds system, which should allow programmers unskilled in fault-tolerant programming to write resilient objects.

Aeolus has its roots in a long line of structured programming languages, including Simula, Pascal, Modula-2, and Ada™. Thus, many of its features should be easy to understand for those familiar with one of these languages, and little space will be devoted here to discussion of such features; a description of the complete language is available in the Reference Manual.[Wilk85] Syntax and examples will be provided here only for those features of Aeolus which differ significantly from those of its predecessors, although the programming example discussed in section 6 should provide a feel for the flavor of the language.

---

1. Aeolus was the king of the winds in Greek mythology.

™ Ada is a registered trademark of the U.S. Government—Ada Joint Program Office.

An overview of the Clouds system from the point of view of Aeolus has been provided in a previous paper.[LeBl85] Briefly, the main structuring features of Aeolus (as of the Clouds system) are objects, actions, and processes. Clouds supports the *object* concept as a convenient structuring principle for facilitating recovery and synchronization. Besides supporting the Clouds object concept, Aeolus also allows the programmer to use the object features of the language for the specification of abstract data types, without necessarily invoking the object and action management features of the Clouds system. Thus, Aeolus objects provide a separate compilation facility as well as access to the object support of Clouds; the separation of object specifications into *definition* and *implementation* parts (much as are *modules* in Modula-2 or *packages* in Ada) provides a safe interface to separately-compiled objects, as well as facilitating the design of large systems consisting of many objects (possibly implemented by several people) or the use of predefined objects. Aeolus *pseudo-objects* provide a means of isolating system dependencies—such as input/output or low-level machine architecture—into object-like modules which provide operations facilitating machine-level programming.

The Clouds notion of *actions* corresponds roughly to the transaction concept of distributed database work, providing an "all-or-nothing" assurance of atomic execution. Support of actions in Aeolus is fairly low-level. Essentially, means are provided for specifying that an operation (procedure) of an object may be invoked as an action, or that an operation invocation is to be executed as a (toplevel or nested) action. Also, the status of action executions may be checked by means of calls to a Clouds action manager.

The *process* concept in Aeolus corresponds roughly to the *program* construct of Pascal or Modula-2. That is, a process ties together the constituent parts (objects) of a programmed system, and the invocation of a process provides activity in the Clouds system, since Clouds objects are passive.

In this paper, the features provided by the language for the support of readable, maintainable systems programs are described briefly, as is the rationale underlying their design. Considerably more space is devoted to the detailed description of features provided for the support of object and action programming. Finally, an example making use of advanced features for action programming is presented, and the current status of the language and its use in the Clouds project is described.

2. Support for Systems Programming

In this section, those features of Aeolus which are provided for the support of readable, maintainable systems programs, and which are not directly related to the support of action/object programming, are described briefly. These include structured types for access to low-level data representation, expression and statement constructs necessary for doing systems programming in a concurrent environment, and the typing mechanism.

2.1 Access to Low-Level Data Representation

Aeolus provides a wide range of traditional type classes. These include type names (the names of previously-declared types, including object types), and anonymous types (including enumerations, pointer types, structured types, and locks). Of interest here are those structured types providing access to the low-level representation of data, as is often required in systems programming, and those constructs providing support for synchronization in a concurrent environment.

Aeolus provides the traditional structured types, such as arrays, records, and sets. All types in Aeolus may be *parameterized* (see below); for example, the parameterized record type in Aeolus is similar in concept to the *discriminated record type* of Ada. The extension of parameterization to other types allows type constraints to be expressed in terms of parameters, and allows parameterized array types to be declared without the necessity of embedding them in record types. The parameterized array construct indirectly provides support for arrays with a flexible number of elements (so-called "dynamic" arrays); these may be simulated by using

pointers to parameterized arrays. Two important parameterized array types provided by the Aeolus implementation are *strings* and *bitstrings*. The string type allows convenient manipulation of character sequences. The bitstring, on the other hand, provides the most primitive structured abstraction of data, that of simply a sequence of *bits*:

```
type bitstring( length : unsigned ) is array [ unsigned[ 1..length ] ] of bit
```

The length constraint of the bitstring (in bits) is indicated by the value of the parameter "length." For example:

```
type nybble is bitstring( 4 )
```

Each "system" object[2] provides declarations of several useful bitstring types. These types are referred to collectively as the *storage classes*, since they define the units of storage supported by the hardware of most computer systems: types *bit*, *byte*, *word*, *longword*, and *quadword*, with lengths BITSIZE, BYTESIZE, WORDSIZE, LONGWORDSIZE, and QUADWORDSIZE, respectively.

Another important bitstring type, *address*, is also defined by the "system" object. The address type is defined as *bitstring(ADDRESSSIZE)*. The relationship between address types and pointer types is discussed below.

Several operations are provided for manipulation of bitstring data, including bitwise Boolean operations and shift operators. Access to individual bits of a bitstring is, of course, through array index operations. The provision of a true bitstring type addresses a lack experienced by users of both the Ada and C languages.[Evan84]

Another structured type providing access to low-level data representation is the *structure*, a special case of a record construct somewhat similar to the *packed record* construct of Pascal or the *packed* pragma as applied to the record construct (with representation specification) in Ada. The declaration of a structure type specifies the storage class which the structure will fit. A field in a structure typically represents a bitstring or scalar; the fields are packed together consecutively within an object of the specified storage class (without implicit padding), with the first field specified starting at the most significant bit position in the storage class. The compiler checks that the fields declared for the structure together fit into the specified storage class.

2.2 Constructs Supporting Synchronization

Features provided by Aeolus for the support of synchronization and mutual exclusion in a concurrent environment include the *lock* construct, the *region* construct, and the *shared* type attribute.

A *lock* type may be used to declare variables which in turn may be used to implement locking protocols on particular *values* in some domain.[3] A lock declaration includes the specification of a *compatibility list*, which defines, for a given *mode* of the lock, which other modes are compatible with that mode.[4] The presence of an identifier in a compatibility list serves as a declaration of that identifier as a mode of the lock type; the modes of a lock type

---

2. There exists a "system" pseudo-object for each computer system for which the Aeolus compiler is implemented. (At present, the "system" objects include *VAX_System* and *PC_System*, for the DEC VAX 11/780 and the Intel 8086-8088-80286 families of computers, respectively.) Each such object defines system-dependent constants, types, and operations required for systems programming. The appropriate object (determined by target code generation and controllable by compiler option) is imported implicitly by every compiland.

3. Note that a lock is obtained for a value of an object, and not on the object itself. Thus, for instance, a lock may be obtained on a file name even if that file does not yet exist. The lock structure is directly supported by the Clouds architecture.

4. A lock may be set with a specified mode only if other modes already set, if any, are compatible with that mode. Thus, a process adhering to a protocol using that lock may wish to block until the requested mode is available. Operations are provided by object *standard* for testing, setting, and releasing locks.

may together be thought of as an enumeration. An empty compatibility list indicates that the given mode is incompatible with all other modes.

The lock declaration may also specify the *domain* of values which may be locked. If the domain specification is omitted, a simple lock (i.e., one which does not lock over any particular domain) is assumed. For example:

```
type simple_lock is lock (   busy : []          )
type file_lock is lock (     read  : [read]     ,
                             write : []         ) domain is string( 20 )
```

The declaration of "simple_lock" above defines a lock type with a single mode "busy" which is incompatible with itself; thus, only one client may set a lock variable of type "simple_lock" at any one time. The declaration of "file_lock," on the other hand, defines a lock type over the domain of strings of length 20. Clients may set a lock variable of type "file_lock" on a given string with modes "read" or "write." The "read" mode is specified as being compatible with other settings of "read" mode; the "write" mode is incompatible with itself and with "read" mode. Thus, a client may set the lock with "read" mode on a given string even if several other clients have outstanding settings of the lock with "read" mode on that string; however, a client wishing to set the lock with "write" mode on a given string must wait for all outstanding settings of "read" mode on that string to be released.

All locks obtained during execution in the environment of an action (see section 4) are retained and propagated to the ancestor of that action upon committal unless explicitly released. Locks obtained under an action are released if the action aborts or successfully performs a toplevel commit. A lock is available to be granted under an action even if conflicting locks are held under one or more of the ancestors of that action, but not if conflicting locks are held under an action which is not an ancestor of that action.

The power of the Aeolus/Clouds lock construct in supporting user-defined synchronization lies in the specification of arbitrary locking modes, and arbitrary compatibilities between those modes, as well as the dissociation of locks from the locked data. To support mutual exclusion, Aeolus provides a *critical region* construct, access to which is controlled by association of the region with a designator for a *shared variable*. The shared variable is associated with a semaphore, yielding the familiar semantics of critical regions. In Aeolus, any type may have the attribute *shared*, which is inherited by any types of which the shared type is an element. In particular, Aeolus arrays may consist of shared elements; thus, the granularity of mutual exclusion may be tailored to achieve higher concurrency.

2.3 Type Compatibility and Conversion in Aeolus

The principal goal in the design of the Aeolus typing mechanism was the provision of strong typing where possible, but also the provision of escapes from strong typing where the special demands of systems programming required. Another concern which affected the design of these mechanisms was that programs be readable and maintainable by members of a fairly rapidly-changing research group. Thus, the desirability of brevity of notation was felt to be subordinate to that of rapid comprehension of code by readers (including the original authors of the code). Code must be entered only once; it must be read and understood possibly many times. Thus, we attempted to make the typing mechanism as simple as possible, simplifying the tasks both of the compiler and of the reader, who must otherwise remember numerous compatibility and implicit conversion rules, increasing the possibility of misunderstanding or confusion.

The operands of a binary operation in Aeolus are said to be compatible if they are of the same type, that is, if the types of the operand are *equivalent*. Type equivalence in Aeolus is by name.

As in Ada, a constraint which is associated with a scalar type (by way of a constraint specification in the type's declaration, or via a constraint declaration) is not considered part of that type, but rather is an attribute which is given to a member of that type.[5] Thus, a constrained member is compatible with a member which has the same type but a different (or no) constraint.

Types may be *parameterized*, that is, some of the attributes of a type may depend on the values of *formal type parameters*. (Object types may also be parameterized; see section 3.) In contrast to constraints, the parameters of a type as specified in the declaration of that type are, in general, considered to be a part of that type. The exception is when the specification of these parameters is delayed (via an empty actual type parameter option). (A member of a type with delayed parameterization is sometimes said to be associated with a *delayed constraint*.) A type with delayed parameterization is compatible with all possible parameterizations of that type. Types with delayed parameterization, when used as the types of formal procedure parameters, make possible generic operations on structured types such as arrays; and when used as pointer base types, allow the definition of pointers to arrays with flexible sizes.

The requirement for strict name equivalence of types is relaxed somewhat in the case of array slices, since slices, by their nature, cannot be associated with a named type. Rather, a slice is similar to a member of an unconstrained array type, any type parameter values of which are derived from the slice bounds, and which takes on as its base type the base type of the named type of the array from which the slice was derived. An array slice with *n* elements is compatible with any array or array slice with *n* elements and a compatible element type. Also, a slice of one element is compatible with any variable of a type compatible with the element type of the slice. Note that this implies the following correspondences:

```
bit    <=>   bitstringslice[1..1]   <=>   bitstring(1)
char   <=>    stringslice[1..1]     <=>   string(1)
```

Thus, bit is compatible with array [integer[1..1]] of bit; char is compatible with array [integer[1..1]] of char; and, in general, type *t* is compatible with an array of one element of element type *t*.

In the interest of keeping the implementation effort for Aeolus within reasonable bounds, it was decided not to provide facilities for the specification of overloading of user-defined operations in the current language. However, certain overloadings are available on predefined operators. In keeping with the goal of simplicity in the typing mechanism as stated above, the overloading of a binary operator is available only for operands which are compatible according to the definitions stated above. As seen from another point of view, this means that Aeolus does not perform implicit conversions. However, it is sometimes desirable to perform operations on operands of differing types. Thus, Aeolus provides the programmer with powerful means of explicit type conversion. Explicit type conversion functions are defined between members of closely related types within certain limitations.[Wilk85] In general, the name of a type may be used as the name of a conversion function; this type is the target type of the conversion.

Explicit conversions are allowed between types one of which is derived from the other (derived types), between different numeric types, between enumeration and numeric types, from an enumeration type to a string type, and between array types each of which meets conditions similar to those specified by Ada. Also, conversion is allowed (in both directions) between a type which is a bitstring type and any type with the same size (in bits) as the source

5. Constraints are used for range checking (if enabled) and for determining the sizes of structures, not for type checking.

type. In particular, conversions may be made between any array, record, or structure type and a bitstring type or array of bitstring type (e.g., array of byte or word) with the same size. Thus, access may be obtained to the bit representation of data in an explicit manner. Also, conversion is allowed from any pointer type to type *address*. In the other direction, conversion is allowed from type *address* to any pointer type; however, the result of such a conversion may be used only for dereferences, and may not be assigned to a pointer variable. Values may be assigned to address variables directly, by conversion from a pointer type, or via the *addr* operation, which yields the storage address of a static or dynamic data item; a value may be assigned to a pointer variable only by use of an allocator, or via assignment from another variable of the same pointer type. Thus, a safe (although restrictive) pointer mechanism is maintained separately from a permissive mechanism permitting address computations when necessary.

Finally, Aeolus provides a less restrictive (and less safe) means of type conversion in addition to the explicit (checked) conversion functions described above; the *retype* function is similar in spirit to the *unchecked_conversion* function of Ada. Of course, unchecked conversion may be used to convert addresses to any pointer type, thus violating the safety of the pointer mechanism. The intent of the *retype* function is to make such "end runs" around the typing mechanism obvious to the reader of the code, and when used with restraint and care it makes possible the sort of generic bookkeeping activity necessary in systems programming (e.g., memory allocation routines).

3. Support for Objects

The object construct provides support for *data abstraction* in Aeolus. A collection of related data items may be *encapsulated* within an object, which also may provide *operations* (procedures that operate) on the data. The only access to the data of an object is via these operations; thus, an object can strictly control manipulation of its encapsulated data, helping guarantee the invariants of the abstraction.

The object concept is supported at the lowest levels of the Clouds kernel; hence, we feel justified in using the term "object" to describe the data-abstraction facility of Aeolus, since an Aeolus/Clouds object has a real existence in the system. However, Aeolus does not provide a sophisticated inheritance mechanism such as that available in Smalltalk, nor does it provide for dynamic typing of objects. Aeolus provides a simple class mechanism in the *object type* described below; all *instances* of an object type inherit all operations (and other definitions) of that type. It was felt that, although an inheritance mechanism providing differential sharing of object operations would be useful, the support for such a mechanism should be left to higher-level portions of the system in order to keep the kernel as simple as possible; thus, inheritance will be among the features to be included in the language design to be evolved from our experience with Aeolus, as mentioned in the Introduction. Also, communication between objects in Aeolus is based on direct operation invocation rather than on message-passing, reflecting the fact that Clouds is not a message-passing system, but uses remote procedure call to support distributed computation. Hence, Aeolus is not strictly an *object-oriented language* in the sense of Rentsch.[Rent82] However, it provides access to an object concept supported throughout the Clouds system. The applications language to be based on our experience with Aeolus will likely come closer to the concept of "object-oriented language" in the strict sense.

An Aeolus object may have parameters indicating, for instance, sizes or element types of the abstraction implemented by the object; thus, an object implementing, for instance, a bounded stack abstraction may be parameterized by the element type and maximum number of elements of the stack. Then, various *instances* of the bounded stack object may be created (instantiated) with differing element types and sizes; the implementation of the object need not be concerned with details such as the element representation, and the programmer does not need to create new object types for each combination of element type and stack size. Support for such *generic objects* increases the level of abstraction available to the programmer, and makes possible the creation of libraries of reusable object types, in a spirit similar to that of the *generic package*

construct of Ada.

The object construct also provides a safe separate compilation mechanism. The separation of an object specification into a *definition part* and an *implementation part* allows checking across the interface to an object, as well as allowing the use of an object definition before the corresponding implementation part is finished (thus facilitating top-down design).

3.1 Object Definition Parts

The definition part of an object defines the interface of the object with other compilands. It specifies the attributes of the object itself as well as the constants, types, and operations which the object provides to other objects and to processes. (Note that variables may not be declared in object definition parts; it was felt that the sharing of variables between objects was not in accord with the principle of data encapsulation embodied by the object construct, which requires that all access to object data be through operations on the object.) The declaration of the object name in the header of the object definition defines a type, called an *object type*, with that name, which may be used in the declaration of variables to hold capabilities to *instances* of that object type (see below). An example of an object definition part is included in the Appendix.

Specification of the *autosynch* keyword in an object definition header causes code to be generated for automatic synchronization of object operation invocations based on programmer-supplied indications of operation effects (see below). This mechanism provides a simple read/write locking protocol; it may be used with any object class (see below).[6]

The *object class* is also specified in the object definition header. The object classes fall into two groups: the non-Clouds object classes (*pseudo* and *local*) do not use any of the Clouds facilities for action or object management, and are thus similar to *modules* in Modula-2 (for pseudo-objects) or to generic packages in Ada (for local objects), while the so-called Clouds object classes (*nonrecoverable, recoverable,* and *autorecoverable*) may make use of the object management facilities and (for recoverable and autorecoverable types) the action management facilities. Thus, the rationale behind the non-Clouds object classes is the same as that underlying the design of the corresponding features in Ada or Modula-2, that is, the provision of data-abstraction facilities usable "locally" (without resorting to operating system facilities). On the other hand, the Clouds object classes provide access to the support for data abstraction provided by the Clouds system when the expense of that support is warranted; the separate classes of Clouds object allow the programmer to specify the degree of support (and of incurred expense) required. The definitions of the object classes are as follows:

non-Clouds object classes:

pseudo (or pseudo-local) There may exist only one instance of a given pseudo-object type. This class of objects is used mainly for definition of system libraries, for interfacing with (separately-compiled) collections of procedures written in another programming language, for abstraction of machine and system dependencies, and as a basic separate-compilation mechanism.

local The standard class of non-Clouds object, which may have multiple instances. Management of local objects is provided by the Aeolus runtime system. Unlike Clouds objects, a local object may have no existence independent of the process or object which created it. Local objects simulate Clouds objects without incurring the expense of the use of the action and object management facilities.

---

6. For more information on the mechanisms supplied by the Clouds system to support synchronization and recovery, see Allchin's dissertation.[Allc83a]

Clouds object classes:

nonrecoverable The basic class of Clouds object. Objects of class *nonrecoverable* make use of the object management facilities, but may not contain features requiring action management, such as recoverable areas, permanent and per-action variables, or action event handlers (see section 4).

recoverable The "roll-your-own recovery" type of Clouds object, as opposed to the *autorecoverable* class of objects (described below), which provides completely automatic recovery. In some cases, the programmer may be able to use knowledge of the semantics of the object and its operations to program synchronization and recovery mechanisms more efficient than the automatic mechanisms supplied by the *autorecoverable* class of objects. Automatic recovery involves checkpointing of the entire object state; automatic synchronization is based on a simple read-write model of operation interactions on entire operations. As will be discussed in section 4, Aeolus provides facilities that allow the programmer to specify which parts of the object state are to be checkpointed (recoverable areas), to access information about the states of actions and to change these states (via operations on the action manager), and to control the recovery process by specification of what is to be done during action events (action event handlers); also, the programmer may specify finer-grained locking mechanisms for greater control of synchronization (via the *lock* type; see section 2). Only *recoverable* objects may contain recoverable area specifications and action event handler specifications.

autorecoverable As mentioned above, *autorecoverable* objects provide completely automatic recovery. The entire object state (the global variables of the object) is recoverable, and the default event handlers are used.

Operations on objects of class *recoverable* or *autorecoverable* may be executed only within an *action environment* (see section 4). An instance of an object (other than of class *pseudo*) is created by use of an *allocator*, a construct also used for allocation of pointer values (and similar to that used for allocating "access objects" [pointers] in Ada). This underscores the similarity in treatment between object capabilities and pointers, in particular, the processes of creation, initialization, and deletion (disposal), as well as their use as elements in lists and other structures and as parameters to objects and object operations. The values of any object parameters must be specified by using a constructor in the allocator. The allocator yields a capability to the newly-created object instance, which may be assigned to a variable of that object type. The variable may thereafter be used to qualify operation invocations on that object instance. The *init* object event handler (see below) for the object is executed during the instantiation process, as are any variable initializations required by the object.

The definition part also performs any necessary imports of other object definitions before the declarations of the object definition are given. These are called its *visible declarations* since the declarations are available to any object which imports the object definition. As stated above, the visible declarations of an object may include specifications of constants, types, or operations, but not of variables. Finally, specifications of the object's operations are provided. An operation specification may optionally be given one of the attributes *examines* or *modifies*, which indicate that the operation reads from or writes to the object's state, respectively. This information is used by the compiler to generate automatic read or write locking for each operation if the *autosynch* attribute is specified for the object. If no operation effect is specified, the compiler assumes that the operation neither reads nor modifies the object state, and thus no automatic locking would be done for that operation. The *autosynch* feature thus provides automatic synchronization according to a simple multiple readers/single writer protocol. An object operation (or other procedure) meeting certain conditions[Wilk85] may also be given the *inline* attribute, indicating that inline code expansion of the operation is desired; thus, the use of

operations to access portions of encapsulated data is made more efficient. We have found in our experience that when complicated data structures are encapsulated (such as compiler type attribute records), the number of operations which exist merely to provide controlled access to small portions of the hidden data structure becomes very large; the efficient support of such operations thus becomes important to encourage the use of encapsulation.

3.2 Object Implementation Parts

The implementation part of an object provides the actual code for the operations of the object, as well as the definitions of any private constants, types, variables, or procedures needed by the object. (An example of an object implementation part is provided in the Appendix.) These definitions are, of course, hidden from other compilands; only those definitions specified in the object definition part are available outside the implementation part. This supports the hiding of encapsulated data required by the data-abstraction feature, and is similar in concept to Ada package and Modula-2 module implementations.

The definition part of the object being implemented is implicitly imported by the implementation part; also, any objects imported by the definition part are available in the implementation part. The implementation part may import other objects as well via its own import clauses. All constants, type definitions, and operations declared in the objects made available by any of these methods are visible in the implementation part; also, the names of these imported object types may be used as the types of variables declared in the implementation part. Such variables must be initialized by use of an allocator.

Variables declared in the outer level of the block of the object implementation part are global to the object, and are static ("own") variables; that is, the values of such variables survive between calls to the object's operations. The global variables of an object are called collectively the object's *state*. In an object of class *recoverable,* part of the object state may be specified to be in a *recoverable area;* also, the programmer may specify an *action events part* and/or a *per-action variables part*. Recoverable areas, action events, and per-action variables are described in section 4.

In order to allow the object to participate in its own creation and deletion, an object implementation part contains specifications of handlers for the so-called *object events*. The object events include the *init* or object initialization event, the handler for which is executed whenever an instance of the object is created by use of an allocator; the *reinit* or object reinitialization event, the handler for which is executed—if the object has registered its desire for reinitialization with the action manager—when the system is reinitialized after a crash or network partition; and the *delete* or object deletion event, the handler for which is executed when the object instance is destroyed. No default handler for the *init* object event is assumed; if no action is desired for the *init* event, the programmer must supply a NULL statement as the handler body. The intent is to help prevent the accidental omission of object state initialization by the programmer. If no handler for the *reinit* object event is specified, the handler is by default the same as that specified for the *init* event. If no handler for the *delete* object event is specified, it is assumed to be NULL.

3.3 Object Operation Invocations

An invocation of an object operation looks much like a procedure invocation, except that, outside the implementation part of the object itself, an operation name must be qualified by the name of a variable representing an instance of that object type (or, for pseudo-objects, by the name of the object type itself). Thus, for an instance of a bounded-stack type, we may have

```
stack_instance@push( elem )
```

When an object invokes one of its own operations, however, the usual procedure call syntax is used.

Invocations of pseudo-object and local object operations have semantics essentially similar to those of calls to procedures local to a compiland. The situation is different for operations

declared in objects which use the Clouds object-management facilities (i.e., the so-called "Clouds objects").[7] Invocations of operations on Clouds objects are handled by the compiler through operations on the Clouds object manager on the machine on which the invoking code is running. The Clouds object on which the operation is being invoked need not be located on the same machine as the invoking code; the object manager then makes a *remote procedure call* (RPC) to the object manager on the machine on which the called object resides. The location—local or remote—of the object being operated upon, however, need not concern the programmer, as the RPC process is transparent above the object-management level.

4. Support for Actions

The action concept provides an abstraction of the idea of work in the Clouds system; an action represents a unit of work. Actions provide *failure atomicity*, that is, they display "all-or-nothing" behavior: an action either runs to completion and *commits* its results, or, if some failure prevents completion, it *aborts* and its effects are cancelled as if the action had never executed. The rationale for the action concept and the mechanisms supporting it in the Clouds system are described in Allchin's dissertation;[Allc83a] the design for the implementation of these mechanisms is described in Kenley's thesis.[Kenl86]

Support for actions in the Aeolus language is relatively low-level. The methodology of programming with actions is not at present well-understood compared with experience in programming with objects; thus, rather than providing high-level syntactical abstractions such as those available for object programming, Aeolus allows access to the full power and detail of the Clouds system facilities for action management. The major syntactic support provided by Aeolus for action programming is in the programming of *action events*, *recoverable areas*, *permanent* and *per-action variables*, and *action invocations*.

4.1 Action Events

At several points during the execution of an action, the action interacts with the *action manager* of the Clouds system to manage the states of objects touched by that action, including writing those states to *permanent* (stable or safe) storage, and recovering previous permanent states upon failure of an action. Thus, failure atomicity may be provided by the action management system. The *action events* include:

event name	*purpose*
BOA	beginning of action
toplevel_precommit	prepare for commit for a toplevel action
nested_precommit	prepare for commit for a nested action
commit	normal end of action (EOA)
abort	abnormal end of action

The interactions with the Clouds action manager necessary when such events take place are done by default procedures supplied by the Aeolus compiler and runtime system; these procedures are called *action event handlers*. When an action event occurs for a particular action, the action manager(s) involved invoke the event handlers for each object touched by that action.

As was described in section 3, by use of the *autorecoverable* class of object, the programmer may take advantage of the recovery facilities of the Clouds system by having the compiler generate the necessary code automatically. This automatic recovery mechanism requires

7. This is because the code for pseudo-objects and for local objects is actually linked into the code of the compiland using these objects, whereas the code for Clouds objects is physically separate from the code of the invoking compiland. This code is paged in on demand by the object manager; see Allchin's dissertation.[Allc83a]

checkpoints of the entire state of the object, and uses the default action event handlers. However, it is sometimes possible for the programmer to improve the performance of object recovery by providing one or more object-specific event handlers which make use of the programmer's knowledge of the object's semantics; these programmer-supplied event handlers then replace the respective default event handlers for that object. Thus, if object class keyword *recoverable* is specified in the definition header of the object being implemented, the programmer may give an optional *action event part* in the object's implementation part. Following the keywords *action events*, the programmer lists the name of each action event handler provided by the object implementation as well as the name of the action event whose default handler the specified handler is to override. Thus, for example, the specification (say, in an object implementing a bounded-stack abstraction):

```
action events
  stack_BOA overrides BOA,
  stack_nested_precommit overrides nested_precommit
```

indicates that the default handlers for the *BOA* and *nested_precommit* action events are to be replaced by the procedures named "stack_BOA" and "stack_nested_precommit," respectively, for the bounded-stack object type only.

4.2 Recoverable Areas

As mentioned in section 3, if an object being implemented is of class *recoverable*, then some of its variables may be declared in a *recoverable area*. The state of a recoverable area which has been touched by an action is maintained on a *version stack* by a Clouds action manager, and is saved to permanent storage upon commit of the action which touched it. If an action which touched a recoverable area is aborted, the version of that area which existed before the action touched it is restored.[8] Thus, the use of recoverable areas allows the programmer to provide finer granularity in the specification of that part of the object state which must be checkpointed, since the use of automatic recovery on object (the *autorecoverable* object class) performs checkpoints on the entire state of the object.

The interaction with the action manager necessary to manage the states of recoverable areas is implemented by the action event handlers as described above. Again, the default event handlers may be overridden by programmer-supplied event handlers for the entire object to achieve better performance.

4.3 Permanent and Per-Action Variables

It may sometimes be desirable to make large data structures resilient. In such cases, the recoverable area mechanism may be inefficient, since it requires the creation of a new version of the entire recoverable area for each action which modifies the area. Often in such cases the programmer make take advantage of knowledge of the semantics of the data structure to efficiently program the recovery of the data structure. The Aeolus language provides two constructs which aid in the custom programming of data recovery, the so-called *permanent* and *per-action variables*, constructs proposed by McKendry.[McKe85]

Any type may be given the attribute *permanent*. This attribute indicates that members of that type are to be allocated on the *permanent heap*, a dynamic storage area in the object storage of each object instance. This area receives special treatment by the Clouds storage manager; in particular, it is *shadow paged* during the *toplevel precommit* action event.[9] Any type which has

8. For more information on the semantics of recoverable areas and the mechanisms to support them, see Allchin's dissertation.[Allc83a]

9. More information on the management of permanent heap storage is available in several papers on the Clouds system.[Pitt85, Keni86, Wilk86]

as its base or element type a type with the attribute *permanent* inherits that attribute. Permanent variables may be assigned values only within a *toplevel precommit* event handler.

Aeolus also provides the per-action variable construct. An object implementation part of class *recoverable* may declare a single per-action variable section. A per-action specification resembles a recoverable area specification, and the semantics is also similar, in that each action which touches an object with per-action variables gets its own version of the variables; however, the programmer may access the per-action variables not only of the current action, but also of the parent of the current action. Also, per-action variables are allocated in *volatile storage,* that is, in storage the contents of which may be lost upon node failure. The variables in a per-action specification are accessed as if they were fields in a record described by the specification; two entities of this "record type" are implicitly declared: Self and Parent, which refer respectively to the per-action variables of the current action and its immediate ancestor.

Permanent and per-action variables may be used together to simulate the effect of recoverable areas at a much lower cost in space per action. In general, the per-action variables are used to propagate changes to the resilient data structure up the action tree; these changes are then applied during the *toplevel precommit* action event to the actual data structure in permanent storage. The use of permanent and per-action variables is shown more fully in the programming example shown in the Appendix (and described in section 6).

4.4 Action Invocations

The right-hand side of an assignment statement may take the form of an *action invocation.* Here, the right-hand side (which consists of an operation invocation which, if the operation is value-returning, is embedded in another assignment statement) is invoked as an action; the *action ID* of this action is assigned to the variable designated by the left-hand side of the action invocation. Thus, for example, if the bounded-stack object mentioned in section 3 were defined as a recoverable object, we might invoke one of its operations as an action:

```
aID := action( stack_instance@push( elem ) )
```

The action ID may be used as a parameter in operations on the action manager which provide information about the status of the action, cause a process to wait on the completion of an action, or explicitly cause an action to commit or abort.[10] The programmer may specify that an action be created as a "top-level" action, that is, as an action with no ancestors; a top-level action cannot be affected by an abort of any other action. Otherwise, the action is created as a "nested" action, that is, as a child (in the so-called *action tree*) of the action which created it; as described below, a nested action may be affected by an abort of one of its ancestors. Optionally, a *timeout value* may be specified; if the action has not committed by the expiration of this timeout, the action will be aborted. If no timeout value is specified, a system-defined default value is used.

In Clouds, we distinguish between operations *invoked as an action* and operations *executing in an action environment.*[11] We say that an operation is executing in an action environment if that operation is invoked as a toplevel or nested action, or if the invoker of the operation is executing in an action environment. Thus, it is a sufficient, but not a necessary, condition that an operation be invoked as an action to be running in an action environment. Only an operation or internal procedure of a *recoverable* or *autorecoverable* object may be invoked as an action; however, all operation invocations on such objects *must* be executing in an action environment. Thus, operations of a non-Clouds object or of a *nonrecoverable* Clouds object may execute in an action environment, but may not be invoked as an action. All *recoverable* or *autorecoverable*

10. The interface to the Clouds action manager is described in the Reference Manual.[Wilk85]

11. Some transaction systems require the creation of one or more nested actions to encapsulate every remote operation invocation. In Clouds, such encapsulation is not required, but is available at the programmer's option.

Clouds objects operations of which are invoked within the environment of an action are said to be *touched* by that action.

The semantics of an action invocation is as follows: the action manager operation *CreateAction* is invoked with the name of the operation to be performed as well as the list of arguments to be passed to that operation.[12] The action manager then invokes the BOA event handler on the object to which the operation belongs. Next, the action manager creates and dispatches a process in which the operation code runs. When an object is first touched by the action, the BOA handler of that object is invoked. An attempt by an operation invoked as an action to return to its caller is considered an implicit attempt to commit the action, and will cause control to transfer to the *Commit* operation of the action manager, which terminates the process and invokes the precommit event handler of each object touched by the action. (An explicit invocation of the *Commit* operation has the same effect.) If precommit of the object is successful, the action manager then invokes the commit event handler of each touched object. If the action (or one of its ancestors) invokes the *Abort* operation of the action manager, the action manager terminates the process corresponding to the action and invokes the abort event handler of each object touched by that action.

It may sometimes occur that an object operation may be called either as an action invocation, or as an ordinary object operation invocation (even in an action environment). In the case that an operation is invoked normally (that is, not invoked as an action), an invocation of the action manager operation *Commit* by the operation will cause the action manager to merely return control to the point of invocation of the original operation; thus, in this case the *Commit* call is effectively a normal procedure return. On the other hand, an invocation of the *Abort* operation by an operation invoked normally will cause the parent action of the invoker of the original operation to abort. Aeolus does not provide an explicit exception-handling mechanism. This function is subsumed, for operations executing within an action environment, by the action event handling mechanism. However, in the case of operations not invoked as actions, a call to the *Abort* action manager operation—as described above—provides a mechanism similar to an exception-handling mechanism with a single exceptional condition (say, *"error"*).

5. Support for Processes

The final structuring feature of the Aeolus language provides an abstraction of the *process* concept of the Clouds system. (The process is analogous to the *program* construct of Pascal or Modula-2.) The invocation of a process provides activity in the Clouds system; processes may be considered the "glue" which binds object operations, and possibly actions, to do useful work.

A process is introduced by a header which gives the name of the process, as well as clauses detailing any imports of object definitions necessary. Following any import clauses, the body (process block) of the process is specified; the statement part of this block is the entry point when the process is activated, and execution begins there after any necessary variable initializations of the process block have been performed.

6. A Programming Example

In this section, we discuss an example of systems programming using the constructs which Aeolus provides for access to the powerful features of the Clouds system for action and object programming. The text of the example object discussed here is provided in the Appendix.

Since the use of a *recoverable* data structure requires the creation of a complete copy of the data structure on the version stack for each action which modifies the data structure, the

12. The exact details of the manner in which this information is provided depends on whether the operation is a local procedure or a publicly-visible operation of the object to which it belongs.

implementation of a replicated object can become inefficient as the size of the data structure increases. Fortunately, we can use semantic knowledge about the object to simulate the effect of recoverable variables at a fraction of their cost. The technique which we use was proposed by McKendry.[McKe85] Essentially, rather than require that the system allocate a new version of a complete data structure for each new action, we make use of per-action variables (described in section 4) to maintain "change lists" for each action. These may be viewed as "intention lists" for operations such as insertion or deletion in a data structure. Since each action can access both its own recoverable variables and those of its parent, we may arrange to propagate the change lists of an action to its parent, usually by coding an event handler for the *nested_precommit* action event. (We may also wish to arrange to clean up after ourselves in an *abort* action event handler.) The actual modification of the data structure itself is delayed until the *toplevel_precommit* action event. In a handler for this event, we must arrange to perform the changes maintained in the change lists (say, insertions and deletions) on the actual data structure; the actual data structure is maintained in permanent storage. Note that the Aeolus/Clouds system enforces the restriction that data in permanent storage may only be modified at toplevel precommit; then, the Clouds storage management system[Pitt85] assures the stability of permanent storage and the atomicity of changes to it.

The use of this technique often requires the use of linked lists or similar data structures allocated in a heap in the permanent area of per-object storage. This *permanent heap* requires special run-time support for its management, which must maintain the heap's consistency across failures.

In the example shown in the Appendix, we show a proposed design for the permanent heap manager. To maintain the consistency of the heap, this PERMHEAP object uses the same techniques which the permanent heap mechanism is designed to support, i.e., per-action variables and associated action-event handlers. In the implementation shown, the "free list" (i.e., the list of available blocks of permanent storage) is itself kept in permanent storage to ensure the resilience of the permanent heap structure. (Thus, the PERMHEAP object must actually be bootstrapped from a non-resilient version.) For the purposes of this example, we have written PERMHEAP as a recoverable object. In practice, the permanent heap support would be part of the runtime support code compiled into a recoverable object, rather than a separate object.

The PERMHEAP object maintains lists of those blocks of the heap allocated and freed by each action, in per-action variables. Also, each action which allocates a block of storage obtains a lock on the value of the address of that block. (Blocks of storage are uniquely identified by their starting addresses.) The presence of a lock on a block of storage indicates that it has been allocated by some action which has not yet committed; since changes to the actual "free list" are not made until toplevel precommit of the action allocating storage, this lock is necessary to inform other actions that the block is probably unavailable. A call to the ALLOCATE operation of PERMHEAP will return the address of a block of memory in the permanent heap area of the object; the address of the block is also added to the ALLOCATED per-action list and is locked. If a block of memory was allocated by the action which is trying to free it, a call to PERMHEAP's FREE operation will remove that block from the ALLOCATED list and release the lock on its address, effectively causing the block to never have been allocated. If, on the other hand, the block was not allocated by the invoking action, the address of the block to be disposed is merely added to the FREED per-action list; actual disposal is performed at toplevel precommit.

No special processing is required if an action allocating or freeing storage aborts, since its locks are released and no alteration to the permanent "free list" has taken place. Thus, effectively no allocations or frees have taken place. (Note that the contents of permanent storage blocks on the "free list" are considered dispensible; nevertheless, these contents may be modified only at toplevel precommit.) When a nested action enters its commit phase, its ALLOCATED and FREED per-action lists are propagated to its parent. Memory blocks on the permanent heap allocated by an action are actually removed from the "free list" when the

action's toplevel ancestor (to which the nested action's per-action lists have been propagated) enters its precommit phase; also, blocks freed by the action are added to the "free list" at this time.

In Clouds, locks (as well as all portions of the object state which are not specified to be *permanent* or *recoverable*, including per-action variables) are maintained in volatile storage. Thus, information concerning uncommitted allocations and frees of an object using permanent heap support may be lost due to node failure. However, this will not cause a problem, since uncommitted actions running at a node at the time of its failure will be aborted. Recall that locks belonging to an action are released if that action is aborted; thus, the effect is the same as if the actions had been aborted in a non-failure case, i.e., as if the allocations and frees had never been performed.

Note that this implementation of the PERMHEAP object does not provide strict serializability. To see this, consider some action, *A*, which exhausts (or nearly exhausts) the permanent heap, causing other actions *B* and *C* trying to allocate permanent memory to fail. Action *A* may well be aborted itself. Actions *B* and *C* which failed because of *A* might not have failed had they been executed serially. However, such breaches of strict serializability do not affect the consistency of the permanent heap mechanism, and thus are of little concern in this context.

7. Status of the Aeolus Implementation

In the Clouds systems programming language effort, work is currently continuing in two major areas as of winter 1986: the implementation of the Aeolus compiler as well as its integration with the Clouds kernel services, and the use of the Aeolus language system as a testbed for studying the problems of programming in action-object systems.

Work on the compiler is in progress on one of the DEC VAX 11/750s of the Clouds system, under the BRL version of Berkeley UNIX™ 4.2. The Amsterdam Compiler Kit[Tane83] is being used for code generation for both the VAXen and the IBM PC AT™ workstations of the Clouds project. The basic portion of the compiler implementation has been finished, including support for non-Clouds objects. Current work on the Aeolus implementation is being concentrated on those areas of functionality needed for interfacing with the kernel to provide support for object and action management. We anticipate that the functionality required for a prototype implementation in Aeolus of the recently-completed action management design[Kenl86] will be available in summer 1986. (The interfaces to action management are described in the Reference Manual[Wilk85] for Aeolus.) The actual implementation of action management is being done in C, as it will be merged with the kernel code (which is also in C). Concurrently, work is progressing on the development of structured editing tools for Aeolus using the GANDALF structured-editor generator system.[Notk85]

Our plans to use the Aeolus/Clouds system as a testbed for studying programming methodologies in action-object systems have been described in a previous paper.[LeBl85] As one of these ongoing studies, we are working towards the development of a distributed object filing system for Clouds; alternate implementations of the file system will compare the efficiency of different schemes for achieving consistency and availability. Of special interest are the trade-offs available among different schemes between consistency and availability, particularly when semantic knowledge of an object may be brought to bear. This research will be described in a forthcoming paper.[Wilk86]

™ UNIX is a registered trademark of AT&T. PC AT is a registered trademark of IBM.

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, Atlanta, GA, 1982.

[Allc83] Allchin, J. E., and M. S. McKendry. "Synchronization and Recovery of Actions." *PROCEEDINGS OF THE SECOND SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING* (ACM SIGACT/SIGOPS), Montreal (August 1983).

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

[Evan84] Evans, A. Jr. "A Comparison of Programming Languages: Ada, Pascal, C." In *Comparing & Assessing Programming Languages*, ed. A. Feuer and N. Gehani, 66-94. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[Kenl86] Kenley, G. G. "An Action Management System for a Distributed Operating System." M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01.)

[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 (July 1985). (Also released, in expanded form, as technical report GIT-ICS-85/03.)

[McKe85] McKendry, M. S. "Ordering Actions for Visibility." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **11**, no. 6 (June 1985). (Also released as technical report GIT-ICS-84/05.)

[Notk85] Notkin, D. "The GANDALF Project." *THE JOURNAL OF SYSTEMS AND SOFTWARE* **5**, no. 2 (May 1985).

[Pitt85] Pitts, D. V., and E. H. Spafford. "Notes on a Storage Manager for the Clouds Kernel." TECHNICAL REPORT GIT-ICS-85/02, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[Rent82] Rentsch, T. "Object Oriented Programming." *SIGPLAN NOTICES* (ACM) **17**, no. 9 (September 1982): 51-57.

[Tane83] Tanenbaum, A. S., H. van Staveren, E. G. Keizer, and J. W. Stevenson. "A Practical Tool Kit for Making Portable Compilers." *COMMUNICATIONS OF THE ACM* **26**, no. 9 (September 1983).

[Wilk85] Wilkes, C. T. "Preliminary Aeolus Reference Manual." TECHNICAL REPORT GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986.)

[Wilk86] Wilkes, C. T. "Programming Methodologies for Resilience and Availability." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (In progress.)

Appendix

The following example is discussed in section 6 of this paper. Reserved words of Aeolus are indicated by boldface.

```
definition of recoverable object permheap is
  ! Gives the publically-visible definitions provided by the PERMHEAP object.

operations

  procedure allocate ( size : unsigned ) returns address modifies
    ! Return a pointer to a block of memory of the given "size" (in words) in
    ! permanent memory.

  procedure free ( block : address ) modifies
    ! Dispose the block of memory indicated by "block".

end definition. ! permheap !


implementation of ! recoverable ! object permheap is
  ! Support for the permanent heap, using per-action variables for recovery management.

import list
  ! The definition part of the LIST object is shown here for clarity.
  !
  ! definition of local object list ( elem_type : type ) is
  !   -- This object implements a linked list abstraction. The object is parameterized
  !   -- by the element type of the list; if the element type is specified to be permanent
  !   -- by a (recoverable) importing object, then the linked list itself will be allocated
  !   -- in permanent storage (only recoverable objects may declare permanent variables).
  !   -- The list is initially empty. Mutual exclusion is provided on MODIFY operations.
  !
  ! type compatible_list is list( elem_type )
  !
  ! operations
  !   procedure add ( elem : elem_type ) modifies
  !     -- Adds ELEM to the list.
  !   procedure append ( l : compatible_list ) modifies
  !     -- Appends all elements in list L to this list.
  !   procedure remove ( elem : elem_type ) modifies
  !     -- If ELEM is on the list, removes it.
  !   procedure find ( elem : elem_type ) returns boolean examines
  !     -- If ELEM is on the list, returns TRUE, otherwise FALSE.
  !   procedure nth ( n : unsigned,  notthere : out boolean )
  !                 returns elem_type modifies
  !     -- If the Nth element exists, returns it and sets NOTTHERE to FALSE,
  !     -- otherwise sets NOTTHERE to TRUE.
  ! end definition.

! The local declarations of the PERMHEAP object.
!
! Here, we give the names of alternate handlers for some of the action events.
! Note that no alternate handler is given for the ABORT event (see section 6).

action events
      nested_commit is permheap_nested_commit,
  toplevel_precommit is permheap_top_precommit
```

```
! The PERM_BLOCKENTRY type is used for the maintenance in the permanent heap of the
! list of free storage blocks. Each block is uniquely identified by its address.

type perm_blockentry is permanent new address

! The list of free storage blocks. Since the base type of this list is permanent,
! the list itself is allocated in permanent storage.
! This list may be modified only during the toplevel_precommit action event.
! The size of each entry is stored in the first word of that entry.

freelist : list( perm_blockentry ) := new list

! The BLOCKENTRY type is used in the declaration of the per-action variables
! below. Pointers to this type are allocated on the normal (not the
! permanent) heap, and may be modified outside of the toplevel_precommit
! event handler.

type blockentry is new address

! The per-action variables for permanent-heap recovery management.
! We will maintain lists of memory blocks allocated and freed by each action.

per action
  allocated : list( blockentry ) := new list
  freed : list( blockentry ) := new list
end per action

! When an action allocates a block of permanent storage, it must obtain a lock on that
! block until it commits to prevent other actions from attempting to allocate that block.
! Rather than associate a lock with the actual storage block, we lock the block's address
! (of type BLOCKENTRY). Recall that locks obtained by an action are propagated to its
! parent upon nested commit, and released upon abort or toplevel commit.

entry_lock : lock ( busy : [] ) domain is blockentry

procedure first_fit ( size : unsigned ) returns blockentry is
  ! A private operation of the PERMHEAP object. Given a size in words, FIRST_FIT finds
  ! the first entry on the FREELIST for a block of storage of size at least as large as
  ! SIZE and returns a pointer to that entry. (For the purposes of this example, we
  ! will assume that such a block exists.) Of course, another strategy could also be
  ! used here (such as best fit, or fragmentation and compaction). We'll assume that
  ! repeated invocations of FIRST_FIT by the same action return different addresses.
  begin
    ! The details of this operation are omitted here. Even if an appropriate block of
    ! storage is found on the FREELIST, FIRST_FIT must also test the ENTRY_LOCK to check
    ! whether this block has not already been allocated by some as yet uncommitted action.
  end procedure ! first_fit !
```

```
!
! ALLOCATE and FREE are public operations of the PERMHEAP object.
!

procedure allocate (! size : unsigned !) ! returns address ! is
  ! Return the address of a block of memory of the given SIZE in permanent storage.
  ! Since the block is from the FREELIST, its former contents are expendable.
  ! The Set_Lock operation used here is non-blocking, i.e., it returns immediately with
  ! value FALSE if the requested lock is not available.

  entry : blockentry

  begin
    loop   ! keep going until we find an available block
      entry := first_fit( size )
      if Set_Lock( entry_lock, busy, entry ) then
        Self.allocated@add( entry )  ! add the entry to the ALLOCATED list for this action
        return address( entry )
      end if
    end loop
  end procedure ! allocate !


procedure free (! block : address !) is
  ! Add a BLOCK of memory to the FREED list for freeing during toplevel precommit.

  entry : blockentry
  notthere : boolean
  i : unsigned := 1

  begin
    ! First, scan the ALLOCATED list to see if BLOCK was allocated by the current action
    loop
      entry := Self.allocated@nth( i, notthere )
      if notthere then
        exit .
      elsif entry = blockentry( block ) then  ! Yes,
        Self.allocated@remove( entry )        ! so remove it from ALLOCATED list
        ReleaseLock( entry_lock, busy, entry )
        return .                              ! we're done
      end if
      i += 1
    end loop

    ! If we get here, BLOCK wasn't allocated by the current action, so put it on the FREED list
    Self.freed@add( entry )
  end procedure ! free !


!
! The following are the alternate action event handlers for this object.
!

procedure permheap_nested_commit () is
  ! The alternate handler for the NESTED_COMMIT action event.  We'll propagate the items on
  ! the ALLOCATED and FREED lists of this action to the corresponding lists of its parent action.

  begin
    Parent.allocated@append( Self.allocated )
    Parent.freed@append( Self.freed )
  end procedure ! permheap_nested_commit !
```

```
procedure permheap_top_precommit () is
  ! The alternate handler for the TOPLEVEL_PRECOMMIT action event.  We'll traverse the FREED
  ! list, adding each entry there to the actual FREELIST in permanent storage; then, we'll
  ! traverse the ALLOCATED list, removing each entry there from the FREELIST.

  entry : blockentry
  notthere : boolean
  i : unsigned := 1

  begin
    ! Add each entry on the FREED list to the FREELIST in permanent storage
    loop
      entry := Self.freed@nth( i, notthere )
      if notthere then
        exit .
      end if
      ! Convert the entry to the permanent type before adding to FREELIST.
      freelist@add( perm_blockentry( entry ) )
    end loop

    ! Remove each entry on the ALLOCATED list from the FREELIST; the locks on these
    ! entries will be released automatically.
    loop
      entry := Self.allocated@nth( i, notthere )
      if notthere then
        exit .
      end if
      freelist@remove( perm_blockentry( entry ) )
    end loop
  end procedure ! permheap_top_precommit !


inithandler is   ! handler for the INIT (initialization) object event
  begin
    ! Perform initialization (not shown) of FREELIST to indicate that all
    ! of the permanent heap is available.
  end inithandler

!
! The REINIT object event handler is by default the same as the INIT handler.
! The DELETE object event handler for this object is by default NULL.
!

end implementation. ! permheap !
```

QUARTERLY PROGRESS REPORT
FAULT TOLERANT DISTRIBUTED COMPUTING
CONTRACT #MDA 904-86-C-5002
REPORTING PERIOD: 1 APR 86 - 30 JUNE 86

1. Project Status

During the past quarter, work has continued on each of the three project tasks. These efforts are closely related to other work in progress within the Clouds Project, our major research effort in the area of reliable distributed computing.

Under the Language Support for Robust Distributed Programs task, work continues in two major areas: the integration of the Aeolus compiler with the Clouds kernel services and the use of the Aeolus language system as a testbed for studying the problems of programming in action-object systems.

Under the Storage Management for an Action-Based Operating System task, the focus of our work has been on documentation of the design and implementation of the kernel storage manager and on implementation of a device driver to enable us to use our large disk drives on machines running the Clouds kernel.

Under the Operating System Support for Reliable Distributed Computing task, our efforts are directed toward specification and functional design of the operating system services which will be implemented on top of the object and action management mechanisms provided by the Clouds kernel. Our immediate focus is to obtain a working, robust kernel to provide a basis for the implementation of these designs.

The work on the tasks of this project is proceeding on schedule. Working in combination with other efforts in progress within the Clouds project, we are expect to have a working system by the end of the next quarter.

2. Language Support for Robust Distributed Programs

Work continues in the two major areas of the systems programming language effort: the design and implementation of the Aeolus language itself, as well as the use of the language as a testbed for the study of programming methodologies to achieve resilience and availability in action/object systems such as Clouds.

2.1 Language Design and Implementation

We consider the design of the Aeolus language to be essentially complete, and thus (barring the discovery of significant flaws) have frozen the design at its present stage. Therefore, we will be concentrating our efforts on the implementation portion of this task in the current quarter.

In our last report, we mentioned our recent paper[Wilk86] describing the rationale underlying the design of the Aeolus language. This paper has been accepted for presentation at the IEEE Computer Society 1986 International Conference on Computer Languages, and for publication in the conference proceedings. (A copy of the latest revision—based on the referees' comments—of this paper is attached as Appendix A.)

During the last quarter, the Clouds team member responsible for the Aeolus implementation was working on structure-editor generating systems at Siemens Research and Technology Laboratory in Princeton, NJ, under a cooperative arrangement between Siemens and the School of Information and Computer Science. Since his return to Georgia Tech at the beginning of July, progress has resumed on the implementation effort. We are now proceeding rapidly towards our goal of providing support for Clouds objects in the compiler by the end of the summer. We have developed a scheme for treating the Clouds object type information generated by the Aeolus as objects—called *TypeManagers*—under the Clouds kernel. The Aeolus compiler currently runs under Unix. Thus, when a Clouds object is compiled, a Unix "a.out"-style load file is created; the Unix header is then stripped from this file to yield a description for the object in the format expected by Clouds. A TypeManager, once created under a system running the Clouds kernel, requests this object description file from the Unix system and stores the description as the TypeManager's object data. Subsequently, when the "create" operation is invoked on the TypeManager, the object description is used to create an

instance of that object type. To create TypeManagers, we will "hard-wire" a TypeManager for TypeManagers into the kernel. A similar scheme will be used to create *ProcessManagers*, which will accept the code of Aeolus processes from the Unix system, and will also provide operations to activate and kill these processes. A TypeManager for ProcessManagers will also be "hard-wired" into the kernel. We are currently working with members of the kernel group to integrate these features into the Clouds kernel; other work proceeding concurrently includes adding runtime support for interfacing with object management, and the generation by the compiler of the data structures (such as tables of operation descriptors and entry points) needed by this runtime support.

2.2 Programming Methodologies for Action/Object Systems

In our last report, we described how our work on programming methodologies for action/object systems such as Clouds had led to some preliminary work on the design of a fault-tolerant job scheduler for the support of availability. During the last quarter, the scope of this investigation has expanded to include work on an object filing system for Clouds. This came about as we grew to realize that the replication scheme which we are currently considering in support of availability would require heavy interaction between the manager for a replicated object, the job scheduler, and the object filing system. The object filing system (OFS) should:

- be resilient and highly available (through replication);
- provide a mapping from object names (strings) to Clouds object capabilities;
- impose some familiar structure (e.g., a Unix-like hierarchical structure) on the flat, global system name space provided by the Clouds object manager;
- provide efficient forms for the most common types of I/O (such as text I/O) without the necessity of the context switches which would be required if such I/O were modelled with Clouds objects.

In the OFS, an object name may represent a *group* of objects (the set of replicas of a replicated object), rather than a single instance. We intend that this mechanism should be, in general, transparent to the user (although special-purpose applications, such as DBMSs, may require that, in addition, finer control of replication be available than that provided by a general mechanism).

We have found that the generality of the abstract object structure supported by Clouds poses problems for replication methods which are not presented by a less general, flat object structure (for instance, files or queues). The problem lies in the possibility of the arbitrarily complex *logical* nesting of Clouds objects. Although Clouds objects may not be *physically* nested (that is, one object may not physically contain another object), an object may contain a capability to another object. If an object *A* creates another object *B*, and retains sole access to *B*'s capability (by refraining from passing the capability to other objects and from registering the capability with the OFS), we say that object *B* is *internal to* object *A*. The internal object *B* may be regarded as being *logically* nested in object *A*. If, on the other hand, object *A* passes *B*'s capability to some object not internal to *A*, or if *A* registers *B*'s capability with the OFS, we say that *B* is an *external* object; an external object is potentially accessible by objects not internal to the object which created the external object.

Problems arise with replication schemes when internal and external objects are mixed together in the same structure, i.e., when an object may contain capabilities to both internal and external objects. These problems are associated with the method which is used to propagate the state of a replicated object among its replicas. One such method is to execute the computation from which the desired state results on each replica; we refer to this scheme as *idemexecution*. Another method is to execute the computation at one replica, and then copy the state of that replica to the other replicas; we refer to this scheme as *cloning*. Note that the scheme which is used to ensure that the replicas maintain consistent states (e.g., quorum consensus) is not involved in these problems, and is considered separately in our investigation.

External objects cause problems when idemexecution is used to propagate state among replicas. If the replicated object performs some operation on an external object (e.g., a print queue server), then—under idemexecution—that operation will be repeated by each replica. If the operation being performed on the external object is not idempotent, this can cause serious problems (e.g., multiple submissions of a job to the print queue). Also, trouble may arise due to idemexecution if the operation on the external object is non-deterministic (for instance, random number generation, or disk block allocation among multiple concurrent processes).

On the other hand, internal objects cause problems when cloning is used to propagate state. For example, assume that each replica of an object creates a set of internal objects. Then, when an operation is performed on one of the replicas, its state—under cloning—is copied to each of the other replicas. However, the capabilities to the internal objects of the replicas are contained in their states; thus, each replica now contains capabilities to the internal objects of that replica on which the operation was actually performed, and the information about the internal objects of the other replicas are lost.

Our current research includes an investigation of a "taxonomy" of object structures on which the corresponding state-propagation methods may be safely used, as well as of how these state-propagation methods—or the Clouds object-naming mechanism—may be altered to safely handle more general cases. Our current feeling is that the latter may be achieved with minimal alterations to the kernel, via having the kernel interact with the OFS and the job scheduler.

3. Storage Management for an Action-Based Operating System

The preceeding quarter saw the additional testing of the storage management system and the writing of a dissertation[Pitt86] which describes the storage manager.

Some additional functions remain to be implemented in the storage manager, primarily at the segment system level. These functions are for the most part cleanup routines. The core functionality of the storage manager, including recovery management, object memory management, and directory management have been implemented and tested.

Work is still proceeding on the RA81 device driver. A prototype driver is expected during summer quarter of 1986. The primary problems in developing the driver have been due to the sophistication of the interface to the drive, as well as to the complexity of the device itself. For the prototype, we have decided to postpone development of some functions, such as the bad-block-forwarding supported by the RA81. The addition of this facility to the RA81 device driver turned out to be much more complex than we had expected.

The dissertation "Storage Management for a Reliable Distributed Operating System"[Pitt86] was defended at the end of June. The dissertation describes the three major subsystems of the storage manager: the device system, the partition system, and the segment system. For each subsystem, the structures and operations that comprise the subsystem are defined. The dissertation describes the basic services provided by the storage manager: object memory support, recovery management, and directory management. The dissertation highlights the integration of virtual memory management with object memory support and recovery management. One of the claims of the dissertation is that this integration provides a efficient system.

The dissertation describes three algorithms that support the two-phase commit of actions in a Clouds system. It is shown how these algorithms support action management and also crash recovery. A chapter in the dissertation is devoted proving the correctness of these algorithms, based on the assumptions made for the Clouds system.

4. Operating System Support for Reliable Distributed Computing

The efforts under this task are aimed toward building operating system services on top of the Clouds kernel. Therefore the availability of the kernel is a crucial factor in the progress of this work.

The implementation of the Clouds kernel is nearing completion. The current status is as follows. The object management system has been tested to handle object invocations, both local and remote. This uses the communication system which uses Ethernet routines to communicate to other Clouds sites as well as Unix machines. The object management system uses a search and invoke strategy for locating objects in a uniform, location independent manner, that works even if some of the sites are non functional. The global searches occur efficiently as they use a hash table based decision function based on the Bloom filter (we call this the "Maybe Table").

The storage management system (Task 2) provides the functions of basic virtual memory, object memory, shadowing, flushing and commit. It also provides directory services for object lookup (using capabilities), and interfaces with the Maybe Table handling routines. This system has also been implemented and tested.

The current thrust is directed at integrating the object management system, the storage management system, and the communication system effectively to get an operational general purpose distributed operating system. With all the components tested individually, we expect the integration phase to last about two to three months. Currently we are using Unix machines to provide terminal access to the Clouds system over an Ethernet. The Unix systems are also providing developmental support for compilation of objects which are transferred to Clouds on demand over the network using some communication utilities that have been developed.

After the integration, we will start implementation of the Action Management system. The action management policies have been designed, but the implementation is not complete.

On the design side, the research has resulted in the design of several subsystems, notably a monitoring system and a distributed database system. The monitoring system fits into the Clouds reconfiguration strategies and uses a new mechanism called probes to monitor the health of the distributed system. The database is a conventional distributed database in a novel implementation environment. The object and action support provided by Clouds lend themselves effectively to implement a database system (modified to the object based structure) and provide concurrency control and recovery mechanisms in an environment that is simple to use.

The monitoring system designed makes use of probes. Probes are high priority messages in Clouds that can be sent to processes, action or objects. If sent to processes or actions, a probe causes a jump to a probe handler (similar to software signals). The probe handler generates a reply to the sender of the probe containing status information about the process or the action. The object probes work along similar lines, except that the probe causes the invocation of the probe handler in the object.

The monitoring system uses probes to monitor the health of critical system components. The monitors are replicated at each site and they keep status information in fully replicated databases. Each monitoring process has a backup monitor that monitors it from another site. Using this scheme we can keep good records of the global system state, and can handle failures by tying into the reconfiguration system and restarting failed actions at healthy sites. The design is reported in detail in [Dasg86]

The relational database system is an application environment under design to function in the object oriented environment supported by Clouds. Conventional database design suffers from two deficiencies. The data models proposed by database designers do not match the components supported by the operating system, and thus the implementors have to contrive mechanisms to support the database. Also the services (concurrency control, recovery) needed by databases are often not available and have to be built on top of a conventional operating system, giving rise to inefficient and often incorrect implementations.

The object oriented approach provided by Clouds allows relational databases to be encapsulated in objects and the implementation matches both the environment as well as the data model, giving rise to better performance, clean elegant systems interfaces and a modular implementation. The synchronization and recovery support provided by Clouds also effectively provides database services giving rise to easier to implement database management functions.

Fine granularities of locking structures can be attained by relation fragmentation, that gives rise to more efficient access strategies. But as the objects hide the fragmentation details, the interfaces are just as clean and transparent. Further details can be found in [Dasg86a]

REFERENCES

[Dasg86] Dasgupta, P. "A Probe-Based Fault Tolerant Scheme for the Clouds Operating System." TECHNICAL REPORT GIT-ICS-86/05, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[Dasg86a] Dasgupta, P., and M. Morsi. "An Object-Based Distributed Database System Supported on the Clouds Operating System." TECHNICAL REPORT GIT-ICS-86/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[Pitt86] Pitts, D. V. "Storage Management for a Reliable Decentralized Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as Technical Report GIT-ICS-86/21.)

[Wilk86] Wilkes, C. T., and R. J. LeBlanc. "Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System." TECHNICAL REPORT GIT-ICS-86/12, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (To be presented at the IEEE Computer Society 1986 International Conference on Computer Languages.)

Rationale for the Design of *Aeolus*: A Systems Programming Language for an Action/Object System

Technical Report GIT-ICS-86/12

C. Thomas Wilkes
Richard J. LeBlanc

The Clouds Project

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280
(404) 894-3152

ABSTRACT

The goal of the *Clouds* project at Georgia Tech is the implementation of a fault-tolerant distributed operating system based on the notions of *objects*, *actions*, and *processes*, to provide an environment for the construction of reliable applications. The *Aeolus* programming language developed from the need for an implementation language for those portions of the Clouds system above the kernel level. Aeolus has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the Clouds notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the Clouds system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as Clouds.

In this paper, the features provided by the language for the support of readability and maintainability in systems programming are described briefly, as is the rationale underlying their design. Considerably more detail is devoted to features provided for support of object and action programming. Finally, an example making use of advanced features for action programming is presented, and the current status of the language and its use in the Clouds project is described.

July 21, 1986

This report is a preprint of a paper to be presented at the IEEE Computer Society 1986 International Conference on Computer Languages, Miami Beach, FL, October 1986.

Rationale for the Design of *Aeolus*: A Systems Programming Language for an Action/Object System

Technical Report GIT-ICS-86/12

C. Thomas Wilkes
Richard J. LeBlanc

The Clouds Project

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280
(404) 894-3152

1. Introduction

The goal of the *Clouds* project at Georgia Tech[Allc82, Allc83, Allc83a] is the implementation of a fault-tolerant distributed operating system based on the notions of *objects*, *actions*, and *processes*, to provide an environment for the construction of reliable applications. The *Aeolus*[1] programming language developed from the need for an implementation language for those portions of the Clouds system above the kernel level. Aeolus has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the Clouds notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the Clouds system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as Clouds.[LeBl85, Wilk86]

The intended users of Aeolus are systems programmers working on servers for the Clouds system. Clouds provides powerful features for the efficient support of resilient objects where the semantics of the objects are taken into account; it is assumed that the intended users have the necessary skills to make use of these features. Thus, although access to the automatic recovery and synchronization features of Clouds is available, we have avoided providing very-high-level features for programming resilient objects in the language, with the intention of evolving designs for such features out of our experience with programming in Aeolus. These features will then be incorporated into an applications language for the Clouds system, which should allow programmers unskilled in fault-tolerant programming to write resilient objects.

Aeolus has its roots in a long line of structured programming languages, including Simula, Pascal, Modula-2, and Ada™. Thus, many of its features should be easy to understand for those familiar with one of these languages, and little space will be devoted here to discussion of such features; a description of the complete language is available in the Reference Manual.[Wilk85] Syntax and examples will be provided here only for those features of Aeolus which differ significantly from those of its predecessors, although the programming example discussed in section 6 should provide a feel for the flavor of the language.

---

1. Aeolus was the king of the winds in Greek mythology.

™ Ada is a registered trademark of the U.S. Government—Ada Joint Program Office.

An overview of the Clouds system from the point of view of Aeolus has been provided in a previous paper.[LeBl85] Briefly, the main structuring features of Aeolus (as of the Clouds system) are objects, actions, and processes. Clouds supports the *object* concept as a convenient structuring principle for facilitating recovery and synchronization. An object encapsulates data and provides operations to access that data; the object's data may be manipulated only via its operations, thus helping maintain the invariants of the object. Besides supporting the Clouds object concept, Aeolus also allows the programmer to use the object features of the language for the specification of abstract data types, without necessarily invoking the object and action management features of the Clouds system. Thus, Aeolus objects provide a separate compilation facility as well as access to the object support of Clouds; the separation of object specifications into *definition* and *implementation* parts (much as are *modules* in Modula-2 or *packages* in Ada) provides a safe interface to separately-compiled objects, as well as facilitating the design of large systems consisting of many objects (possibly implemented by several people) or the use of predefined objects. Aeolus *pseudo-objects* provide a means of isolating system dependencies—such as input/output or low-level machine architecture—into object-like modules which provide operations facilitating machine-level programming.

The Clouds notion of *actions* corresponds roughly to the transaction concept of distributed database work, providing an "all-or-nothing" assurance of atomic execution (a property sometimes called *failure atomicity*). Support of actions in Aeolus is fairly low-level. Essentially, means are provided for specifying that an operation invocation is to be executed as a toplevel or nested action. Also, the status of an action execution may be checked or altered by means of calls to a Clouds action manager. In Clouds, we distinguish between operations *invoked as an action* and operations *executing in an action environment*. We say that an operation is executing in an action environment if that operation is invoked as a toplevel or nested action, or if the invoker of the operation is executing in an action environment. Thus, it is a sufficient, but not a necessary, condition that an operation be invoked as an action to be running in an action environment. (Some transaction systems require the creation of one or more nested actions to encapsulate every remote operation invocation. In Clouds, such encapsulation is not required, but is available at the programmer's option.)

The *process* concept in Aeolus is similar to the *program* construct of Pascal or Modula-2. That is, a process ties together the constituent parts (objects) of a programmed system, and the invocation of a process provides activity in the Clouds system, since Clouds objects are passive.

In this paper, the features provided by the language for the support of readable, maintainable systems programs are described briefly, as is the rationale underlying their design. Considerably more space is devoted to the detailed description of features provided for the support of object and action programming. Finally, an example making use of advanced features for action programming is presented, and the current status of the language and its use in the Clouds project is described.

2. Support for Systems Programming

In this section, those features of Aeolus which are provided for the support of readable, maintainable systems programs, and which are not directly related to the support of action/object programming, are described briefly. These include structured types for access to low-level data representation, expression and statement constructs necessary for doing systems programming in a concurrent environment, and the typing mechanism.

2.1 Access to Low-Level Data Representation

Aeolus provides a wide range of traditional type classes, including enumerations, pointer types, structured types, and locks. (Objects are also treated as types in Aeolus, as will be described in section 3.) Of interest here are those structured types providing access to the low-level representation of data, as is often required in systems programming, and those constructs providing support for synchronization in a concurrent environment.

Aeolus provides the traditional structured types, such as arrays, records, and sets. All types in Aeolus may be *parameterized* (see below); for example, the parameterized record type in Aeolus is similar in concept to the *discriminated record type* of Ada. The extension of parameterization to other types allows type constraints to be expressed in terms of parameters, and allows parameterized array types to be declared without the necessity of embedding them in record types. The parameterized array construct indirectly provides support for arrays with a flexible number of elements (so-called "dynamic" arrays); these may be simulated by using pointers to parameterized arrays. Two important parameterized array types provided by the Aeolus implementation are *strings* and *bitstrings*. The string type allows convenient manipulation of character sequences. The bitstring, on the other hand, provides the most primitive structured abstraction of data, that of simply a sequence of *bits*:

```
type bitstring( length : unsigned ) is array [ unsigned[ 1..length ] ] of bit
```

The length constraint of the bitstring (in bits) is indicated by the value of the parameter "length." For example:

```
type nybble is bitstring( 4 )
```

Each "system" object[2] provides declarations of several useful bitstring types. These types are referred to collectively as the *storage classes*, since they define the units of storage supported by the hardware of most computer systems: types *bit*, *byte*, *word*, *longword*, and *quadword*, with lengths BITSIZE, BYTESIZE, WORDSIZE, LONGWORDSIZE, and QUADWORDSIZE, respectively.

Another important bitstring type, *address*, is also defined by the "system" object. The address type is defined as *bitstring(ADDRESSSIZE)*. The relationship between address types and pointer types is discussed below.

Several operations are provided for manipulation of bitstring data, including bitwise Boolean operations and shift operators. Access to individual bits of a bitstring is, of course, through array index operations. The provision of a true bitstring type addresses a lack experienced by users of both the Ada and C languages.[Evan84]

Another structured type providing access to low-level data representation is the *structure*, a special case of a record construct somewhat similar to the *packed record* construct of Pascal or the *packed* pragma as applied to the record construct (with representation specification) in Ada. The declaration of a structure type specifies the storage class which the structure will fit. A field in a structure typically represents a bitstring or scalar; the fields are packed together consecutively within an object of the specified storage class (without implicit padding), with the first field specified starting at the most significant bit position in the storage class. The compiler checks that the fields declared for the structure together fit into the specified storage class.

2.2 Constructs Supporting Synchronization

Features provided by Aeolus for the support of synchronization and mutual exclusion in a concurrent environment include the *lock* construct, the *region* construct, and the *shared* type attribute.

A *lock* type may be used to declare variables which in turn may be used to implement locking protocols on particular *values* in some domain. Note that an Aeolus/Clouds lock is obtained for a value of an object, and not on the object itself. Thus, for instance, a lock may be

2. There exists a "system" pseudo-object for each computer system for which the Aeolus compiler is implemented. (At present, the "system" objects include *VAX_System* and *PC_System*, for the DEC VAX 11/780 and the Intel 8086-8088-80286 families of computers, respectively.) Each such object defines system-dependent constants, types, and operations required for systems programming. The appropriate object (determined by target code generation and controllable by compiler option) is imported implicitly by every compiland.

obtained on a file name even if that file does not yet exist. (The lock structure is directly supported by the Clouds architecture.) A lock declaration includes the specification of a *compatibility list*, which defines, for a given *mode* of the lock, which other modes are compatible with that mode. A lock may be set with a specified mode only if other modes already set, if any, are compatible with that mode. (Thus, a process adhering to a protocol using that lock may wish to block until the requested mode is available. Operations are provided by object *standard* for testing, setting, and releasing locks.) The presence of an identifier in a compatibility list serves as a declaration of that identifier as a mode of the lock type; the modes of a lock type may together be thought of as an enumeration. An empty compatibility list indicates that the given mode is incompatible with all other modes.

The lock declaration may also specify the *domain* of values which may be locked. If the domain specification is omitted, a simple lock (i.e., one which does not lock over any particular domain) is assumed. For example:

```
type simple_lock is lock (   busy : []          )
type file_lock is lock (     read  : [read]     ,
                             write : []         ) domain is string( 20 )
```

The declaration of "simple_lock" above defines a lock type with a single mode "busy" which is incompatible with itself; thus, only one client may set a lock variable of type "simple_lock" at any one time. The declaration of "file_lock," on the other hand, defines a lock type over the domain of strings of length 20. Clients may set a lock variable of type "file_lock" on a given string with modes "read" or "write." The "read" mode is specified as being compatible with other settings of "read" mode; the "write" mode is incompatible with itself and with "read" mode. Thus, a client may set the lock with "read" mode on a given string even if several other clients have outstanding settings of the lock with "read" mode on that string; however, a client wishing to set the lock with "write" mode on a given string must wait for all outstanding settings of "read" mode on that string to be released.

All locks obtained during execution in the environment of a nested action are retained and propagated to the immediate ancestor of that action upon committal unless explicitly released by the programmer. Locks obtained under an action are automatically released if the action aborts or successfully performs a toplevel commit. Thus, a two-phase locking protocol (2PL) is maintained, with violations to 2PL allowed (via explicit release of locks) if the programmer deems such violations acceptable. A lock is available to be granted under a nested action even if conflicting locks are held under one or more of the ancestors of that action, but not if conflicting locks are held under an action which is not an ancestor of the nested action.

The power of the Aeolus/Clouds lock construct in supporting user-defined synchronization lies in the specification of arbitrary locking modes, and arbitrary compatibilities between those modes, as well as the dissociation of locks from the locked data. To support mutual exclusion, Aeolus provides a *critical region* construct, access to which is controlled by association of the region with a designator for a *shared variable*. The shared variable is associated with a semaphore, yielding the familiar semantics of critical regions. In Aeolus, any type may have the attribute *shared*, which is inherited by any types of which the shared type is an element. In particular, Aeolus arrays may consist of shared elements; thus, the granularity of mutual exclusion may be tailored to achieve higher concurrency.

2.3 Type Compatibility and Conversion in Aeolus

The principal goal in the design of the Aeolus typing mechanism was the provision of strong typing where possible, but also the provision of escapes from strong typing where the special demands of systems programming required. Another concern which affected the design of these mechanisms was that programs be readable and maintainable by members of a fairly rapidly-changing research group. Thus, the desirability of brevity of notation was felt to be subordinate to that of rapid comprehension of code by readers (including the original authors of

the code). Thus, we attempted to make the typing mechanism as simple as possible, simplifying the tasks both of the compiler and of the reader, who must otherwise remember numerous compatibility and implicit conversion rules, increasing the possibility of misunderstanding or confusion.

The type of an operand is said to be compatible with that required by an operation if they are the same type, that is, if the types are *equivalent*. Type equivalence in Aeolus is by name.

As in Ada, a constraint which is associated with a scalar type (by way of a constraint specification in the type's declaration, or via a constraint declaration) is not considered part of that type, but rather is an attribute which is given to a member of that type.[3] Thus, a constrained member is compatible with a member which has the same type but a different (or no) constraint.

Types may be *parameterized*, that is, some of the attributes of a type may depend on the values of *formal type parameters*. (Object types may also be parameterized; see section 3.) In contrast to constraints, the parameters of a type as specified in the declaration of that type are, in general, considered to be a part of that type. The exception is when the specification of these parameters is delayed (via an empty actual type parameter option). (A member of a type with delayed parameterization is sometimes said to be associated with a *delayed constraint*.) A type with delayed parameterization is compatible with all possible parameterizations of that type. Types with delayed parameterization, when used as the types of formal procedure parameters, make possible generic operations on structured types such as arrays; and when used as pointer base types, allow the definition of pointers to arrays with flexible sizes.

The requirement for strict name equivalence of types is relaxed somewhat in the case of array slices, since slices, by their nature, cannot be associated with a named type. Rather, a slice is similar to a member of an unconstrained array type, any type parameter values of which are derived from the slice bounds, and which takes on as its base type the base type of the named type of the array from which the slice was derived. An array slice with *n* elements is compatible with any array or array slice with *n* elements and a compatible element type. Also, a slice of one element is compatible with any variable of a type compatible with the element type of the slice. Note that this implies the following correspondences:

```
bit   <=>  bitstringslice[1..1]  <=>  bitstring(1)
char  <=>   stringslice[1..1]    <=>  string(1)
```

Thus, bit is compatible with array [integer[1..1]] of bit; char is compatible with array [integer[1..1]] of char; and, in general, type *t* is compatible with an array of one element of element type *t*.

In the interest of keeping the implementation effort for Aeolus within reasonable bounds, it was decided not to provide facilities for the specification of overloading of user-defined operations in the current language. However, certain overloadings are available on predefined operators. In keeping with the goal of simplicity in the typing mechanism as stated above, the overloading of a binary operator is available only for operands which are compatible according to the definitions stated above. As seen from another point of view, this means that Aeolus does not perform implicit conversions. However, it is sometimes desirable to perform operations on operands of differing types. Thus, Aeolus provides the programmer with powerful means of explicit type conversion. Explicit type conversion functions are defined between members of closely related types within certain limitations.[Wilk85] In general, the name

3. Constraints are used for range checking (if enabled) and for determining the sizes of structures, but not for type checking.

of a type may be used as the name of a conversion function; this type is the target type of the conversion.

Explicit conversions are allowed between types one of which is derived from the other (derived types), between different numeric types, between enumeration and numeric types, from an enumeration type to a string type, and between array types each of which meets conditions similar to those specified by Ada. Also, conversion is allowed (in both directions) between a type which is a bitstring type and any type with the same size (in bits) as the source type. In particular, conversions may be made between any array, record, or structure type and a bitstring type or array of bitstring type (e.g., array of byte or word) with the same size. Thus, access may be obtained to the bit representation of data in an explicit manner. Also, conversion is allowed from any pointer type to type *address*. In the other direction, conversion is allowed from type *address* to any pointer type; however, the result of such a conversion may be used only for dereferences, and may not be assigned to a pointer variable. Values may be assigned to address variables directly, by conversion from a pointer type, or via the *addr* operation, which yields the storage address of a static or dynamic data item; a value may be assigned to a pointer variable only by use of an allocator, or via assignment from another variable of the same pointer type. Thus, a safe (although restrictive) pointer mechanism is maintained separately from a permissive mechanism permitting address computations when necessary.

Finally, Aeolus provides a less restrictive (and less safe) means of type conversion in addition to the explicit (checked) conversion functions described above; the *retype* function is similar in spirit to the *unchecked_conversion* function of Ada. Of course, unchecked conversion may be used to convert addresses to any pointer type, thus violating the safety of the pointer mechanism. The intent of the *retype* function is to make such "end runs" around the typing mechanism obvious to the reader of the code, and when used with restraint and care it makes possible the sort of generic bookkeeping activity necessary in systems programming (e.g., memory allocation routines).

3. Support for Objects

The object construct provides support for *data abstraction* in Aeolus. A collection of related data items may be *encapsulated* within an object, which also may provide *operations* (procedures that operate) on the data. The only access to the data of an object is via these operations; thus, an object can strictly control manipulation of its encapsulated data, helping guarantee the invariants of the abstraction.

The object concept is supported at the lowest levels of the Clouds kernel; hence, we feel justified in using the term "object" to describe the data-abstraction facility of Aeolus, since an Aeolus/Clouds object has a real existence in the system. However, Aeolus does not provide a sophisticated inheritance mechanism such as that available in Smalltalk, nor does it provide for dynamic typing of objects. Aeolus provides a simple class mechanism in the *object type* described below; all *instances* of an object type inherit all operations (and other definitions) of that type. It was felt that, although an inheritance mechanism providing differential sharing of object operations would be useful, the support for such a mechanism should be left to higher-level portions of the system in order to keep the kernel as simple as possible; thus, inheritance will be among the features to be included in the language design to be evolved from our experience with Aeolus, as mentioned in the Introduction. Also, communication between objects in Aeolus is based on direct operation invocation rather than on message-passing, reflecting the fact that Clouds is not a message-passing system, but uses remote procedure call to support distributed computation. Hence, Aeolus is not strictly an *object-oriented language* in the sense of Rentsch.[Rent82] However, it provides access to an object concept supported throughout the Clouds system. The applications language to be based on our experience with Aeolus will likely come closer to the concept of "object-oriented language" in the strict sense.

An Aeolus object may have parameters indicating, for instance, sizes or element types of the abstraction implemented by the object; thus, an object implementing, for instance, a bounded stack abstraction may be parameterized by the element type and maximum number of elements of the stack. Then, various *instances* of the bounded stack object may be created (instantiated) with differing element types and sizes; the implementation of the object need not be concerned with details such as the element representation, and the programmer does not need to create new object types for each combination of element type and stack size. Support for such *generic objects* increases the level of abstraction available to the programmer, and makes possible the creation of libraries of reusable object types, in a spirit similar to that of the *generic package* construct of Ada.

The object construct also provides a safe separate compilation mechanism. The separation of an object specification into a *definition part* and an *implementation part* allows checking across the interface to an object, as well as allowing the use of an object definition before the corresponding implementation part is finished (thus facilitating top-down design).

3.1 Object Definition Parts

The definition part of an object defines the interface of the object with other compilands. It specifies the attributes of the object itself as well as the constants, types, and operations which the object provides to other objects and to processes. (Note that variables may not be declared in object definition parts; it was felt that the sharing of variables between objects was not in accord with the principle of data encapsulation embodied by the object construct, which requires that all access to object data be through operations on the object. Also, there is no counterpart to the *class variable* construct of Smalltalk, that is, a variable which is shared by all instances of an object type; it was felt that this would violate the principle[Ens178] that a fully-distributed system should have no shared memory.) The declaration of the object name in the header of the object definition defines a type, called an *object type*, with that name, which may be used in the declaration of variables to hold capabilities to *instances* of that object type (see below). An example of an object definition part is included in the Appendix.

Specification of the *autosynch* keyword in an object definition header causes code to be generated for automatic synchronization of object operation invocations based on programmer-supplied indications of operation effects (see below). This mechanism provides a simple read/write locking protocol; it may be used with any object class (see below).[4]

The *object class* is also specified in the object definition header. The object classes fall into two groups: the non-Clouds object classes (*pseudo* and *local*) do not use any of the Clouds facilities for action or object management, and are thus similar to *modules* in Modula-2 (for pseudo-objects) or to generic packages in Ada (for local objects), while the so-called Clouds object classes (*nonrecoverable*, *recoverable*, and *autorecoverable*) may make use of the object management facilities and (for recoverable and autorecoverable types) the action management facilities. Thus, the rationale behind the non-Clouds object classes is the same as that underlying the design of the corresponding features in Ada or Modula-2, that is, the provision of data-abstraction facilities usable "locally" (without resorting to operating system facilities). On the other hand, the Clouds object classes provide access to the support for data abstraction provided by the Clouds system when the expense of that support is warranted; the separate classes of Clouds object allow the programmer to specify the degree of support (and of incurred expense) required. The definitions of the object classes are as follows:

4. For more information on the mechanisms supplied by the Clouds system to support synchronization and recovery, see Allchin's dissertation.[Allc83a].

non-Clouds object classes:

pseudo (or pseudo-local) There may exist only one instance of a given pseudo-object type. This class of objects is used mainly for definition of system libraries, for interfacing with (separately-compiled) collections of procedures written in another programming language, for abstraction of machine and system dependencies, and as a basic separate-compilation mechanism.

local The standard class of non-Clouds object, which may have multiple instances. Management of local objects is provided by the Aeolus runtime system. Unlike Clouds objects, a local object may have no existence independent of the process or object which created it. Local objects simulate Clouds objects without incurring the expense of the use of the action and object management facilities.

Clouds object classes:

nonrecoverable The basic class of Clouds object. Objects of class *nonrecoverable* make use of the object management facilities, but may not contain features requiring action management, such as recoverable areas, permanent and per-action variables, or action event handlers (see section 4).

recoverable The "roll-your-own recovery" type of Clouds object, as opposed to the *autorecoverable* class of objects (described below), which provides completely automatic recovery. In some cases, the programmer may be able to use knowledge of the semantics of the object and its operations to program synchronization and recovery mechanisms more efficient than the automatic mechanisms supplied by the *autorecoverable* class of objects. Automatic recovery involves checkpointing of the entire object state; automatic synchronization is based on a simple read-write model of operation interactions on entire operations. As will be discussed in section 4, Aeolus provides facilities that allow the programmer to specify which parts of the object state are to be checkpointed (recoverable areas), to access information about the states of actions and to change these states (via operations on the action manager), and to control the recovery process by specification of what is to be done during action events (action event handlers); also, the programmer may specify finer-grained locking mechanisms for greater control of synchronization (via the *lock* type; see section 2). Only *recoverable* objects may contain recoverable area specifications and action event handler specifications.

autorecoverable As mentioned above, *autorecoverable* objects provide completely automatic recovery. The entire object state (the global variables of the object) is recoverable, and the default event handlers are used.

Operations on objects of class *recoverable* or *autorecoverable* may be executed only within an action environment; this restriction will be explained further in section 4.4. An instance of an object (other than of class *pseudo*) is created by use of an *allocator*, a construct also used for allocation of pointer values (and similar to that used for allocating "access objects" [pointers] in Ada). This underscores the similarity in treatment between object capabilities and pointers, in particular, the processes of creation, initialization, and deletion (disposal), as well as their use as elements in lists and other structures and as parameters to objects and object operations. The values of any object parameters must be specified by using a constructor in the allocator. The allocator yields a capability to the newly-created object instance, which may be assigned to a variable of that object type. The variable may thereafter be used to qualify operation invocations on that object instance. The *init* object event handler (see below) for the object is executed during the instantiation process, as are any variable initializations required by the object.

The definition part also performs any necessary imports of other object definitions before the declarations of the object definition are given. These are called its *visible declarations* since the declarations are available to any object which imports the object definition. As stated above, the visible declarations of an object may include specifications of constants, types, or operations, but not of variables. Finally, specifications of the object's operations are provided. An operation specification may optionally be given one of the attributes *examines* or *modifies*, which indicate that the operation reads from or writes to the object's state, respectively. This information is used by the compiler to generate automatic read or write locking for each operation if the *autosynch* attribute is specified for the object. If no operation effect is specified, the compiler assumes that the operation neither reads nor modifies the object state, and thus no automatic locking would be done for that operation. The *autosynch* feature thus provides automatic synchronization according to a simple multiple readers/single writer protocol. An object operation (or other procedure) meeting certain conditions[Wilk85] may also be given the *inline* attribute, indicating that inline code expansion of the operation is desired; thus, the use of operations to access portions of encapsulated data can be made more efficient. We have found in our experience that when complicated data structures are encapsulated (such as compiler type attribute records), the number of operations which exist merely to provide controlled access to small portions of the hidden data structure becomes very large; the efficient support of such operations thus becomes important to encourage the use of encapsulation.

3.2 Object Implementation Parts

The implementation part of an object provides the actual code for the operations of the object, as well as the definitions of any private constants, types, variables, or procedures needed by the object. (An example of an object implementation part is provided in the Appendix.) These definitions are, of course, hidden from other compilands; only those definitions specified in the object definition part are available outside the implementation part. This supports the hiding of encapsulated data required by the data-abstraction feature, and is similar in concept to Ada package and Modula-2 module implementations.

The definition part of the object being implemented is implicitly imported by the implementation part; also, any objects imported by the definition part are available in the implementation part. The implementation part may import other objects as well via its own import clauses. All constants, type definitions, and operations declared in the objects made available by any of these methods are visible in the implementation part; also, the names of these imported object types may be used as the types of variables declared in the implementation part. Such variables must be initialized by use of an allocator.

Variables declared in the outer level of the block of the object implementation part are global to the object, and the values of such variables survive between invocations of the object's operations. The global variables of an object are called collectively the object's *state*. In an object of class *recoverable*, part of the object state may be specified to be in a *recoverable area;* also, the programmer may specify an *action events part* and/or a *per-action variables part*. Recoverable areas, action events, and per-action variables are described in section 4.

In order to allow the object to participate in its own creation and deletion, an object implementation part contains specifications of handlers for the so-called *object events*. The object events include the *init* or object initialization event, the handler for which is executed whenever an instance of the object is created by use of an allocator; the *reinit* or object reinitialization event, the handler for which is executed—if the object has registered its desire for reinitialization with the action manager—when the system is reinitialized after a crash or network partition; and the *delete* or object deletion event, the handler for which is executed when the object instance is destroyed. No default handler for the *init* object event is assumed; if no action is desired for the *init* event, the programmer must supply a NULL statement as the handler body. The intent is to help prevent the accidental omission of object state initialization by the programmer. If no handler for the *reinit* object event is specified, the handler is by default the same as that specified for the *init* event. If no handler for the *delete* object event is

specified, it is assumed to be NULL.

3.3 Object Operation Invocations

An invocation of an object operation looks much like a procedure invocation, except that, outside the implementation part of the object itself, an operation name must be qualified by the name of a variable representing an instance of that object type (or, for pseudo-objects, by the name of the object type itself). Thus, for an instance of a bounded-stack type, we may have

```
stack_instance @ push( elem )
```

When an object invokes one of its own operations, however, the usual procedure call syntax is used.

Invocations of pseudo-object and local object operations have semantics essentially similar to those of calls to procedures local to a compiland. The situation is different for operations declared in objects which use the Clouds object-management facilities (i.e., the so-called "Clouds objects").[5] Invocations of operations on Clouds objects are handled by the compiler through operations on the Clouds object manager on the machine on which the invoking code is running. The Clouds object on which the operation is being invoked need not be located on the same machine as the invoking code; the object manager then makes a *remote procedure call* (RPC) to the object manager on the machine on which the called object resides. The location—local or remote—of the object being operated upon, however, need not concern the programmer, as the RPC process is transparent above the object-management level.

4. Support for Actions

The action concept provides an abstraction of the idea of work in the Clouds system; an action represents a unit of work. Actions provide *failure atomicity*, that is, they display "all-or-nothing" behavior: an action either runs to completion and *commits* its results, or, if some failure prevents completion, it *aborts* and its effects are cancelled as if the action had never executed. The rationale for the action concept and the mechanisms supporting it in the Clouds system are described in Allchin's dissertation;[Allc83a] the design for the implementation of these mechanisms is described in Kenley's thesis.[Kenl86]

Support for actions in the Aeolus language is relatively low-level. The methodology of programming with actions is not at present well-understood compared with experience in programming with objects; thus, rather than providing high-level syntactical abstractions such as those available for object programming, Aeolus allows access to the full power and detail of the Clouds system facilities for action management. The major syntactic support provided by Aeolus for action programming is in the programming of *action events*, *recoverable areas*, *permanent* and *per-action variables*, and *action invocations*.

4.1 Action Events

At several points during the execution of an action, the action interacts with the *action manager* of the Clouds system to manage the states of objects touched by that action, including writing those states to *permanent* (stable or safe) storage, and recovering previous permanent states upon failure of an action. Thus, failure atomicity may be provided by the action

5. This is because the code for pseudo-objects and for local objects is actually linked into the code of the compiland using these objects, whereas the code for Clouds objects is physically separate from the code of the invoking compiland. This code is paged in on demand by the object manager; see Allchin's dissertation.[Allc83a]

management system. The *action events* include:

event name	*purpose*
BOA	beginning of action
toplevel_precommit	prepare for commit for a toplevel action
nested_precommit	prepare for commit for a nested action
commit	normal end of action (EOA)
abort	abnormal end of action

The interactions with the Clouds action manager necessary when such events take place are done by default procedures supplied by the Aeolus compiler and runtime system; these procedures are called *action event handlers*. When an action event occurs for a particular action, the action manager(s) involved invoke the event handlers for each object touched by that action.

As was described in section 3, by use of the *autorecoverable* class of object, the programmer may take advantage of the recovery facilities of the Clouds system by having the compiler generate the necessary code automatically. This automatic recovery mechanism requires checkpoints of the entire state of the object, and uses the default action event handlers. However, it is sometimes possible for the programmer to improve the performance of object recovery by providing one or more object-specific event handlers which make use of the programmer's knowledge of the object's semantics; these programmer-supplied event handlers then replace the respective default event handlers for that object. Thus, if object class keyword *recoverable* is specified in the definition header of the object being implemented, the programmer may give an optional *action event part* in the object's implementation part. Following the keywords *action events*, the programmer lists the name of each action event handler provided by the object implementation as well as the name of the action event whose default handler the specified handler is to override. Thus, for example, the specification (say, in an object implementing a bounded-stack abstraction):

```
action events
  stack_BOA overrides BOA,
  stack_nested_precommit overrides nested_precommit
```

indicates that the default handlers for the *BOA* and *nested_precommit* action events are to be replaced by the procedures named "stack_BOA" and "stack_nested_precommit," respectively, for the bounded-stack object type only.

4.2 Recoverable Areas

As mentioned in section 3, if an object being implemented is of class *recoverable*, then some of its variables may be declared in a *recoverable area*. When a nested action first invokes an operation on a recoverable object ("touches" that object), the action is given a new *version* of the recoverable area which initially has the same value as the version belonging to the action's immediate ancestor. The set of versions belonging to uncommitted actions which have touched a recoverable object is maintained on a *version stack* by a Clouds action manager. When a nested action commits, its version replaces that of its immediate ancestor. When a toplevel action commits, its version is saved to permanent storage. If an action is aborted, its version is popped from the version stack.[6] Thus, recoverable areas (in conjunction with appropriate use of synchronization) provide *view atomicity*, that is, an action does not see the intermediate (uncommitted) results of other actions. Also, the use of recoverable areas allows the

6. For more information on the semantics of recoverable areas and the mechanisms to support them, see Allchin's dissertation.[Allc83a]

programmer to provide finer granularity in the specification of that part of the object state which must be checkpointed, since the use of automatic recovery on object (the *autorecoverable* object class) performs checkpoints on the entire state of the object.

The interaction with the action manager necessary to manage the states of recoverable areas is implemented by the action event handlers as described above. Again, the default event handlers may be overridden by programmer-supplied event handlers for the entire object to achieve better performance.

4.3 Permanent and Per-Action Variables

It may sometimes be desirable to make large data structures resilient. In such cases, the recoverable area mechanism may be inefficient, since it requires the creation of a new version of the entire recoverable area for each action which modifies the area. Often in such cases the programmer make take advantage of knowledge of the semantics of the data structure to efficiently program the recovery of the data structure. The Aeolus language provides two constructs which aid in the custom programming of data recovery, the so-called *permanent* and *per-action variables*, constructs proposed by McKendry.[McKe85]

Any type may be given the attribute *permanent*. This attribute indicates that members of that type are to be allocated on the *permanent heap*, a dynamic storage area in the object storage of each object instance. This area receives special treatment by the Clouds storage manager; in particular, it is *shadow paged* during the *toplevel precommit* action event.[7] Any type which has as its base or element type a type with the attribute *permanent* inherits that attribute. Other than during object initialization, permanent variables may be assigned values only within a *toplevel precommit* event handler.

Aeolus also provides the per-action variable construct. An object implementation part of class *recoverable* may declare a single per-action variable section. A per-action specification resembles a recoverable area specification, and the semantics is also similar, in that each action which touches an object with per-action variables gets its own version of the variables; however, the programmer may access the per-action variables not only of the current action, but also of the parent of the current action. Also, per-action variables are allocated in *volatile storage*, that is, in storage the contents of which may be lost upon node failure. The variables in a per-action specification are accessed as if they were fields in a record described by the specification; two entities of this "record type" are implicitly declared: Self and Parent, which refer respectively to the per-action variables of the current action and its immediate ancestor.

Permanent and per-action variables may be used together to simulate the effect of recoverable areas at a much lower cost in space per action. In general, the per-action variables are used to propagate changes to the resilient data structure up the action tree; these changes are then applied during the *toplevel precommit* action event to the actual data structure in permanent storage. The use of permanent and per-action variables is shown more fully in the programming example shown in the Appendix (and described in section 6).

4.4 Action Invocations

The right-hand side of an assignment statement may take the form of an *action invocation*. Here, the right-hand side (which consists of an operation invocation which, if the operation is value-returning, is embedded in another assignment statement) is invoked as an action; the *action ID* of this action is assigned to the variable designated by the left-hand side of the action invocation. Thus, for example, if the bounded-stack object mentioned in section 3 were defined as a recoverable object, we might invoke one of its operations as an action:

---

7. More information on the management of permanent heap storage is available in several papers on the Clouds system.[Pitt85,Keni86,Wilk86]

```
aID := action( stack_Instance @ push( elem ) )
```

The action ID may be used as a parameter in operations on the action manager which provide information about the status of the action, cause a process to wait on the completion of an action, or explicitly cause an action to commit or abort.[8] The programmer may specify that an action be created as a "top-level" action, that is, as an action with no ancestors; a top-level action cannot be affected by an abort of any other action. Otherwise, the action is created as a "nested" action, that is, as a child (in the so-called *action tree*) of the action which created it; as described below, a nested action may be affected by an abort of one of its ancestors. Optionally, a *timeout value* may be specified; if the action has not committed by the expiration of this timeout, the action will be aborted. If no timeout value is specified, a system-defined default value is used.

As described in section 1, object operations may possibly execute in an action environment or may be invoked as an action. Only an operation or internal procedure of a *recoverable* or *autorecoverable* object may be invoked as an action; however, all operation invocations on such objects *must* be executing in an action environment. Thus, operations of a non-Clouds object or of a *nonrecoverable* Clouds object may execute in an action environment, but may not be invoked as an action. A *recoverable* or *autorecoverable* Clouds object is said to be *touched* by an action if one or more of the operations of the object are invoked within the environment of that action.

The semantics of an action invocation is as follows: the action manager operation *CreateAction* is invoked with the name of the operation to be performed as well as the list of arguments to be passed to that operation.[9] The action manager then invokes the BOA event handler on the object to which the operation belongs. Next, the action manager creates and dispatches a process in which the operation code runs. When an object is first touched by the action, the BOA handler of that object is invoked. An attempt by an operation invoked as an action to return to its caller is considered an implicit attempt to commit the action, and will cause control to transfer to the *Commit* operation of the action manager, which terminates the process and invokes the precommit event handler of each object touched by the action. (An explicit invocation of the *Commit* operation has the same effect.) If precommit of all touched objects is successful, the action manager then invokes the commit event handler of each touched object; otherwise, the objects' abort event handlers are invoked. If the action (or one of its ancestors) invokes the *Abort* operation of the action manager, the action manager terminates the process corresponding to the action and invokes the abort event handler of each object touched by that action.

It may sometimes occur that an object operation may be called either as an action invocation, or as an ordinary object operation invocation (even in an action environment). In the case that an operation is invoked normally (that is, not invoked as an action), an invocation of the action manager operation *Commit* by the operation will cause the action manager to merely return control to the point of invocation of the original operation; thus, in this case the *Commit* call is effectively a normal procedure return. On the other hand, an invocation of the *Abort* operation by an operation invoked normally will cause the parent action of the invoker of the original operation (that is, the action in the environment of which the operation is executing) to abort. Aeolus does not provide an explicit exception-handling mechanism. This function is subsumed to some extent, for operations executing within an action environment, by the action event handling mechanism. However, in the case of operations not invoked as actions, a call to the *Abort* action manager operation—as described above—provides a mechanism similar to an

8. The interface to the Clouds action manager is described in the Reference Manual.[Wilk85]

9. The exact details of the manner in which this information is provided depends on whether the operation is a local procedure or a publicly-visible operation of the object to which it belongs.

exception-handling mechanism with a single exceptional condition (say, *"error"*); the abort event is effectively propagated to the parent action, and is handled by the action event handlers of the objects which the action touched.

5. Support for Processes

The final structuring feature of the Aeolus language provides an abstraction of the *process* concept of the Clouds system. (The process is analogous to the *program* construct of Pascal or Modula-2.) The invocation of a process provides activity in the Clouds system; processes may be considered the "glue" which binds object operations, and possibly actions, to do useful work.

A process is introduced by a header which gives the name of the process, as well as clauses detailing any imports of object definitions necessary. Following any import clauses, the body (process block) of the process is specified; the statement part of this block is the entry point when the process is activated, and execution begins there after any necessary variable initializations of the process block have been performed.

6. A Programming Example

In this section, we discuss an example of systems programming using the constructs which Aeolus provides for access to the powerful features of the Clouds system for action and object programming. The text of the example object discussed here is provided in the Appendix.

Since the use of a *recoverable* data structure requires the creation of a complete copy of the data structure on the version stack for each action which modifies the data structure, the implementation of a replicated object can become inefficient as the size of the data structure increases. Fortunately, we can use semantic knowledge about the object to simulate the effect of recoverable variables at a fraction of their cost. Essentially, rather than require that the system allocate a new version of a complete data structure for each new action, we make use of per-action variables to maintain "change lists" for each action. These may be viewed as "intention lists" for operations such as insertion or deletion in a data structure. Since each action can access both its own recoverable variables and those of its parent, we may arrange to propagate the change lists of an action to its parent, usually by coding an event handler for either the *nested_precommit* or the *commit* action event. (We may also wish to arrange to clean up after ourselves in an *abort* action event handler.) The actual modification of the data structure itself is delayed until the *toplevel_precommit* action event. In a handler for this event, we must arrange to perform the changes maintained in the change lists (say, insertions and deletions) on the actual data structure; the actual data structure is maintained in permanent storage. Note that the Aeolus/Clouds system enforces the restriction that data in permanent storage may be modified only at toplevel precommit; then, the Clouds storage management system[Pitt85] assures the stability of permanent storage and the atomicity of changes to it.

The use of this technique often requires the use of linked lists or similar data structures allocated in a heap in the permanent area of per-object storage. This *permanent heap* requires special run-time support for its management, which must maintain the heap's consistency across failures.

In the example shown in the Appendix, we show a proposed design for the permanent heap manager. To maintain the consistency of the heap, this PERMHEAP object uses the same techniques which the permanent heap mechanism is designed to support, i.e., per-action variables and associated action-event handlers. In the implementation shown, the "free list" (i.e., the list of available blocks of permanent storage) is itself kept in permanent storage to ensure the resilience of the permanent heap structure. (Thus, the PERMHEAP object must actually be bootstrapped from a non-resilient version.) For the purposes of this example, we have written PERMHEAP as a recoverable object. In practice, the permanent heap support would be part of the runtime support code compiled into a recoverable object, rather than a separate object.

The PERMHEAP object maintains lists of those blocks of the heap allocated and freed by each action, in per-action variables. Also, each action which allocates a block of storage obtains a lock on the value of the address of that block. (Blocks of storage are uniquely identified by their starting addresses.) The presence of a lock on a block of storage indicates that it has been allocated by some action which has not yet committed; since changes to the actual "free list" are not made until toplevel precommit of the action allocating storage, this lock is necessary to inform other actions that the block is probably unavailable. A call to the ALLOCATE operation of PERMHEAP will return the address of a block of memory in the permanent heap area of the object; the address of the block is also added to the ALLOCATED per-action list and is locked. If a block of memory was allocated by the action which is trying to free it, a call to PERMHEAP's FREE operation will remove that block from the ALLOCATED list and release the lock on its address, effectively causing the block to never have been allocated. If, on the other hand, the block was not allocated by the invoking action, the address of the block to be disposed is merely added to the FREED per-action list; actual disposal is performed at toplevel precommit.

No special processing is required if an action allocating or freeing storage aborts, since its locks are released and no alteration to the permanent "free list" has taken place. Thus, effectively no allocations or frees have taken place. (Note that the contents of permanent storage blocks on the "free list" are considered dispensible; nevertheless, these contents may be modified only at toplevel precommit.) When a nested action enters its commit phase, its ALLOCATED and FREED per-action lists are propagated to its parent. Memory blocks on the permanent heap allocated by an action are actually removed from the "free list" when the action's toplevel ancestor (to which the nested action's per-action lists have been propagated) enters its precommit phase; also, blocks freed by the action are added to the "free list" at this time.

In Clouds, locks (as well as all portions of the object state which are not specified to be *permanent* or *recoverable*, including per-action variables) are maintained in volatile storage. Thus, information concerning uncommitted allocations and frees of an object using permanent heap support may be lost due to node failure. However, this will not cause a problem, since uncommitted actions running at a node at the time of its failure will be aborted. Recall that locks belonging to an action are released if that action is aborted; thus, the effect is the same as if the actions had been aborted in a non-failure case, i.e., as if the allocations and frees had never been performed.

Note that this implementation of the PERMHEAP object does not provide strict serializability. To see this, consider some action, *A*, which exhausts (or nearly exhausts) the permanent heap, causing other actions *B* and *C* trying to allocate permanent memory to fail. Action *A* may well be aborted itself. Actions *B* and *C* which failed because of *A* might not have failed had they been executed serially. However, such breaches of strict serializability do not affect the consistency of the permanent heap mechanism, and thus are of little concern in this context.

7. Status of the Aeolus Implementation

In the Clouds systems programming language effort, work is currently continuing in two major areas as of summer 1986: the implementation of the Aeolus compiler as well as its integration with the Clouds kernel services, and the use of the Aeolus language system as a testbed for studying the problems of programming in action-object systems.

Work on the compiler is in progress on one of the DEC VAX 11/750s of the Clouds system, under the BRL version of Berkeley UNIX™ 4.2. The Amsterdam Compiler Kit[Tane83] is being

used for code generation. The basic portion of the compiler implementation has been finished, including support for non-Clouds objects. Current work on the Aeolus implementation is being concentrated on those areas of functionality needed for interfacing with the kernel to provide support for object and action management. We anticipate that support for Clouds objects will be available in summer 1986, and that the functionality needed for a prototype implementation in Aeolus of the recently-completed action management design[Kenl86] will be available in fall 1986. (The interfaces to action management are described in the Reference Manual[Wilk85] for Aeolus.) The actual implementation of action management is being done in C, as it will be merged with the kernel code (which is also in C). Concurrently, work is progressing on the development of structured editing tools for Aeolus using the ALOEGEN structured-editor generator system developed under the GANDALF project.[Notk85]

Our plans to use the Aeolus/Clouds system as a testbed for studying programming methodologies in action-object systems have been described in a previous paper.[LeBl85] As one of these ongoing studies, we are working towards the development of a distributed object filing system for Clouds; alternate implementations of the file system will compare the efficiency of different schemes for achieving consistency and availability. Of special interest are the trade-offs available among different schemes between consistency and availability, particularly when semantic knowledge of an object may be brought to bear. This research will be described in a forthcoming dissertation.[Wilk86]

[TM] UNIX is a registered trademark of AT&T.

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, Atlanta, GA, 1982.

[Allc83] Allchin, J. E., and M. S. McKendry. "Synchronization and Recovery of Actions." *PROCEEDINGS OF THE SECOND SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING* (ACM SIGACT/SIGOPS), Montreal (August 1983).

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

[Ensl78] Enslow, P. H. "What is a "Distributed" Processing System?." *COMPUTER* (IEEE) **11**, no. 1 (January 1978): 13-21.

[Evan84] Evans, A. Jr. "A Comparison of Programming Languages: Ada, Pascal, C." In *Comparing & Assessing Programming Languages*, ed. A. Feuer and N. Gehani, 66-94. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[Kenl86] Kenley, G. G. "An Action Management System for a Distributed Operating System." M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01.)

[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 (July 1985). (Also released, in expanded form, as technical report GIT-ICS-85/03.)

[McKe85] McKendry, M. S. "Ordering Actions for Visibility." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **11**, no. 6 (June 1985). (Also released as technical report GIT-ICS-84/05.)

[Notk85] Notkin, D. "The GANDALF Project." *THE JOURNAL OF SYSTEMS AND SOFTWARE* **5**, no. 2 (May 1985).

[Pitt85] Pitts, D. V., and E. H. Spafford. "Notes on a Storage Manager for the Clouds Kernel." TECHNICAL REPORT GIT-ICS-85/02, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[Rent82] Rentsch, T. "Object Oriented Programming." *SIGPLAN NOTICES* (ACM) **17**, no. 9 (September 1982): 51-57.

[Tane83] Tanenbaum, A. S., H. van Staveren, E. G. Keizer, and J. W. Stevenson. "A Practical Tool Kit for Making Portable Compilers." *COMMUNICATIONS OF THE ACM* **26**, no. 9 (September 1983).

[Wilk85] Wilkes, C. T. "Preliminary Aeolus Reference Manual." TECHNICAL REPORT GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986.)

[Wilk86] Wilkes, C. T. "Programming Methodologies for Resilience and Availability." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (In progress.)

Appendix

The following example is discussed in section 6 of this paper. Reserved words of Aeolus are indicated by boldface.

```
definition of recoverable object permheap is
  ! Gives the publically-visible definitions provided by the PERMHEAP object.

operations

  procedure allocate ( size : unsigned ) returns address modifies
    ! Return a pointer to a block of memory of the given "size" (in words) in
    ! permanent memory.

  procedure free ( block : address ) modifies
    ! Dispose the block of memory indicated by "block".

end definition. ! permheap !
```

```
implementation of ! recoverable ! object permheap is
  ! Support for the permanent heap, using per-action variables for recovery management.

import list
  ! The definition part of the LIST object is shown here for clarity.
  !
  ! definition of local object list ( elem_type : type ) is
  !   -- This object implements a linked list abstraction. The object is parameterized
  !   -- by the element type of the list; if the element type is specified to be permanent
  !   -- by a (recoverable) importing object, then the linked list itself will be allocated
  !   -- in permanent storage (only recoverable objects may declare permanent variables).
  !   -- The list is initially empty. Mutual exclusion is provided on MODIFY operations.
  !
  ! operations
  !   procedure add ( elem : elem_type ) modifies
  !     -- Adds ELEM to the list.
  !   procedure append ( l : list ) modifies
  !     -- Appends all elements in list L to this list. Use of the object type "list"
  !     -- here with no parameters implies that list L must have the same element type
  !     -- as this list.
  !   procedure remove ( elem : elem_type ) modifies
  !     -- If ELEM is on the list, removes it.
  !   procedure find ( elem : elem_type ) returns boolean examines
  !     -- If ELEM is on the list, returns TRUE, otherwise FALSE.
  !   procedure nth ( n : unsigned,  notthere : out boolean )
  !                 returns elem_type modifies
  !     -- If the Nth element exists, returns it and sets NOTTHERE to FALSE,
  !     -- otherwise sets NOTTHERE to TRUE.
  ! end definition.
```

```
! The local declarations of the PERMHEAP object.
!
! Here, we give the names of alternate handlers for some of the action events.
! Note that no alternate handler is given for the ABORT event (see section 6).

action events
        commit is permheap_commit,
  toplevel_precommit is permheap_top_precommit


!-------------------------------------------------------------------------
! The PERM_BLOCKENTRY type is used for the maintenance in the permanent heap of the
! list of free storage blocks.  Each block is uniquely identified by its address.

type perm_blockentry is permanent new address


!-------------------------------------------------------------------------
! The list of free storage blocks.  Since the base type of this list is permanent,
! the list itself is allocated in permanent storage.
! This list may be modified only during the toplevel_precommit action event.
! The size of each entry is stored in the first word of that entry.

freelist : list( perm_blockentry ) := new list


!-------------------------------------------------------------------------
! The BLOCKENTRY type is used in the declaration of the per-action variables
! below.  Pointers to this type are allocated on the normal (not the
! permanent) heap, and may be modified outside of the toplevel_precommit
! event handler.

type blockentry is new address


!-------------------------------------------------------------------------
! The per-action variables for permanent-heap recovery management.
! We will maintain lists of memory blocks allocated and freed by each action.

per action
  allocated : list( blockentry ) := new list
  freed : list( blockentry ) := new list
end per action


!-------------------------------------------------------------------------
! When an action allocates a block of permanent storage, it must obtain a lock on that
! block until it commits to prevent other actions from attempting to allocate that block.
! Rather than associate a lock with the actual storage block, we lock the block's address
! (of type BLOCKENTRY).  Recall that locks obtained by an action are propagated to its
! parent upon nested commit, and released upon abort or toplevel commit.

entry_lock : lock ( busy : [] ) domain is blockentry
```

```
procedure first_fit ( size : unsigned ) returns blockentry is
  ! A private operation of the PERMHEAP object.  Given a size in words, FIRST_FIT finds
  ! the first entry on the FREELIST for a block of storage of size at least as large as
  ! SIZE and returns a pointer to that entry.  (For the purposes of this example, we
  ! will assume that such a block exists.)  Of course, another strategy could also be
  ! used here (such as best fit, or fragmentation and compaction).  We'll assume that
  ! repeated invocations of FIRST_FIT by the same action return different addresses.
  begin
    ! The details of this operation are omitted here.  Even if an appropriate block of
    ! storage is found on the FREELIST, FIRST_FIT must also test the ENTRY_LOCK to check
    ! whether this block has not already been allocated by some as yet uncommitted action.
  end procedure ! first_fit !

!
! ALLOCATE and FREE are public operations of the PERMHEAP object.
!

procedure allocate (! size : unsigned !) ! returns address ! is
  ! Return the address of a block of memory of the given SIZE in permanent storage.
  ! Since the block is from the FREELIST, its former contents are expendable.
  ! The Set_Lock operation used here is non-blocking, i.e., it returns immediately with
  ! value FALSE if the requested lock is not available.

  entry : blockentry

  begin
    loop   ! keep going until we find an available block
      entry := first_fit( size )
      if Set_Lock( entry_lock, busy, entry ) then
        Self.allocated @ add( entry )  ! add the entry to the ALLOCATED list for this action
        return address( entry )
      end if
    end loop
  end procedure ! allocate !


procedure free (! block : address !) is
  ! Add a BLOCK of memory to the FREED list for freeing during toplevel precommit.

  entry : blockentry
  notthere : boolean
  i : unsigned := 1

  begin
    ! First, scan the ALLOCATED list to see if BLOCK was allocated by the current action
    loop
      entry := Self.allocated @ nth( i, notthere )
      if notthere then
        exit .
      elsif entry = blockentry( block ) then  ! Yes,
        Self.allocated @ remove( entry )      ! so remove it from ALLOCATED list
        ReleaseLock( entry_lock, busy, entry )
        return .                              ! we're done
      end if
      i += 1
    end loop

    ! If we get here, BLOCK wasn't allocated by the current action, so put it on the FREED list
    Self.freed @ add( entry )
  end procedure ! free !
```

```
|
| The following are the alternate action event handlers for this object.
|

procedure permheap_commit () is
  | The alternate handler for the COMMIT action event.  We'll propagate the items on
  | the ALLOCATED and FREED lists of this action to the corresponding lists of its parent action.

  aID : action_ID
  status : action_status
  level : action_level

  begin
    aID := ActionManager @ Tell_ID( status, level )   | see if we're in a nested action
    if level = nested_action then
      Parent.allocated @ append( Self.allocated )
      Parent.freed @ append( Self.freed )
    end if
  end procedure | permheap_commit |


procedure permheap_top_precommit () is
  | The alternate handler for the TOPLEVEL_PRECOMMIT action event.  We'll traverse the FREED
  | list, adding each entry there to the actual FREELIST in permanent storage; then, we'll
  | traverse the ALLOCATED list, removing each entry there from the FREELIST.

  entry : blockentry
  notthere : boolean
  i : unsigned := 1

  begin
    | Add each entry on the FREED list to the FREELIST in permanent storage
    loop
      entry := Self.freed @ nth( i, notthere )
      if notthere then
        exit .
      end if
      | Convert the entry to the permanent type before adding to FREELIST.
      freelist @ add( perm_blockentry( entry ) )
    end loop

    | Remove each entry on the ALLOCATED list from the FREELIST; the locks on these
    | entries will be released automatically.
    loop
      entry := Self.allocated @ nth( i, notthere )
      if notthere then
        exit .
      end if
      freelist @ remove( perm_blockentry( entry ) )
    end loop
  end procedure | permheap_top_precommit |


inithandler is   | handler for the INIT (initialization) object event
  begin
    | Perform initialization (not shown) of FREELIST to indicate that all
    | of the permanent heap is available.
  end inithandler
```

```
reinithandler is  ! handler for the REINIT (reinitialization) object event
  begin
    NULL  ! This handler would by default be the same as the INIT handler
  end reinithandler

!
! The DELETE object event handler for this object is by default NULL.
!

end implementation. ! permheap !
```

QUARTERLY PROGRESS REPORT
RESEARCH ON RELIABLE DISTRIBUTED
COMPUTING
CONTRACT #MDA 904-86-C-5002
REPORTING PERIOD: 1 OCT 85 - 31 DEC 85

1. Project Status

During the first quarter of this project, work has continued on each of the two tasks carried over from our previous project and we have begun work on the new task. These efforts are closely related to other work in progress within the Clouds Project, our major research effort in the area of reliable distributed computing.

Under the Language Support for Robust Distributed Programs task, work continues in two major areas: the implementation of the Aeolus compiler as well as its integration with the Clouds kernel services, and the use of the Aeolus language system as a testbed for studying the problems of programming in action-object systems.

Under the Storage Management for an Action-Based Operating System task, the focus of our work has been on implementation, testing and integration with the virtual memory management mechanisms of the Clouds kernel.

Under the Operating System Support for Reliable Distributed Computing Task, our efforts have been directed toward specification and functional design of the operating system services which will be implemented on top of the object and action management mechanisms provided by the Clouds kernel.

The work on the tasks of this project is proceeding on schedule. Working in combination with other efforts in progress within the Clouds project, we are now in the process of debugging our initial prototype system.

2. Language Support for Robust distributed Programs

In the Clouds systems programming language effort, work continues in two major areas: the implementation of the Aeolus compiler as well as its integration with the Clouds kernel services, and the use of the Aeolus language system as a testbed for studying the problems of programming in action-object systems.

The major changes made to the Aeolus design over the summer quarter were described in our last report. Work on the language implementation is proceeding well, with many of the changes to the language made over the summer now incorporated into the compiler. Current work is being concentrated on those areas of functionality needed for interfacing with the kernel. In addition, we are working towards the functionality required for an implementation in Aeolus of the recently-completed action management design.

The design of the interfaces of the runtime system with the Clouds action and object managers is essentially complete. As was mentioned in our last report, members of the Aeolus group have been assisting members of the kernel group in the design of these interfaces as well as in strategies for efficient action management. The detailed designs of the action and object managers are now complete, and are described in [Kenl86]. Our basic designs for the Aeolus/Clouds interfaces (from the kernel side) are also described in this document; the Aeolus interfaces with these kernel routines are being codified as appendices to [Wilk85b]. The action management routines themselves are now being programmed from Kenley's detailed pseudo-code; implementations are being done both in the C language and in Aeolus. The Aeolus implementation is being done principally to pinpoint weaknesses which the language may have as a systems programming language, before the design of the language is finalized; since the rest of the kernel is written in C, and since the pseudo-code design is based on C, it was felt that the first production implementation of the action management routines should be done in C. Because Aeolus was designed to allow easy interfacing with other languages (through use of the *pseudo-local object* construct [Wilk85b]), addition of action management support to Aeolus will be relatively trivial once the kernel routines are available; most of the interaction with action management will take place through kernel calls, implemented as operations on an action management pseudo-object. Other support for actions required from the Aeolus compiler includes the identification of recoverable areas of storage, permanent and per-action variables,

and alternate action-event handlers; information about such constructs must be placed by the compiler in the header of the compiled object for use by action management at runtime.

Our plans to use the Aeolus/Clouds system as a testbed for studying programming methodologies in action-object systems have been described in previous reports as well as in [Wilk85a]. As one of these ongoing studies, we are working towards the development of a distributed file system for Clouds; alternate implementations of the file system will compare the efficiency of different schemes for achieving consistency and availability. Of special interest are the trade-offs available among different schemes between consistency and availability, particularly when semantic knowledge of an object may be brought to bear. As an example of such a trade-off, there may be applicationssuch as air-traffic controlin which violation of the consistency requirement among replicated objects may be tolerable for short periods (for instance, during a network partition) in exchange for increased availability; such reduced quality of service would be preferable to no service at all in these types of applications. Our work on the distributed file system study is concentrating on these issues in relation to two schemes for replicated data management: the quorum method, which assigns a weighted number of votes to each replicant of a data object, and requires that a quorum of these votes be gathered before a read or write operation may take place; and the master/slave method of McKendry (as described in [Wilk85a]), which uses "probes" to determine the availability of the master replicant to the slave replicant (and vice-versa) before operations are executed. The quorum method emphasizes consistency over availability, in that consistency among the replicants is guaranteed by the requirement that a quorum of objects be gathered before an operation may take place; however, an operation may not take place in a partition in which a quorum of objects is not available, even if one (or more) of the replicants is present in the partition. An algorithm for using the quorum method for distributed directories has been developed by Daniels and Spector [Dani83]; we will be modifying this algorithm for use in our comparative study. The master/slave method, on the other hand, maintains consistency among replicants in the absence of failures by requiring that any operation invoked on a slave be relayed to the master object, which in turn invokes that operation on all slaves. (Thus, this scheme partially resembles the so-called "primary copy" methods.) However, when a failure (for instance, network partition) occurs, any slave replicant may detect its isolation from the master by use of itself the master in that partition. Thus, service may continue in a partition containing at least one replicant, at the price of possible inconsistency among replicants in different partitions. These inconsistencies must be resolved when the failure (partition) is resolved; methods for doing this are demonstrated in the examples in [Wilk85a]. Thus, the master/slave method emphasizes availability over consistency, at least during failures. In our studies, we are examining combining the quorum method with the master/slave method to improve efficiency during the non-failure case.

3. Storage Management for an Action-Based Operating System

The storage manager is almost completely implemented and is currently undergoing testing. There currently exists a working driver for the RL02 removable pack disk, which is being used for interim testing while development of a driver for the major storage device (the RA81 fixed medium disk) of the Clouds kernel is completed. The completion of the driver for the RA81 is expected by April 1986. The recent discovery of some technical information concerning the functioning of the disk and its relationship to the UDA50 controller has caused some modifications in the design for the driver.

The partition level software for the Clouds kernel is on-line and working. Some changes in the partition interface and design were made during this period as a result of the refinement of the segment system design. We now can create complete Clouds partitions and perform several functions on the partitions. The functions include a complete set of partition directory operations (add an entry to the directory, remove an entry to the directory, and find the location of an entry in the directory), a set of directory dump operations (for collecting lists of segment

sysnames that reside on a partition, and operations for allocating and deallocating partition storage. Two components remain to be implemented for the partition level software: quick look-up mechanism (called the Maybe Table), and an activation routine, which will bring certain partition structures into memory and create the lock and semaphores necessary to manage the partition. These components will be added by the end of January.

Coding of the segment level software is continuing, with only a few major routines remaining uncoded. These routines form the object data recovery mechanism of the Clouds kernel. The final of these routines depend on the decisions reached for the rest of the segment mechanism. Now that these decisions have been made, implementation of the recovery routines can be completed.

The basic segment mechanism provides a means for performing input/output requests at the segment level. All input/output requests at the kernel level occur as part of virtual memory management. That is, if object data is needed, it is brought in to virtual memory as part of a page fault. Thus, the design of the segment system had to be integrated into the object and action management subsystems as part of the virtual memory management subsystem.

The following portions of the segment system are coded and are being tested:

A segment activation mechanism: When handling an object operation invocation, object management initiates a search to find the object and to insure the object is active. Activating the object involves creating an active segment descriptor for the disk image of the object and bringing the segment header into memory. A mapping for the segment is created. If the segment is already active, the current mapping may be modified.

Segment create and destroy operations: Disk images of objects may be placed on and removed from the device. The operations support the creation and destruction of recoverable segments under the auspice of an action.

Segment read and write operations: Data may be transferred to and from the disk, giving a segment offset as the source or destination. Memory locations used in the transfer are physical addresses. The operations use the current mapping to determine where the segment offset reside on disk.

A page mechanism on top of the segment i/o: Pages faults are handled in tandem by object management and storage management. Initially, object management determines which object and where in the object, the fault occurred. The storage manager allocates a physical page and does a segment read to fill the page.

Testing of the basic segment software will be completed by mid-January. The remaining segment operations will be completed by the end of January.

4. Operating System Support for Reliable Distributed Computing

4.1 Introduction

The Clouds Project at Georgia Tech includes 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 maintenance 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 are being designed into the Clouds operating system. Most of the functions have been designed into the kernel of the system. 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 hard-wired 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.

4.2 Progress Report

The kernel has been designed and implemented to a large degree. The process dispatcher, the virtual memory, object invocation procedures, and some storage and communication software has been implemented and tested. We currently do not have software to build Clouds objects, and thus have not been able to test the invocation in a multiprocess environment. The kernel has been tested in a stand alone system with hand-coded objects.

The most important communications package in Clouds, the ethernet driver has been implemented and tested. The driver in based on a very general design and has the ability to support a host of protocols that can be hooked to it. It currently talks to the Clouds machines as well as the machines running Unix 4.2bsd (tm).

The storage management subsystem is partially implemented. Disk drivers for implementing the file system (for object storage) is in the test phase. The advanced virtual memory features needed by Clouds (partitions, object mapping, segment handling) is being coded and tested. Implementation of virtual disks using the ethernet (for intersite paging) is underway as the ethernet driver is now available.

The next phase will integrate the results of the compiler building with the kernel to allow building of services and user programs as objects and running them on using multiple processes, and multiple sites.

The action management is an advanced kernel subsystem that ensures the atomicity of the distributed actions of the Clouds system. The action management is responsible for creation, deletion, proper or improper termination of actions, commitment, and failure containment. The design of the action management subsystem is as far complete as can be achieved theoretically without availability of implementation experience. The implementation will begin as soon as the base kernel is fully tested.

5. References

[Dani83] Daniels, D., and A. Z. Spector, "An Algorithm for Replicated Directories," *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Montreal, August 1983

[Kenl86] Kenley, G. G., "An Action Management System for a Decentralized Operating System," M.S. Thesis, School of Information and Computer Science, Georgia Institute of Technology, January 1986 (also available as Technical Report GIT-ICS-86/01)

[Wilk85a] Wilkes, C. T., "Programming Methodologies for Resilience and Availability," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology, January 1985

[Wilk85b] Wilkes, C. T., "Preliminary Aeolus Reference Manual," Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, July 1985

FINAL REPORT
GIT Project No. G36-636

FAULT TOLERANT DISTRIBUTED COMPUTING

Richard J. LeBlanc

Prepared for

Maryland Procurement Office
Ft. George G. Meade, MD 20755

Under

Contract Number MDA904-86-C-5002

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

1986

1. Project Accomplishments

During the course of this contract, substantial progress has been made on each of the three project tasks. These efforts are closely related to other work in progress within the Clouds Project, our major research effort in the area of reliable distributed computing.

Under the Language Support for Robust Distributed Programs task, the work has proceeded in two major areas: the integration of the Aeolus compiler with the Clouds kernel services and the use of the Aeolus language system as a testbed for studying the problems of programming in action-object systems. A discussion of the design of Aeolus was provided in an appendix to the third quarterly progress report delivered under this contract.

Under the Storage Management for an Action-Based Operating System task, the focus of our work has been on the design and implementation of the kernel storage manager and on implementation of a device driver to enable us to use our large disk drives on machines running the Clouds kernel. The design of the Clouds storage manager is described in a docment attached as Appendix A to this report.

Under the Operating System Support for Reliable Distributed Computing task, our efforts have been directed toward specification and functional design of the operating system services which will be implemented on top of the object and action management mechanisms provided by the Clouds kernel. Our short term goal has been to obtain a working, robust kernel to provide a basis for the implementation of these designs. Recently, that goal has been achieved through the integration of a number of separate projects. A description of he Clouds distributed operating system with a comparison to other related efforts is attached as Appendix B.

2. Language Support for Robust Distributed Programs

Work under this task has included efforts in two major areas: the design and implementation of the Aeolus language, and the use of Aeolus for the study of programming methodologies for action/object systems.

2.1 Language Design and Implementation

As mentioned in the last report, the design of the Aeolus language is now "frozen" (we hope permanently), and the implementation effort is proceeding. Our goal of providing support for Clouds objects in the compiler is now nearly achieved. This support is realized in two different areas. The first is run-time support for Clouds object operation invocations. This involves formatting arguments suitably for remote procedure call (since the target object may not reside on the machine where the invocation is produced), and handling such things as copying return values and "out" parameter values upon return from the invocation. Code for this has been produced, and the compiler generates all the necessary data structures and invocations.

The second area is the creation of TypeManager objects. When a Clouds object is compiled, a Unix "a.out"-style load file is created; the Unix header is then stripped from this file to yield a description for the object in the format expected by Clouds. A TypeManager, once created under a system running the Clouds kernel, requests this object description file from the Unix system and stores the description as the TypeManager's object data. Subsequently, when the "create" operation is invoked on the TypeManager, the object description is used to create an instance of that object type. To create TypeManagers, we will "hard-wire" a TypeManager for TypeManagers into the kernel. Work on this original TypeManager is proceeding, as well as on the supporting code which brings the objects code and data from the Unix system. We are currently working with members of the kernel group to integrate these features into the Clouds kernel.

2.2 Programming Methodologies for Action/Object Systems

During the final quarter, our work on achieving availability of resources in the Clouds system has continued with study of the work of Herlihy, presented in his dissertation,

"Replication Methods for Abstract Data Types,"[Herl84] and with correspondences between Herlihy's techniques and the synchronization mechanisms used in Clouds, which should allow us to apply Herlihy's methods to our problem of generating replicated objects.

Herlihy's work concerns the extension of quorum intersection methods to take advantage of the semantic properties of abstract data types. Previously, work on quorum methods—mostly in the database area—has been limited to a simple read/write model of operations. Herlihy's extensions allow the selection of optimal quorums for each operation of an abstract data type based on the semantics of that operation and its interaction with the other operations of the data type.

Herlihy's method is based on the analysis of the algebraic structure of abstract data types. This entails the construction of a "quorum intersection graph," each node of which represents an operation of the data type, and each edge of which is directed from the node representing an operation *O1* to the node representing operation *O2*, where each quorum of *O2* is required to intersect each quorum of *O1*. From the quorum intersection graph, optimal quorums for each operation may be calculated, given the number of replicas of the data, and the desired availability of each operation in relation to the other operations of the data type.

Herlihy shows that his method can enhance the concurrency of operations on replicated data over that obtained from a read/write model of operations. He also claims advantages for his methods in the support of on-the-fly reconfiguration of replicated data, and in enhancing the availability of the data in the presence of network partitions.

There appears to be a close relationship between Herlihy's quorum intersection graphs and the lock compatibility matrices used in Aeolus and the Clouds system; a graph constructed from the lock compatibility matrices for an Aeolus/Clouds object is either the complement of the quorum intersection graph for the operations of that object, or a subset of the complement. This is not really surprising, since the specification of our lock compatibilities is based on the programmer's analysis of the compatibilities between the object operations, while Herlihy's quorum intersection graph may be viewed as being based on an analysis of the *incompatibilities* between operations.

Thus, we should be able to apply Herlihy's techniques to our problem of generating replicated objects given an unreplicated object version and a specification of the desired replication properties. This entails extending the notion of the Aeolus/Clouds lock to include the "distributed" lock; that is, the state of the lock is shared logically among all replicas of an object. This will, of course, require the transmission of lock state information among all replicas. However, the concurrency properties of the unreplicated version of the object will be retained by the replicated version generated from it. This is especially significant given the power of the Aeolus/Clouds lock mechanism in expressing arbitrary compatibilities and in allowing the expression of synchronization at arbitrary levels of granularity.

We are currently investigating these possibilities in the course of the design of the object filing system (OFS) for Clouds. The replication scheme which we are currently considering in support of availability will require heavy interaction between the manager for a replicated object, the job scheduler, and the OFS. The OFS should:

- be resilient and highly available (through replication);
- provide a mapping from object names (strings) to Clouds object capabilities;
- impose some familiar structure (e.g., a Unix-like hierarchical structure) on the flat, global system name space provided by the Clouds object manager;
- provide efficient forms for the most common types of I/O (such as text I/O) without the necessity of the context switches which would be required if such I/O were modelled with Clouds objects.

In the OFS, an object name may represent a *group* of objects (the set of replicas of a replicated object), rather than a single instance. We intend that this mechanism should be, in general, transparent to the user (although special-purpose applications, such as DBMSs, may require that, in addition, finer control of replication be available than that provided by a general mechanism).

We are currently considering two different capability-based naming schemes which may be used by the OFS in support of *state cloning*, as described in a previous report. The first scheme requires minimal changes to the kernel, but relies on facets of the Clouds object lookup mechanism which may not be applicable to other systems. In Clouds, the search for an object begins locally (that is, on the node which invoked the search), and—if the object is not found locally—proceeds to a broadcast search. If the internal objects belonging to a replica are constrained to reside on the same node as their parent object, then the local search will locate the local instance of the internal object. (We do not consider this constraint to be onerous, since the internal objects of each replica need to be highly available to that replica in any case, and thus should logically reside on the same node as the parent replica. This constraint may be enforced by the OFS.) Thus, each replica of an object (each of which resides on a separate node) may maintain its set of internal objects using the same capabilities as each other replica. Although we will thus have multiple instances (on separate nodes) of internal objects referenced by the same capability, there should be no problems caused by this, since—by the definition of internal object—only the parent object or its internal objects may possess the capability to an internal object, and the object search will always locate the correct (local) instance. Thus, state cloning may be used to copy the state of a replica to the other replicas without causing the problems with respect to internal objects mentioned in the previous report (concerning references to internal objects contained in the replica's state), since under this scheme all replicas may use the same capabilities for referencing internal objects. This scheme is an extension of a facility already supported by the Clouds kernel for cloning read-only objects such as code. We call this scheme *vertical replication*, since it maintains the grouping of internal objects with their parent object.

The other naming scheme makes fewer assumptions about the lookup mechanism than vertical replication, but requires more kernel modifications. In the second scheme, each instance of the replicas' internal objects is again named by the same capability, at least as far as the user is concerned; however, the kernel maintains several additional bits associated with each capability identifying a unique instance. (These additional bits may be derived from, for instance, the birth node of the instance.) When a (parent) replica invokes an operation on an internal object, the kernel selects one of the replicas of the internal object according to some scheme (e.g., iteration through the list of nodes containing such objects until an available copy is located). Thus, a set of replicas of internal objects is maintained in a "pool" for access by all parent replicas. Again, each parent appears to use the same (user) capability to reference a given internal object, so the problems of state cloning disappear. Since this scheme maintains a logical grouping of the copies of an internal object, rather than grouping internal objects with their parent object, we refer to the scheme as *horizontal replication*.

Our initial design of the OFS is concerned with an unreplicated version; when completed, the design will be extended to a replicated version by use of the "distributed lock" mechanism and an analysis of the desired replication properties of the OFS.

3. Storage Management for an Action-Based Operating System

During the final quarter of the contract, final testing and documentation of the storage management system for the Clouds kernel was completed. A copy of the documentation can be found in Appendix A. During the previous quarter, a dissertation[Pitt86] based on the storage manager development effort was completed and defended by David Pitts. The dissertation describes the three major subsystems of the storage manager: the device system, the partition system, and the segment system. For each subsystem, the structures and operations that comprise the subsystem are defined. The dissertation describes the basic services provided by

the storage manager: object memory support, recovery management, and directory management. The dissertation highlights the integration of virtual memory management with object memory support and recovery management. One of the claims of the dissertation is that this integration provides a efficient system.

The dissertation describes three algorithms that support the two-phase commit of actions in a Clouds system. It is shown how these algorithms support action management and also crash recovery. A chapter in the dissertation is devoted proving the correctness of these algorithms, based on the assumptions made for the Clouds system.

4. Operating System Support for Reliable Distributed Computing

The Clouds Operating System kernel provides the systems support for objects and actions. Two primary attributes supported at the kernel level are persistent object memory and atomic actions.

The Clouds object memory consists of a virtual address space per object. This virtual space is also persistent or permanent. That is, any modifications to the virtual state of the object remain forever (unless explicitly deleted). Thus, the objects are longer lived than the processes that create, access, and modify them.

The atomic action paradigm allow processes (executing on behalf of the actions) to update the objects atomically. That is, either all objects touched by the action get updated, or none of the objects are updated.

The object memory in Clouds is supported by the object management system, which supports distributed object invocations and demand paged object virtual memory. Two recent Ph.D. graduates have completed most of the kernel support for the reliable object memory.[Pitt86, Spaf86] The details are as follows.

The object management system has been tested to handle object invocations, both local and remote. This uses the communication system which uses Ethernet routines to communicate with other Clouds sites as well as Unix machines. The object management system uses a search and invoke strategy for locating objects in a uniform, location independent manner, that works even if some of the sites are non functional. The global searches occur efficiently, as they use a hash table based decision function based on the Bloom filter (we call this the "Maybe Table").

The storage management system provide the functions of basic virtual memory, object memory, shadowing, flushing, and commit. It also provides directory services for object lookup (using capabilities), and interfaces with the Maybe Table handling routines. This system has also been implemented and tested.

The communication system has been developed to be compatible with the Unix conventions (Berkeley 4.2 bsd and 4.3 bsd). This provides us with the ability to access the Clouds system from Unix, and to allow the use of Unix system calls from Clouds applications. Work in this aspect is under progress. As of present, we generate Clouds objects on Unix and transmit them to Clouds. We also have the capability to create object instances from Unix machines and perform object invocations to these objects from Unix programs. We also have integrated the object management and storage management systems to work together, allowing us to use a usable integrated kernel that is capable of distributed object handling.

All of these subsystems have been integrated and tested to achieve a working Clouds kernel. We now have the capability to run distributed programs on our Clouds testbed. This capability enables us to begin implementing some of our designs for operating system services, such as those described below.

Work on the action management system[Kenl86] is underway. This system uses the reliable storage management system to provide atomic nested actions. Atomic nested actions are the first step towards fault tolerance.

On the design side, research has resulted in the design of several subsystems, notably a monitoring system and a distributed database system. The monitoring system fits into the Clouds reconfiguration strategies and uses a new mechanism called probes to monitor the health of the distributed system. The database is a conventional distributed database in a novel implementation environment. The object and action support provided by Clouds lend themselves effectively to implement a database system (modified to the object based structure), and provide concurrency control and recovery mechanisms in an environment that is simple to use.

The monitoring system design makes use of probes. Probes are high priority messages in Clouds that can be sent to processes, actions, or objects. If sent to processes or actions, a probe causes a jump to a probe handler (similar to software signals). The probe handler generates a reply to the sender of the probe containing status information about the process or the action. The object probes work along similar lines, except that the probe causes the invocation of the probe handler in the object. The monitoring system uses probes to monitor the health of critical system components. The monitors are replicated at each site and they keep status information in fully replicated databases. Each monitoring process has a backup monitor that monitors it from another site. Using this scheme, we can keep good records of the global system state, and can handle failures by tying into the reconfiguration system and restarting failed actions at healthy sites. The design is reported in detail in [Dasg86].

The relational database system is an application environment under design to function in the object oriented environment supported by Clouds. Conventional database design suffers from two deficiencies. The data models proposed by database designers do not match the components supported by the operating system, and thus the implementors have to contrive mechanisms to support the database. Also, the services (concurrency control, recovery) needed by databases are often not available and have to be built on top of a conventional operating system, giving rise to inefficient and often incorrect implementations. The object oriented approach provided by Clouds allows relational databases to be encapsulated in objects, and the implementation matches both the environment as well as the data model, giving rise to better performance, clean elegant systems interfaces, and a modular implementation. The synchronization and recovery support provided by Clouds also effectively provides database services, giving rise to database management functions which are easier to implement. Fine granularities of locking structures can be attained by relation fragmentation, that gives rise to more efficient access strategies. But as the objects hide the fragmentation details, the interfaces are just as clean and transparent. Further details can be found in [Dasg86a].

Research toward the design of fault tolerant systems management for Clouds has led to the design of an object replication system that is capable of providing non-stop systems services and processing capabilities. This system uses two basic mechanisms and a novel processing scheme. Replicated objects are named by a modified version of the present capability mechanism which allows Clouds to name a replicated object without referring to any particular replica. The invocation scheme for replicated objects causes the invocation of any one replica. We use these basic mechanisms to set up multiple processing threads which produce the effect of only one execution thread, but with far superior reliability characteristics. Unlike most systems which provide replicated data for reliability purposes, our scheme allows processing to continue even in case of transient failures which abort parts of the computation, thus providing non-stop processing capabilities.

REFERENCES

[Dasg86] Dasgupta, P. "A Probe-Based Fault Tolerant Scheme for the Clouds Operating System." TECHNICAL REPORT GIT-ICS-86/05, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[Dasg86a] Dasgupta, P., and M. Morsi. "An Object-Based Distributed Database System

Supported on the Clouds Operating System." TECHNICAL REPORT GIT-ICS-86/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[Herl84] Herlihy, M. "Replication Methods for Abstract Data Types." PH.D. DISS., Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, May 1984. (Also released as Technical Report MIT/LCS/TR-319.)

[Kenl86] Kenley, G. G. "An Action Management System for a Distributed Operating System." M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01.)

[Pitt86] Pitts, D. V. "Storage Management for a Reliable Decentralized Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as Technical Report GIT-ICS-86/21.)

[Spaf86] Spafford, E. H. "Kernel Structures for a Distributed Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/16.)

Storage Management in the Clouds Kernel

David V. Pitts

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280

ABSTRACT

The Clouds storage management system supports the object and action primitives provided by the Clouds kernel. Particularly, the storage manager is concerned with mapping object data into virtual memory and providing action and crash recovery for recoverable objects. This document presents some of the technical details in the testbed implementation of the storage manager. The storage manager's general strategy is presented first. Then, the major routines which implement the segment, partition, and device subcomponents of the storage manager are described. The document includes a description of the functional relationship of these routines. The interface between the Clouds kernel and the storage manager is described also.

November 12, 1986

1. AN OVERVIEW OF STORAGE MANAGEMENT

This section presents an overview of the storage manager. The major structures used by the storage manager and the major services provided by the storage manager are discussed. This section is concerned primarily with the functionality and the interrelationships of the components. The implemenation details of the components is described in the following sections of this document.

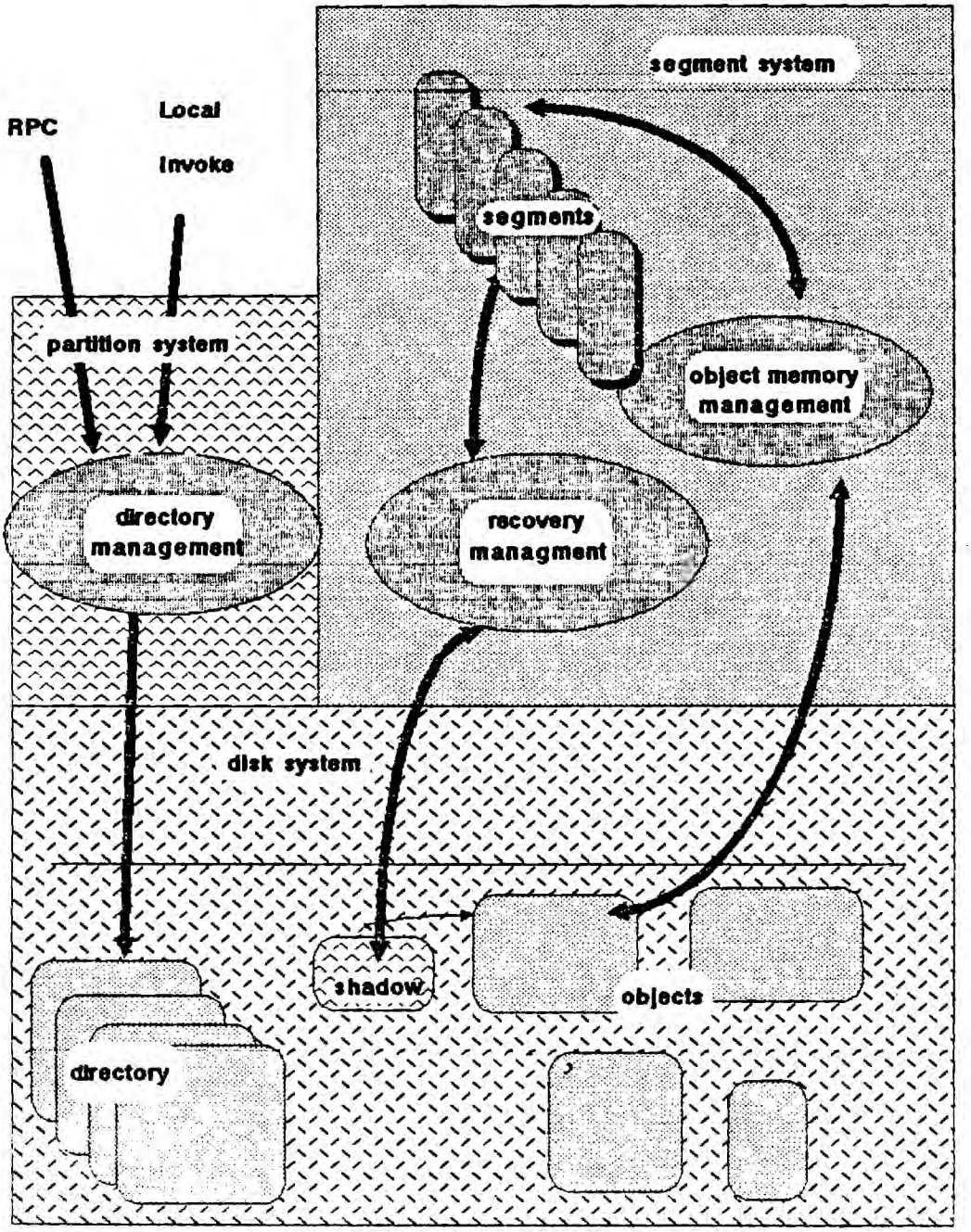


Figure 1. Storage management services and responsibilities

Figure 1 gives an overall view of the types of services provided by the storage manager. These services fall into three categories: object memory management, recovery management, and directory management. As implied in the diagram, these facilities are provided by three storage management subsystems: the device system, the partition system, and the segment system. Each of the subsystems is modelled as a collection of objects. The device objects making up the device subsystem represent the lowest level in a hierarchy formed by these three object classes. These objects provide the kernel with a direct interface to the hardware (the secondary storage devices). These objects are very device dependent; the device object type is a class of objects, one for each type of secondary storage device provided to the system. They perform the functions that device drivers handle in conventional systems, in addition to providing recovery and virtual memory support.

Partition objects enable a Clouds system to divide physical storage devices (media) into logical units for administrative/policy purposes. An important function of the partition system is the management of directories, which indicate where the permanent state of objects reside. This is important to the object invocation mechanism. Storage allocation is also done at the partition level. Typical uses for partitions might be to divide a physical device into a paging partition, an object storage partition, and a kernel storage partition. It might also be useful for a Clouds system to provide separate partitions for recoverable and non-recoverable objects.

At the highest level of the storage manager hierarchy is the segment system. The segment system manages segment objects. The segment system of the storage manager will provide the storage manager's main interface to the rest of the kernel. Paging, mapping, and other manipulations of secondary storage are performed by calls to segment objects. There are four classes of segments supported by the storage manager. There are uninterpreted segments called *datafiles*; the information in these files is simply a stream of bytes as far as the storage manager is concerned. The other three segment classes represent object data and code. *Non-recoverable* segments are simply segments which cannot be used to perform recoverable computations. *Auto-recoverable* segments can be used effectively by actions when recovery is required. The entire data state of auto-recoverable objects is recoverable on site failures. *Recoverable* segments support the customized recovery of objects. The programmer of the object may provide alternate definitions for the precommit, commit, and abort routines, and specifies exactly what data maintained by the object is recoverable and what is not.

The segment object type provides the interface between the permanent representation of the data maintained for reliability and the volatile representation used to access the data.

1.1 Device Objects

The device object provides the mechanism for sending requests to the device. The device object class provides a uniform interface to secondary storage devices. The type of devices initially considered are disks, but other device types may also be considered. The Clouds secondary storage model is very simple: it is simply a sequence of secondary storage *blocks*, which are labeled by a per-device offset, called a *device block number* (DBN). Translation of the DBN to the corresponding physical address on the device (for instance, a cylinder/track/sector specification for a disk device) is performed by the device object. Generally, the block size on secondary storage and the virtual memory page size are related; i.e., one is a multiple of the other. In the case of the prototype implementation of the kernel, the block size and page size are equal.

1.1.1 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 not important 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. One of the goals of the Clouds system is to allow machines (which support a Clouds kernel and operating system) to dynamically join the multicomputer on-the-fly simply by making them part of the physical network. Similarly, the Clouds system allows objects, partitions, media, and even devices to migrate through the system. When one site in the Clouds system fails, it is possible to take a

disk pack to another system or make the disk device accessible through an alternate port. Therefore, 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 the storage manager hides this separation by providing 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. A header is written on the medium which contains the device and medium sysnames, the allocation table is cleared so that partitions may be created, and the in-memory structures that the storage manager requires to activate a device are created.

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. This design does not attempt to address security issues; the intent 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.

1.1.2 Device Object Structures Each storage medium contains basic information about the medium and the device using it as part of the *medium header*. 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 medium header also contains the index table, which describes the partitions that exist on this device. This includes information such as location, extent, and type.

In addition, the device objects maintains a structure in memory called a *flush table*. The flush table allows a device to associate an action sysname with a set of requests to a device. This supports the commit operation performed on recoverable objects, which is discussed later in this section.

The device object uses one other structure, the *active device table* (ADT). Each entry in the ADT is an *active device descriptor* (ADD). The ADT is not a part of the device object proper, but is actually the mechanism for managing the various instances of the device objects. Each ADD contains the volatile state of a device object which is active at the local site. Included in the ADD are device and medium sysnames, status variables for the device, device registers, and entry points into the operations for the device object. By necessity, the code for each device object is heavily dependent on the particular device for which it is written. The ADT provides a means not only to identify the active devices on a site, but also provides a uniform interface to the more hardware independent portions of the storage manager.

Some of the devices that are to be used for secondary storage on the Clouds system may be *dual-ported*; i.e., they may be physically connected to two sites. At any given time, however, the device is logically connected to only one of the sites. All requests to the device for I/O transfer are handled by the logically connected site. The device may be switched between the two sites via the panel switches on the device or via software. This mechanism provides a convenient way of migrating a device to another site because of a failure at the site controlling the device. Logically, it does not matter from which site the device is available because it is referenced by its sysname. Similarly, the objects and partitions residing on the disk can also be accessed independently of their location. To perform the transfer of control after a site failure,

the device must be switched to the alternate site and then mounted on the new site. Because the previous site failed, the objects and partitions residing on the device may be in an inconsistent state, so the activation of the device at the new site may need to complete some of the action processing that had been begun on the old site. Once again, this processing can be performed in a location independent manner.

There is also the possibility of sharing the device between two running sites if the device supports software transfer of control. There are many coordination and policy issues to be address in this situation. There must be some protocol for performing the transfer of control and some mechanism for synchronizing access by the two sites to the same storage blocks. These issues are beyond the scope of this dissertation.

1.1.3 Functionality To simplify interactions with the device-level operations, each device object implementation provides the same set of operations, each of which provides the same interface to the higher level objects. The operations are of three general types. The first group of operations deals with device management and controls the availability of devices. The device management set includes an operation for formatting device media; an operation for binding a device-medium pair, making the device available to the system; and an operation which breaks bindings, making the device unavailable. The operations control availability of the devices by the creation and initialization of ADDs. Availability of devices at a site is dynamic. Devices may be stopped for maintenance or moved to a new system for availability of the resources on the device. The above operations provide the mechanism for the reconfiguration of secondary storage at sites.

Allocation of secondary storage is done primarily at the partition level, where space allocated is to be used for segment data. However, the device objects also have some limited allocation duties. Device storage management is intended to provide storage for newly created partitions. Information about the newly created partitions is stored in the medium header. Operations also exist for removing partition information from the header and providing information about the currently existing partitions on a device. The latter operation is useful for activating partitions on system restart. Storage allocation at the device is a rare operation, occurring only when partitions are created or destroyed.

Three operations are concerned with data transfers. The device read operation transfers data from the device to memory. This operation blocks until the request completes. The device write operation provides two options: writes may block as is the case for reads, or the write may be done asynchronously. In either case, the operation takes a block of data from memory and copies it onto the device. In the synchronous case, the caller is sure when the write actually is completed. This is an important concern to action management and to the recovery management portions of the storage manager. In the asynchronous case, the caller is allowed to flag write requests as "belonging" to an action. At a later time, the action may use the device flush operation to determine when the action's requests are complete. The flush operation uses the flush table discussed earlier. This operation is particularly important to the storage manager in the performance of its recovery management duties, as it allows the actions to perform asynchronous writes to secondary storage, while still maintaining control as to when these writes complete.

1.2 The Partition Object

Each partition object resides completely on one physical device. 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 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 possibly bear no relationship to each other. The partition concept is simply an administrative organization imposed by the storage management system indicating how storage in a particular partition is managed. For example, some partitions might be used for the storage of object data while others are used simply as backing storage. Different partitions may manage and allocate storage differently. Partitions may be defined that provide some specific recovery support, for

example a log partition. Partitions may simply be used to categorize objects to such classes as recoverable, non-recoverable, or temporary. In summary, partitions provide additional flexibility to the Clouds design. Because all partitions will provide essentially the same interface, new storage management features can be added in a transparent manner.

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.

1.2.1 Partition Data Structures A partition is described by a *partition header* containing most of the information found in the medium index table entry for this partition, plus information about the partition's state and type. The type information specifies what the partition is used for: storage of object data; storage for paging; and any other special purpose storage required by the kernel. The partition object uses this information to decide what structures are required to support the object. For example, paging partitions do not need directories.

Storage allocation for the partition is done using the *allocation map*. The allocation map is not permanent. Instead, it is reconstructed whenever the partition is activated. Handling storage allocation in this manner made support for action event handling more straightforward and more efficient, since the shadowing technique used requires allocation of partition storage for the block copies created. Since the allocation map is volatile, no special overhead is required to make the allocations and deallocations recoverable. Reconstruction does produce significant overhead at the time of system restart. Generally, however, this overhead is necessary in any event because the secondary storage system, which contains the permanent states of objects, must be examined on restart to ensure the consistency of the data residing on secondary storage. This is particularly true after a site failure in which action events may have been interrupted. If it could be ensured that the storage system was in a consistent state when the site is halted, then this overhead is unnecessary. The overhead could be avoided in these cases by simply writing the allocation map to secondary storage when a site is halted gracefully.

Another structure used by the partition object is the *active partition table* (APT), which contains *active partition descriptors* (APD) for partitions currently available to the system. Each APD in the table associates a partition sysname with the data structures and information for that partition. The 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 reference to the device object on which the partition resides.

Another task of the partition object is to maintain the location of segments and make this information available upon request. As mentioned earlier, access to an object involves a search. For objects which have not been accessed recently, the search generally involves querying the active partitions on the various sites to determine where the object resides on secondary storage. Each partition therefore maintains a partition directory, which contains a sysname/PBN pair for each segment residing on the partition. At this time there is no restriction on the format of the partition directory other than the requirement that any entry in the directory must reside completely within one secondary storage block.

1.2.2 The Maybe Table As can be imagined, such searches can be time-consuming. The partition system maintains another structure, called the *maybe table*, which it uses to avoid unnecessary secondary storage accesses altogether (or at least make such accesses rare). The maybe table is an approximate membership checker. It indicates either that the object in question definitely does not reside on the partition being checked, or 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.

Figure 2 illustrates the use of the maybe table. It is the first stage of a search for both a local request from the site's object manager and a remote request from the RPC mechanism. A good deal of overhead and time is saved when the maybe table indicates that the object is not at the

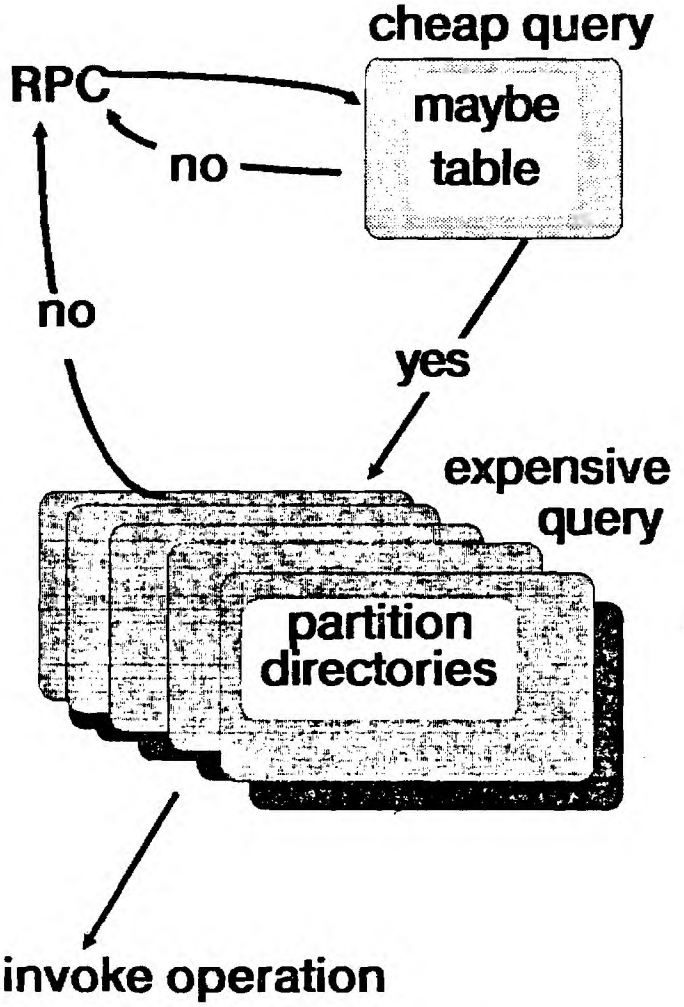


Figure 2. The maybe table

local site because the maybe table query is cheap (it is an in-memory query) and no slave process is created. Even in the case of a positive response, indicating that the object may be local, the maybe table query overhead is much less than the overhead incurred by partition directory queries. If the object resides locally, the directory queries are necessary to locate and activate the object (bring it into memory for use) and the maybe table query is a small part of this procedure. If the object is not local, then the additional work was done to uncover this fact, but this additional effort is small and with good performance on the part of the maybe table it is not frequent. The maybe table mechanism provides an excellent means of short circuiting local searches.

The maybe table for a site is reconstructed from the partition directories upon site re
other occasions during which a partition is mounted. In a running Clouds
population of segments at a site is dynamic. Segments may be created and dest
may migrate to and from other sites. When a new segment arrives at a
creation or migration, an entry for the new segment is added to the
entering the segment's sysname into a partition directory. This allo
represent the segment population at a site. Deletions of segments sh
the maybe table.

1.2.3 Functionality The functions provided by the partition object include partition management, directory management, storage allocation, and data transfers. Partition management consists of three operations, the first of which is the creation of partitions. Creation of a new partition automatically activates the partition in addition to allocating its storage. Partition creations also initialize the structures associated with the partition, such as the APD, the directory, and the allocation map. Partitions are destroyed by deallocating the secondary storage on which they reside. Only partitions which are not active may be destroyed. Activation and deactivation are two other partition management functions. Activation makes an existing partition available for use by the Clouds system. It involves not only the creation of an APD for the partition, but also the examination of the partition for consistency. Deactivation makes a partition unavailable.

Directory management is concerned with registering and searching for segments which may reside on the partition. Generally, names are entered for newly created segments, and removed for destroyed segments, but similar management takes place for segments being moved from one partition to another. Two other operations are available which provide a means to list the segment names stored in the directory. These operations are typically used to construct the maybe table at system restart, or to reconstruct the table in order to remedy degradation of the table's performance. The partition is also responsible for the allocation and freeing of blocks of storage for use by segments or the virtual memory system. The two operations responsible allow callers to allocate storage in multiples of blocks. The blocks allocated or deallocated might not be contiguous; this is not a concern since segment storage is not extent-based.[1]

Lastly, three operations for data transfers are provided. They are similar in functionality to those provided by the device objects. The partition read request blocks until the request completes. The partition write operation provides both blocking and non-blocking transfers. Support for recovery is provided both in the write operation, which allows requests to be flagged by an action in the same manner as device requests, and by the flush operation, which provides the same function as the device flush operation.

1.3 The Segment Object

The segment object type provides the final level of abstraction for secondary storage. The abstraction provided by the segment object is that of a sequence of bytes (kernel segment type). Segment objects provide a standard abstraction for the kernel to manipulate and process all Clouds objects; indeed, in some cases, a segment object is just an alternate type description for a Clouds object. However, the mechanism is more general, in that an object may be represented by several segments. For instance, an object may have a code segment and data segment which reside on secondary storage. In cases such as these, the sysname of the object's data segment is equivalent to the object's sysname. The object implementation provides mechanisms for mapping segment data into and out of virtual memory, creating and destroying segments, and modifying segments. Thus, segments have two different representations: one on secondary storage, and the other in virtual memory. 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 sequences of bytes. Structural interpretation of segments is performed by other parts of the kernel, such as the object manager. The storage manager is aware of and can recognize the administrative portions of an object's data, specifically the *object descriptor*. This allows the storage manager to provide low level support for the creation and initialization of objects.

---

1. Extent-based file systems allocate storage for files in very large chunks, such as a cylinder at a time. Since large portions of the file are contiguous on the device, sequential access to the files is enhanced.

1.3.1 Segment Object Representation Recall that a partition directory has a set of entries which contain the partition block numbers for the segments residing on the partition. The partition block addressed by any of these entries contains a *segment header* which identifies the segment. The segment header consists of the *segment descriptor*, which contains the information which describes the segment, such as the size, type, and state of the segment. The header also contains the *segment map* through which the segment data can be accessed. Each entry in the segment map contains a PBN of some other part of the segment. The remainder of the segment is constructed of *mapping blocks* and *data blocks*. Mapping blocks are internal nodes of a tree formed by the segment and contain the PBNs of other mapping blocks or to the data blocks of the segments. The data blocks contain the segment data.

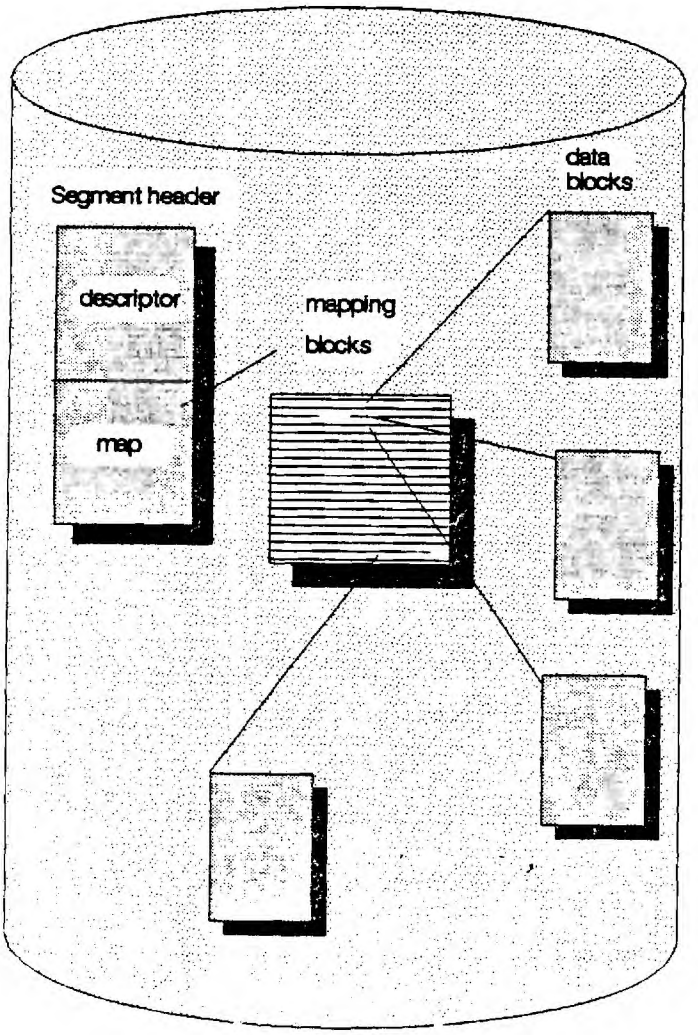


Figure 3. A segment object on secondary storage

Figure 3 shows the relationship of the described structures in a segment as it exists on secondary storage. The author will frequently refer to the data blocks of the segment as the segment pages.[2] Data blocks are always found at the leaves of the segments' mapping block trees. A

segment may require zero, one, or more levels of mapping blocks to access its data, depending upon the size of the segment. All of this structure is invisible outside the storage manager. Other parts of the kernel see only the data blocks of the segment.

1.3.2 Virtual Memory Support Any segment object may have two instantiations: one on secondary storage and the other in virtual memory. The representations of these two instances are quite different, as are their functions. The instance on secondary storage is intended to represent the permanent state of the segment data; that is, this instance remains available after recovery from system failures. The virtual memory instance exists for manipulation.

Segment objects are used by the Clouds kernel to represent Clouds objects or portions of Clouds objects. For example, an object may be represented by a single segment which contains all of the code and data necessary to perform operations on the object. On the other hand, an object may be partitioned for policy reasons into several segments. One segment may contain the data, another segment the operation code, and yet another may provide the object with dynamic heap storage.[3] The segments necessary to provide access to an object and to allow operations to be performed on the object are mapped into virtual memory through the cooperation of the object management and storage management. The storage manager maintains the *active segment table* (AST), which contains an entry for each segment mapped into virtual memory. These entries are called *active segment descriptors* (ASD). Any segment with a descriptor in the AST is said to be an *active* segment. Object management maintains a similar table, called the *active object table* (AOT).[Spaf86] Similarly, objects referenced by the AOT are active. Segments are referenced by the AOT to provide a complete virtual memory image of an activated object. Note that some of the entries in both the AST and the AOT may represent remote segments and objects, respectively. In these cases, the descriptors are not complete specifications of the segments or object, but simply refer the object manager to remote instances.

Each object refers to the various segments that comprise its virtual memory image through entities called *windows*. A window is simply a consecutive block of bytes in virtual memory. Each window in the system is described by a *window descriptor*, which specifies where the window is mapped, how large it is, and protection information. The window descriptors provide the primary interface between the active object system and the active segment system. Windows may describe whole segments or only portions of segments. In the case of a large file object, for example, it may be convenient to have only a portion of the data segment actually mapped into virtual memory. The window describes which portion of the segment is mapped. A segment may be described by several windows, allowing segments to be shared by several objects. As an example, the code segment for an object may be mapped into several object instances. Windows into segments may be mapped on demand. For example, a process with a window into a large object may cause the window to be modified or a new window to be created by referencing a part of the object that is not mapped by the current window.

The storage manager is responsible for specifying how the secondary storage image of a segment is mapped into virtual memory. The ASD refers to the APT to indicate the partition on which the segment resides. The ASD also refers to mapping tables which are maintained by the storage manager. These tables map the virtual memory image of the segment to the secondary storage image. Figure 4 illustrates the structures used in the mapping of segments.

---

2. This is rather imprecise terminology in that it gives the impression that virtual memory pages and secondary storage blocks are equivalent. This is not a restriction in the design of the storage manager, but the initial implementation makes this assumption and the equivalence will make some of the following discussions simpler.

3. Not all of these segments may have permanent states. Segments used to map volatile heap space for objects have no image on secondary storage except for backing storage for page-fault handling.

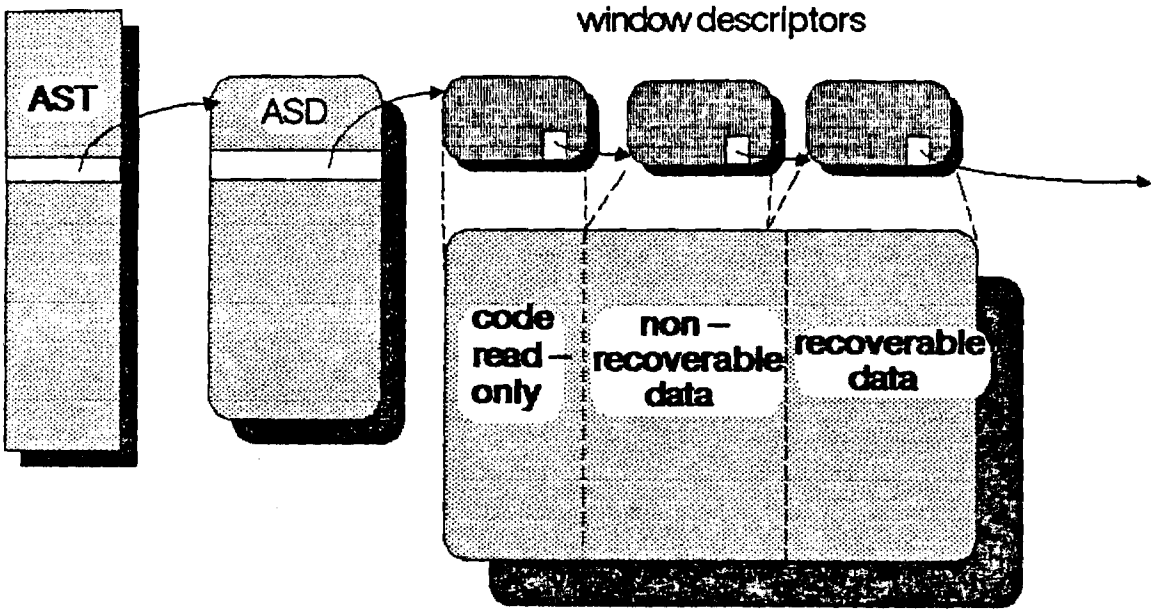


Figure 4. Mapping structures used in the storage manager

1.3.3 Segment Object Functionality The operations provided by the segment object type may be classed into four types: segment management, virtual memory management, data transfer, and recovery management. Segment management includes operations to create and destroy segments. When a segment is created, it is automatically activated by the system so that it may be operated upon. The initial segment descriptor is allocated and the segment is registered with the partition directory (and also the maybe table). The destruction of a segment can occur only when all activity on the segment has ceased. The segment remains in the AST, but no windows are mapped into the segment. No activity can occur on the segment in this case, and the segment storage may be deallocated and the directory entry removed. Both creation and destruction of segments can be recoverable if done by an action. Segment objects also provide operations that can change the size of the segment and determine the status of the segment.

The second type of operation, dealing with virtual memory management, provide the bridge between the virtual memory instance of a segment and the image on secondary storage. Part of the functionality of this group is simply to activate the segment. This includes locating the segment,[4] creating an ASD, and partially initializing the ASD. Other operations then allow windows to be mapped into the active segment and supply the information which maps the segment data into virtual memory. Later, these window mappings may be modified or destroyed by other operations.

---

4. In some cases, remote segments are activated by local storage managers. This situation generally arises when the segment containing the object data is local, while the segment containing the code resides on another site. The data segment is activated locally as usual and the code segment is activated remotely; virtual memory management for the segment is shared between the sites. Backing storage is provided at the local site if a local cache is desired, but all pages are initially fetched from the remote site on demand. This facility is available only for read-only segments, such as code segments.

The operations that perform data transfer provide the means to move segment data between the image on secondary storage and the virtual memory image. Principally, these operations are used for page-fault handling, reclaiming physical pages, and to support action management. They use the information set up by the mapping operations. The pages containing the data to be transferred must be mapped into virtual memory by the ASD.

2. The Segment Module

2.1 The Storage Manager Prototype

This section describes the prototype segment system, examining the structures used by the system and the interface provided to the rest of the storage management system. The definitions in this section (and the succeeding ones) are written in the C programming language and are taken from the source code for the storage manager. The storage management code is found in several C source files. The following list presents a synopsis of the contents of these files:

segment.h Segment.h contains most of the type definitions used by the segment system. In addition, the file defines a set of named masks and constant codes used by the routines in the segment module.

obj.h This file provides definitions for structures used for object manipulations. Because of the close relationship between objects and segments, a couple of the definitions used by the segment system are located in obj.h.

segment.c Segment.c contains the operations that implement the segment system. The file also declares global variables and structures needed by the segment system. The routines found this file fall into two broad categories: interface routine, and utility routines. Utility routines are defined as static, meaning that they cannot be seen outside the file. The interface routines are visible and may be used anywhere in the kernel.

parttab.h This file defines the active partition descriptors and table.

parttab.c Parttab.c defines a set of routines which manipulate the active partition table. The routines provide a simple interface for creating and destroying active partition descriptors. The routines also enter, locate, and remove the descriptors. Otherwise, descriptors are manipulated directly. Parttab.c also declares a set of partition lists which facilitate the implementation of several partition system services.

partition.h Most of the data structure definitions used by the partition system reside in this file. The formulas for determining the size of partition structures such as the directory and allocation map are defined here.

partition.c The interface to the partition system is defined in this file. The partition system defines no local utility routines.

devtab.h Devtab.h defines the active device descriptors and table.

devtab.c A set of routines analogous to those defined in parttab.c are found in devtab.c.

buffer.h The request packet structure for the RL02 device is defined in this file

buffer.c Buffer.c contains the routines used to allocate request packets for the RL02 device. A pool of such packets is maintained by the device.

rl_dev.h Rl_dev.h defines a series of mnemonic codes used by the RL02 device module.

rl_dev.c This file contains the routines used to implement the RL02 device object. The routines consist of a set of interface operations available to other parts of the storage manager and private utility routines visible only inside rl_dev.c.

storeman.h This file defines constants important to the storage management system as a whole.

Descriptions of other files making up the Clouds kernel prototype may be found in [Spaf86]. The remaining sections in this document do not attempt to describe the contents of these files in total detail. Only the major structures presented earlier in the first section, interface routines, and a few important utility routines are discussed. There also many mnemonics and macros defined to facilitate the maintenance, readability, and the implementation of the storage manager. These are important to a complete understanding of the prototype. The reader is referred to the above files for these definitions.

2.2 Some General Definitions and Notes

The remaining sections refer to structures and definitions not described in this document. In addition to the standard C types, many typedefs are defined in the prototype for convenience and necessity. A summary of the some of the important definitions is presented in this section. The major attributes presented will be the size of the structure and it purpose. In many cases the internal format will not be specified.

u_int This is an unsigned integer. On the VAX integers are by default long integers (32 bits). There is a corresponding definition for u_short, an unsigned short integer (16 bits long).

address This type definition represents a generic address. It is actually a typedef of u_int. On the VAX, pointers and addresses happen to be the same length (32 bits) and have the same format.

QH QH is a type definition for a VAX hardware queue header. QH is 64 bits long and consist simply of two u_ints, one a pointer to the head of the queue and one a pointer to the tail.

QE This type definition represents the linkage fields needed for an element on a VAX hardware queue. Like QH, this structure is 64 bits long and must be aligned on a quadword (multiple of 8 bytes) boundary. It is also two u_ints, one a forward pointer, and the other a backward pointer.

SYSNAME This type represents a Clouds sysname. Sysnames are 48 bits long and contain a site id, a site unique id, and a type.

PMAP This is a typedef for a page table entry. It is nothing more than a long unsigned integer (32 bits).

The sections present the definition used for each of the major sub-components of the storage manager. This section presents the segment module. The next section discusses the partition system implementation. The third section describes the RL02 device module. For each structure and routine, the C definition and the file in which the definition is contained is provided. It is hoped that through these sections, it is possible for an interested reader to quickly gain a familiarity with the storage management prototype.

2.3 Segment Module Structures

This section presents the major structures used by the segment system. The source files for

these definitions are segment.h and obj.h.

2.3.1 Segment Descriptors

```
typedef struct {
        SYSNAME             segname;
        SYSNAME             actname;
        u_int               size,
                            raoffset,
                            phoffset;
        u_int               header;
        u_int               shadow;
        u_int               state;
        u_int               unused[55];
        u_int               indices[64];
}   SegHdr;

typedef struct
{
        QE      links;
        QH      windows;
        SYSNAME     segname;
        address offset;
        u_int   length;
        address hdroff;
        address segpart;
        address backpart;
        u_int   usecount;
        u_int   state;
        address hdr;
        PMAP    *vdmap,
                *vpmap;
} ASD;
```

These two definitions provide the secondary storage and virtual memory descriptors for a segment. **SegHdr** is the segment descriptor that resides on disk and which is the root of the segment tree. **SegHdr** is 512 bytes long and so fits into a secondary storage block. Because of alignment restrictions imposed by the C compiler, the structure is not compact; there are unused holes not represented by the definition. The fields have been discussed previously and there is not much more to add, except to note the units used by some of the fields. **Size**, for example, is the segment size in bytes. The header field indicates the number of storage blocks used by the object header. **Raoffset** and **phoffset** are both block offsets. **Shadow** is used during recovery processing and is the PBN of the shadow segment descriptor.

The active segment descriptor is also presented in this section. The active segment table is organized as a hashing table using the VAX queue mechanism. The ASDs are allocated dynamically from the system heap. **Usecount** is always the number of window descriptor referenced by the **windows** field. However, **windows** is also used to hold the commit record used by recovery management, so that **usecount** does not always represent the length of the queue of descriptors.

Offset and **length** are both in block units. **Offset** is the base of the mapped version of the segment; it indicates the lowest segment page mapped into virtual memory. **Length** indicates the extent of the mapped region of the segment. In general, **length** may not specify a contiguously mapped region of the segment. In an extreme case, only the first and last blocks of a segment might be mapped. **Offset** would contain the offset of the first page and **length** would be the size of the segment in blocks.

Hdroff is the PBN of the segment descriptor on disk. The correct partition can be found through either the **segpart** field or the **backpart** field. These both refer to APDs. Only one

such field is necessary for any given segment, but the ASD retains two fields for historical reasons.

The **state** field represents a combination of status information, primarily dealing with recovery management events, and segment type information. The field is a bit field. Masks are defined in segment.h for querying the field to determine the segment status and type.

The **vpmap** and **vdmap** fields represent the virtual page table and disk block table, respectively. Both are dynamically allocation arrays of PMAP-typed entries. The entries of both tables are bit strings with various fields defined. The reader is referred to the VAX Hardware Manual [DEC82] for the definitions of the fields in these entries, as they are identical to the page table entries described there. Also, see [Kenl86] and [Spaf86] for some of the software defined bits in these entries. The format of the **vdmap** entries is defined here. Recall that any entry may be in one of two formats, one representing an unshadowed page and the other representing a shadowed page. Both formats are presented below:

Alt bit (bit 0) - The **Alt** bit controls the format of the rest of the entry. If the bit is clear, the page is unshadowed; otherwise, the entry represents a shadowed page.

Type field (bits 1-2) - The **type** field is present in both formats and indicates the page type (recoverable, non-recoverable, volatile, or read-only).

AltListHdrPtr (bits 3-31) - This field is present only when the **Alt** bit is set, indicating a shadowed page. In this case, the field points to an **AltListHdr** structure which represents the page and its shadows. Since all system heap memory is allocation of quadword (8 byte) boundaries, the lower three bits of the heap address are zero. This fact is used to retain the **type** and **alt** bits in the entry. These bits are masked off when referenced the **AltListHdr**.

Mapped bit (bit 3) - This field is present only when the **Alt** bit is clear. The bit indicates that the **Dbn** field of the entry is valid.

Cow bit (bit 4) - The **Cow** bit is the copy-on-write bit. It indicates that the page is part of an action version.

Dbn (bits 11-31) - This field contains a PBN for the segment page.

Bits five through 10 are unused in the **Alt** bit clear format.

2.3.2 A Structure for Defining Windows into Segments

```
typedef struct
{
        QE ASDlinks;
        ASD *segmnt;
        SYSNAME segname;
        address begin;
        u_int length;
        u_int offset;
        u_int mask;
} WindowDesc;
```

The **WindowDesc** structures are used to map portions of the segment into virtual memory. The begin field is the base virtual address of the window described by this descriptor. **Offset** is the base segment page. **Length** is the length of the window in bytes. The **mask** field provides two types of information. The first is the window type: read-only, volatile, remote, or non-pageable. The second type of information is the virtual memory protection bits used in the page

tables. These bits are defined in the kernel source file vm.h.

2.3.3 Alternate List Headers

```
typedef struct {
        QH      links;
        u_short count;
        address offset;
        PMAP    original;
} ShadowPmap;
```

When a **vdmap** entry refers to a shadowed page (the **Alt** bit is set), the **AltListHdrPtr** field refers to a structure of this type. **Count** is the number of versions represented by this entry (not including the base version of the page). The **links** field contains a queue of descriptors of these versions. **Offset** is the segment offset of the page being described. **Original** is the base version of the page. It uses the **Alt** bit clear format of a **vdmap** entry. These structures are created dynamically as needed.

2.3.4 Shadow Entries for Virtual Memory Support

```
typedef struct {
        QE      links;
        SYSNAME name;
        PMAP    shadow;
        ShadowPmap *shadowentry;
} ActionPmap;
```

ActionPmap represents the action versions of segment pages. Structures of this type are the queue elements found in from the link field of the alternate list header. **Name** is the sysname of the action the page of which this structure describes. **Shadow** describes the pages using the format of a **vdmap** entry (with the **Alt** bit clear). **Shadowentry** is a reference back to the alternate list header maintained for convenience.

2.3.5 Recovery Structures

```
typedef struct {
        QE ASDlinks;
        u_int    shdr;
        SYSNAME actname;
        address *old;
        address *new;
        u_int    ocount, ncount;
        address *indexshad;
        u_int    nindexshad;
        QH       *chglist;
} ActionDesc;

typedef union {
        WindowDesc *w;
        ActionDesc *a;
} CommitDesc;

typedef struct {
        QE links;
        address new, old;
} CRLIST;
```

These structures are used only during recovery management. **ActionDesc** describes a precommitted segment, both in memory and on disk. Shdr is an in-memory cache of the shadowed segment header. **Old** is an array containing the PBN of partition blocks to be deallocated on a successful commit. New is an array containing the PBN of the shadow blocks,

which replace the blocks in the **old** field on a successful commit. **Ocount** and **ncount** indicate the number of blocks on each of these lists. **Indexshad** is an array containing the address of mapping block buffers. The buffers are being written to disk asynchronously, and as the buffers are dynamically allocated, the storage manager must maintain a record of their existence until the writes are completed. **Nindexshad** is the number of such buffers. **Chglist** is a queue of elements of type **ActionPmap**. Each element represents the virtual memory-secondary storage mapping for the action version of a segment page. These elements are taken either from the **vdmap**, or are created by **S_Precommit**. The **CommitDesc** structure simply maps an **ActionDesc** structure onto a **WindowDesc** structure. This is because the two structures both reside on the **windows** field of the active segment descriptor.

CRLIST is used during crash recovery to collect blocks allocated for the permanent and shadow versions of a precommitted segment.

2.4 Segment System Interface

2.4.1 Locating Segments

```
int    S_Find (segname)
SYSNAME segname;
```

S_Find determines whether the segment is local or not. The operation queries the AST, the maybe table, and the partition directories at this site. A successful query of the AST means that the segment is already activated. Otherwise, the segment is either dormant, remote, or unavailable. The maybe table and directories are used in the manner described previously. **S_Find** returns success for local segments and failure for remote segments or unavailable segments.

2.4.2 Activation of Dormant Segments

```
int    S_Activate (segname, header, number)
SYSNAME segname;
address * header;
u_int * number;
```

This operation activates a local segment by creating an active segment descriptor. **S_Activate** is used only for data segment and object segments. Remote segments and volatile segments use **S_MapWindow**. If an object header exists, it is read into a buffer, which is returned in the parameter header. **S_Activate** passes the size of the object header in blocks through the parameter number. The ASD for the segment is initialized with the segment attributes (name, header location on disk, type, etc.). The **offset** field of the ASD is given the number of data pages in the segment, and the **length** field is assigned to zero. This sets the ASD for the addition of the first window to the segment. See the section on **S_MapWindow** for more details. The segment header on disk is read into a buffer the address of which is placed in the

ASD. The operation returns an indication of the success or failure of **S_Activate**.

2.4.3 Operations for Mapping Segments into Virtual Memory

```
int    S_ModWindow (wptr, vpmap)
WindowDesc * wptr;
PMAP * vpmap;

int    S_MapWindow (wptr, vpmap)
WindowDesc * wptr;
PMAP * vpmap;
```

S_MapWindow maps a new window into a segment. In the case of remote and volatile segments, a new segment may be created. Data and object segments are assumed to have been activated by **S_Activate.** Remote activation is done for remote windows. A object header block may be returned for remote windows in the argument header.

The window descriptor is passed to **S_MapWindow** with some information already supplied. This includes the base segment page of the window, the base virtual address of the window, and the length of the window in bytes. The **mask** field is also set. The storage for the **vpmap** parameter must be allocated prior to the call. The **vpmap** parameter points into a page table for an object using the segment and **S_MapWindow** initializes this page table from the virtual page table maintained in the ASD. For object and datafile segments, **S_MapWindow** first determines whether the mapped range of the segment must be modified; i.e., whether the lower or upper bound or both bounds must be extended to accomodate the new window. This will always be the case for the first window mapped into a segment due to the way in which **S_Activate** initializes the ASD. The extension of the segment requires that new storage for the descriptor's virtual page table and disk block table be allocated and the new area initialized from the old tables. The areas for the newly mapped portion of the segment must be initialized to the default values for the segment. The **usecount** is incremented and the window descriptor is added to the **windows** field of the ASD.

For volatile and remote windows, **S_MapWindow** also creates the ASD. Note that these segments do not have a permanent image at this site. For a volatile window, the call to **S_MapWindow** performs both an activate and a window map. For remote segments, **S_MapWindow** in addition must perform the remote mapping protocol discussed in Chapter IV. Remote mapping is not currently implemented in the prototype.

S_ModWindow modifies an existing window allowing the caller to extend or shorten a window. The return value of the operation indicates its success or failure.

2.4.4 Removing Segment Data from Virtual Memory

```
int    S_UnMapWindow (wptr, vpmap)
WindowDesc * wptr;
PMAP * vpmap;
```

S_UnMapWindow is the complement to **S_MapWindow**. The specified window is unmapped from the segment. An examination of all windows in the segment is necessary to support overlapping windows. If the window being unmapped does not overlap with another window in the segment, then all of the physical pages which map the window may be freed. The parameter **vpmap** supplies a record of these pages. If the window does overlap some other window, then none or only part of the physical page may be released. The removal of the window may also cause the size of the mapped segment area to decrease. In this case, as in **S_MapWindow**, new virtual page and disk block tables must be allocated (smaller than the previous ones) and the new tables must be initialized for the old ones. The window descriptor must be removed from the **windows** field and the **usecount** is decremented. **S_UnMapWindow** is not implemented in the prototype. The return value of the operation indicates its success or

failure.

2.4.5 Initializing Segments

```
int S_LoadS(source, soff, dest, doff, len)
SYSNAME source, dest;
address soff, doff;
u_int len;

int S_LoadM(addr, dest, doff, len)
SYSNAME dest;
address addr, doff;
u_int len;
```

These operations provide the means by which a newly created segment can be initialized with the appropriate data or code. **S_LoadS** initializes a segment from another segment. **Source** and **dest** are the sysnames of the initializing segment and the initialized segment, respectively. **Soff** and **doff** are the offsets into these segment at which the initialization takes place. **Len** indicates how much data is initialized (in bytes).

S_LoadM performs the same function except that the newly created segment is initialized from virtual memory. **Addr** contains the base address of the area from which the new segment is initialized. **Doff**, **len**, and **dest** are as in **S_LoadS**. Both operations return a value indicating the success or failure of the operation.

2.4.6 Segment Creation

```
int    S_Create (partname, segname, size, hbsize, hblock, type, raoffset, phoffset)
SYSNAME * partname, *segname;
u_int size;
u_int hbsize;
address hblock;
u_int type;
address raoffset, phoffset;
```

S_Create creates a new permanent segment. (Volatile segments are created by **S_MapWindow**). Recoverable segments are labelled "CREATED" to facilitate recovery processing. The following structures are created and initialized:

1. An active segment descriptor is created for the segment. The size and length of the segment are initialized as in **S_Activate**. The attributes for the segment are initialized.
2. A block for the segment descriptor on disk is allocated, along with a virtual memory buffer for the descriptor. The descriptor is initialized from the parameters and the locations of both the volatile and permanent version of the descriptor is placed in the appropriate ASD fields.
3. Storage blocks for the object header are allocated. The parameter **hbsize** gives the size of the header in blocks, while the **hblock** parameter is a pointer to the buffer containing the object header. The header is written to disk and the PBNs of the blocks containing the header are stored in the segment index.
4. Any mapping blocks required by the segment are allocated at this time. These blocks are initialized to zero indicating that no data blocks currently exist for the segment. The PBNs for the mapping blocks are stored in the **index** field of the disk segment descriptor. It is only at this time that the segment descriptor is actually written to disk.

For non-recoverable segments, an entry is made in the partition directory for the segment. Recoverable segments have no entry made during creation; this is done only at commit. The

S_Create operation returns a value indicating success or failure.

2.4.7 Destroying Segments

```
int    S_Destroy (partname, segname)
SYSNAME partname, segname;
```

This operation removes a segment from the partition. **S_Destroy** is only applicable for data segments and object segments. Generally, **S_Destroy** removes the permanent segment state and the active segment descriptor (deallocates them), while recoverable segments simply are labelled as "DELETED" for future recovery processing.

The segment must be activated before the destroy operation can be perform. Any windows mapped into the segment must be unmapped prior to the call. The return value of the call indicates success or failure.

2.4.8 Segment Reads

```
int    S_Read (sptr, offset, addr)
ASD * sptr;
u_int offset;
address addr;
```

S_Read reads a segment block from disk to a physical page frame. As discussed in Chapter IV, **S_Read** is a case analysis on the type of page and the state of that page's mapping. **S_Read** potentially modifies the **vdmap** field of the ASD for the segment. For example, the first time a segment page is read, there is no entry in the appropriate **vdmap** entry. **S_Read** locates the appropriate partition block and places the PBN in the entry. If the segment page has never been written, a new partition may be allocated. Thus the side effects of **S_Read** include not only the virtual memory page receiving the data, but potentially the **vdmap**, the segment descriptor on disk, and mapping blocks used by the segment. **S_Read** returns a value indicating that the operation succeeding or failed.

2.4.9 Segment Writes

```
int    S_Write (sptr, offset, addr)
ASD * sptr;
u_int offset;
address addr;
```

This operation performs a write to a segment block from a physical page frame to a segment block on disk. As with **S_Read**, this operation performs a case analysis to determine the appropriate measures to apply to the page in question. The side effects are similar to those of **S_Read**. In addition, entries in the **vdmap** of the ASD may have their formats changed to alternate list pointers, in the case of writes to recoverable pages. The return value of this routine is either success or failure.

2.4.10 Phase I Recovery Support

```
int    S_Precommit (actname, touchlist, number)
SYSNAME actname;
SYSNAME * touchlist;
u_int number;
```

S_Precommit shadows the number of segments indicated by its second and third parameters. The caller passes the names of objects touched by a committing action through the **touchlist** parameter, but by convention these sysnames are equivalent (agree in all but the type) to the names of the data segments for the object. Several major functions are performed.

1. The first step taken is to determine which data pages have been modified. For a newly created segment or a deleted segment, all data pages are considered modified. For

modified segments, the object page table (found through the object descriptor [Spaf86]) and the disk block table are examined. The object page table will indicate which pages have been modified. These must be moved into shadow blocks on secondary storage. The disk block table will show which pages have already been written to disk. Shadow blocks for such pages effectively have been allocated; however, it may still be necessary to write the pages if the object page table indicates the page has been modified since it was moved to disk. Note that this is information which is available through the normal operations of virtual memory. A list of changed pages must be kept, along with enough information to update the disk block tables during commit. The information contained in the **ActionPmap** structure is satisfactory for this purpose. The entries are either created and initialized by **S_Precommit**, or they are found already in the disk block table.

2. Next, **S_Precommit** must determine which, if any, mapping blocks must be shadowed. For each mapping block that is shadowed, a page-sized buffer is allocated to do the necessary modifications. The memory for the buffers must be contiguous because the buffers must be held till the end of **S_Eoa**. The contents of the modified buffers are written to disk.

3. A commit record is created if any modified pages exist. Pointers to the entries for the modified data pages and the buffers for the modified mapping blocks are placed in the **chglist** and **indexshad** fields of the commit record, respectively. From the information contained in the entries on the **chglist**, the **new** and **old** fields are set to contain the PBNs of the shadowed blocks and the modified blocks of the segment, respectively. The **new** field is empty for a deleted segment, while the **old** field is empty for a newly created segment.

4. The last update is to the segment descriptor. A storage block on disk and a buffer page in memory are allocated to hold the modified descriptor. Their locations are placed in the commit record. The buffer is modified so that the segment index refers to the shadow blocks, and is written to the shadow block for the descriptor. The segment descriptor itself is updated so that the state of the segment is either PRECOMMITTTED, CREATED, or DELETED. The record is placed in the **windows** field of the segment ASD.

5. After all segments have been processed, the operation flushes all remaining write requests using a call to **P_Flush.**

The steps are taken for each segment listed in **touchlist**. The return value of the operation is an indication of the success or failure of the operation.

2.4.11 Phase II Recovery Support

```
int    S_Eoa (actname, touchlist, number, flag)
SYSNAME actname;
SYSNAME * touchlist;
u_int number;
char   flag;
```

S_Eoa performs either a commit or abort on the segments indicating in the second parameter. The desired procedure is specified by the flag parameter. **S_Eoa** locates the commit record created by **S_Precommit** and does the following:

1. The operation uses the entries in **chglist** to modify the **vdmap**. If the flag indicates a commit, the **chglist** entries replace the **vdmap** entries. If the flag indicates an abort, the entries are simply destroyed.

2. Any buffers for mapping blocks are deallocated.

3. If the operation is performing a commit the partition blocks in **old** are deallocated. If an abort is being done, the partition blocks in the **new** field of the commit record are

deallocated.

4. The directory entry for the segment is modified to reference the shadow segment descriptor if a commit is being done. Otherwise, the entry is left unmodified. However, if the segment is being deleted by the action, the directory entry must be clear on a commit. For a created segment, the directory entry was modified by **S_Precommit**. In this latter case, **S_Eoa** must restore the directory entry when the operation is performing an abort.

The operation returns success or failure.

2.4.12 Crash Recovery Support

```
int    S_Check (pptr, segname, hdroff, allocs)
address pptr;
SYSNAME segname;
address hdroff;
address *allocs;
```

S_Check is called by **P_Restore** as part of system restart. The sysname indicates which segment is to be examined. **P_Restore** calls **S_Check** once for each segment found in the partition directory. **Hdroff** contains the location of the segment descriptor on the partition. **Allocs** is a large array through which a set of PBNs is passed to the caller, **P_Restore**. **P_Restore** uses these PBNs in the reconstruction of the partition allocation map. The purpose of **S_Check** is two-fold:

1. The basic purpose is to determine the allocation for the segment. This requires a traversal of the segment blocks. For small segments (less than 64 kilobytes), the allocation of storage for the segment can be determine solely through the segment index in the segment descriptor. For large segments, the mapping blocks must also be examined. This requires a buffer to read each mapping block to collect the PBNs in the block entries.
2. **S_Check** also examines the segment to determine whether or not there was an unfinished action event for the segment. This is determine by looking at the segment **state** field. If the state is PRECOMMITTED, CREATED, or DELETED, then the segment is shadowed and recovery processing must be performed.

For recovery processing, storage allocation information is collected as normal except that first, the shadow segment descriptor is located and brought into memory. Also, an ASD for the segment is created and placed in the active segment table. Then the two versions are examined in tandem. When the operation discovers a difference in the allocation for a segment page, both PBNs are kept. The one from the shadow version is placed in the **new** field of a **CRLIST** structure. The PBN from the permanent version is placed in the **old** field of the same structure. All of the PBNs collected are placed in the **allocs** parameter. The **CRLIST** structure is used to create a commit record for the segment which is placed in the **windows** field of the ASD for the segment. The commit record is not complete; for example, no virtual memory information is included. **S_Eoa** recognizes these sorts of commit records and processes them accordingly. **P_Restore** calls **S_Eoa** with the appropriate flag (commit or abort) based on a query to the kernel database. The operation returns number of blocks allocated for the segment.

2.4.13 Changing a Segment's Size

```
int    S_Chgsize (sptr, delta)
ASD * sptr;
u_int delta;
```

S_Chgsize appends **delta** extra bytes to the end of the segment. This changes the permanent segment on disk as well as the mapped in segment. Any new data blocks necessary are not allocated until data is actually written to them by a S_Write operation. However, the size change may require new mapping blocks, and these are allocated immediately and integrated

into the structure. New mapping blocks are initialized with null entries. A size change may cause a reorganization of the segment structure. For example, an increase in the segment size may require the allocation of mapping blocks, whereas before only the segment index was necessary. The operation returns success or failure.

2.4.14 Segment Status

```
int    S_Status (segname)
SYSNAME * segname;
```

The **S_Status** operation returns the status of a segment with respect to any action management processing taking place on the segment. This information is pulled from the **state** fields of the segment header or active segment descriptor. The return value of the operation is the status of the segment.

3. The Partition Module

This section describes the partition module of the kernel prototype. The major structures used by the module are described and the interface to the partition system is defined. The structures described in this section can be found in the files partition.c, partition.h, and parttab.h.

3.1 Partition Module Structures

The structure definitions described below are found for the most part in partition.h. However, definition of structures used by the partition system in general, such as the APT, are found in parttab.h.

3.1.1 The Partition Header

```
typedef struct {
        SYSNAME             partname, devname;
        u_int               start, extent;
        u_short             parttype;
        u_int               pagemap, pd, sd, phdr, shdr;
} partition_hdr;
```

The partition header is duplicated at either end of the partition. Both copies must be updates. The sysname contained in **devname** is that of the device on which the partition resides. All locations and sizes are in terms of blocks (512 byte). **Start** is a DBN, as it indicates the base location on the device for the partition. **Extent** is the size of the partition. **Pd** and **sd** are the locations of the partition header copies. **Pd** is set to zero and **sd** is set to the PBN of the last block on the partition. **Phdr** and **shdr** are the base locations of the directory copies. **Phdr** is set to one, and the PBN in **shdr** is determined by the partition size. **Pagemap** is currently unused, but is available for an implements of the allocation map which uses a permanent map version.

3.1.2 Partition Directories

```
typedef  struct {
         int       count;

         struct {
                   SYSNAME       segname;
                   u_int    pbn;
         }         list[MaxBuckEnt];

} pdbucket;
```

The above structure is the definition for a directory bucket. **Count** is the number of free entries in the bucket. Initially, **count** is set to **MaxBuckEnt**, the maximum number of entries that will fit in the bucket. This number is less than the size of the entries would seem to indicate because of alignment restrictions imposed by the C compiler. **MaxBuckEnt** is 41 for the prototype.

The remainder of the bucket is an array of entries. Empty buckets have a **pbn** field set to zero, as that partition block is not available for segment storage.

3.1.3 A Locking Structure for the Allocation Map

```
typedef struct {
        char    *map;
        char    inuse;
} mlentry;
```

There is a structure of this type for each page (512 bytes) in the allocation map. **Map** contains the address of the page and is use as the semaphore ID for that page's semaphore. The **Inuse** field is set after the semaphore for the page has been taken. The field is clear when the page is not being referenced.

3.1.4 A Partition Recovery Support

```
typedef struct {
        QE      links;
        SYSNAME     name;
        int     count;
        address list[8];
} PActDesc;
```

These structures provide recovery support at the partition level. **Name** is the sysname for the action that caused this entry to be created. Each entry contains a list of APDs for partitions to which the action has written. **Count** is the number of such partitions. Using the **device** field of the APD, **P_Flush** can call the necessary device flush operations.

3.1.5 Global Partition Support

```
typedef struct {
        QE      links;
        SYSNAME     partname;
        u_int   start, size, active;
        u_int   phdr, shdr;
        ADD     *device;
        u_int   pdsize, pmsize, pmbsize, pdbsize;
        u_int   pd, sd, pm, partattr, fspace;
        char    *pmbuf;
        mlentry *pmassgn;
} APD;

QH      APT[PTSIZE];
QH      pactlon[PActDescSize];
address partitions[32];
SYSNAME partitionsysnames[32];
address paging[4];
QH pdbuf;
```

The first structure is the definition of an active partition descriptor. The **links** field supports the hardware queue mechanism used to link the descriptor to the APT. **Partname** is the sysname of the partition. **Start, size**, and **parttype** describe the base location of the partition (it is a DBN), the length of the partition in blocks (512 bytes), and the type of the partition (paging or object). **Phdr** and **shdr** indicate the location of the partition header, but unlike the partition header, these field contain DBNs. The **pd** and **sd** fields, containing the base locations of the directory copies, are also DBNs. This reduces the overhead in accessing these structures. **Device** is a pointer to the active device descriptor of the partition's device. The sizes of the directory and allocation map are stored in **pdsize** and **pmsize**, respectively. **Pmbsize** and **pdbsize** are the sizes (in 512 byte pages) of the buffer areas for the directory and allocation

map, respectively. **Pmbsize** is always one, since there is a shared pool of buffers for all partition directories. **Pdbsize** is simply the size of the allocation map, since it is contained entirely in memory. **Fspace** is the amount of free space on the partition. Pm is unused, but is intended for a permanent allocation map version. **Pmbuf** points to the allocation map. The map is allocated from the system heap. **Pmassgn** points to an array of lock structures for the allocation map. The array is **pdbsize** long.

The active partition table is declared as an array of hardware queue headers. The APDs are placed in the table using the VAX queue instructions.

Partitions, **partitionsysnames**, and **paging** are all arrays kept by the storage manager for convenience. **Partitions** contains pointers to APDs for all active partitions. **Partitionsysnames** contains the sysnames of all active partitions. **Paging** contains the APDs of only paging partitions.

Paction is a hash table used to manage the **PactDesc** entries described earlier. It is managed similarly to the active partition table.

Pdbuf is the communal buffer pool for the partition directories. A single semaphore (with the address of **pdbuf** as its ID) is used to manage the buffers.

3.2 The Partition System Interface

3.2.1 Partition Creation

```
void P_Create (devname, size, partattr, partname)
SYSNAME * devname, *partname;
u_int size, partattr;
```

P_Create creates a new partition on the device specified by the parameter **devname**. In addition, **P_Create** activates the partition by creating an active partition descriptor and entering this descriptor into the active partition table. The parameter **size** specifies the size of the new partition in terms of blocks (512 bytes long in the prototype). **Partattr** is the type of partition created. Currently, the prototype provides support for only paging and object partitions. **Partname** is used to return the sysname generated by **P_Create**. The major tasks that **P_Create** performs are:

1. the generation of the partition header. The partition header contains the information passed as parameters to the operation, the newly generated sysname for the partition, and the starting address on disk for the partition. The later piece of information is obtained with a call to RL_Enter. The partition header is written to the beginning and end of the partition;
2. the creation of the (in-memory) allocation map for partition storage allocation. For each page in the allocation map, **P_Create** also generates a semaphore used to provide mutual exclusion on that page;
3. the allocation of buffer space for the partition directory, along with the generation of the read/write lock for partition directory itself and a semaphore to control access to these buffers;
4. the initialization of the partition directory on the partition. This requires a determination of the size of the directory (based on the partition size) and the initialization of each directory bucket to be empty;
5. the allocation map must be initialized so that the partition blocks containing the partition header (two copies) and the partition directory (also two copies) are shown as allocated. Also, since the allocation map is an integral number of pages, excess bits at the end of the last block are also set; and

6. an APD is created and entered into the APT. The descriptor is initialized with the partition sysname, its size and location on the device, the location of per partition structures. The per partition structures include the partition header, the directories, the allocation, etc. The partition descriptor also contains a pointer to the device on which the partition resides, giving access to the entry points for the partition;

Currently the partition system maintains one set of buffers for the partition directories for all the partitions. Each partition created adds an additional buffer to the set. The system keeps the allocation map in memory for performance and because partitions are not large in this first coding. Later implementations may add a buffer scheme for the allocation map. Synchronization is done at a page granularity for the allocation map and as a whole for the directory. However, the system maintains a read/write lock for the directory, unlike the simple semaphore locks for the allocation map. The return value of **P_Create** is either success or failure.

3.2.2 Removing Partitions

```
void P_Destroy (partname)
SYSNAME * partname;
```

This call removes a partition from the device on which it resides. The operation assumes that the partition is active and it has an entry in the partition table. The active flag in the APD is cleared so that no further operations are performed on the partition. **P_Destroy** is the complement of **P_Create**. Everything that was allocated in **P_Create** is deallocated in **P_Destroy**. Locks and semaphores are taken before they are removed. The return value of the call indicates success or failure.

3.2.3 Directory Management—Entering Data

```
int     P_Enter (pptr, segname, pbn)
        address pptr;
SYSNAME * segname;
address pbn;
```

This call enters a sysname/partition block number pair into the partition directory. **Pptr** is a pointer into the partition table identifying the partition. A buffer is selected, the sysname is hashed, and the appropriate bucket from the directory is read into the buffer. **P_Enter** attempts to place the entry in this bucket. Collision handling is a simple sequential scheme that first searches for an empty entry in this bucket and then, if no entry is found, **P_Enter** examines the next bucket. This requires another partition read to the buffer. Once the buffer has been updated correctly, the buffer contents are written to both copies of the directory bucket. Because of the C structure used, there is a good bit of wasted space due to alignment restrictions. Later implementations may make changes to the structure to eliminate this waste. The return value of the operation indicates the index of the directory bucket into which the entry was placed if the operation was successful; otherwise, the operation returns failure.

3.2.4 Directory Management—Removing Entries

```
int     P_Remove (pptr, segname)
        address pptr;
SYSNAME * segname;
```

P_Remove is the complement of **P_Enter**. It uses the same hashing and collision scheme to remove an entry in the partition directory. The return value of the function indicates success or

failure.

3.2.5 Directory Management—Modifying Entries

```
int    P_Mod (pptr, segname, hdroffset)
       address pptr;
SYSNAME * segname;
address hdroffset;
```

P_Mod allows the modification of an existing directory entry. The same hashing and collision strategies used in **P_Enter** and **P_Remove** are also used in this operation. The return value of **P_Mod** is the same as that for **P_Enter**.

3.2.6 Directory Management—Locating Entries

```
int    P_Find (pptr, segname)
       address pptr;
SYSNAME * segname;
```

P_Find locates the entry in the directory for the segname passed. The same hashing and collision scheme is used as in **P_Enter** and **P_Remove**. The return value of the function is the partition offset of the segment, if it resides on this partition, and is failure otherwise.

3.2.7 Directory Management—Examining Entries

```
int    P_GetFirst (pptr, number, segarray)
       address pptr;
int    number;
SYSNAME * segarray;
```

The first **number** segment sysnames found in the directory are returned in the array pointed to by **segarray**. The array is provided by the caller. A global variable **DirIndex** is set to zero and then the operation **P_GetNext** is called to perform the actual work. **Pptr** determines the partition to use. The return value indicates the number of sysnames actually returned.

3.2.8 Directory Management—Examining Entries, Part II

```
int    P_GetNext (pptr, number, segarray)
       address pptr;
int    number;
SYSNAME * segarray;
```

The next **number** segnames in the directory are placed in the array pointed to by **segarray**. The array is provided by the caller. As in **P_GetFirst**, **pptr** determines the partition to use. **DirIndex** determines where in the partition directory to start collecting names. Directory is processed bucket by bucket until the required number of sysnames are collected. Less than **number** names may be collected if the operation runs out of directory entries. A read lock on the directory is required. One of the directory buffers is used to read the directory buckets for processing. The return value indicates the number of sysname actually collected.

3.2.9 Available Partition Storage

```
u_int P_AvailableSpace (pptr)
address pptr;
```

The amount of free space on the partition is returned. This value may not be entirely accurate.

3.2.10 Partition Reads

```
P_Read (pptr, pbn, addr)
address pptr;
address pbn, addr;
```

A block of the partition referred to by **pptr** is copied into memory at **addr**. **Pbn** is the block to be read. **Addr** should contain a physical address. **P_Read** calls its device read operation to perform the request. Prior to this call, the value in **pbn** must be converted from a PBN to a DBN for the device, using the partition base address found in the APD for the partition. Success or failure is return as the value of the function.

3.2.11 Partition Writes

```
P_Write (pptr, pbn, addr, id, flag)
address pptr;
address pbn, addr;
SYSNAME *id;
u_short force;
```

A page from memory is written to the partition referred to by **pptr**. **Pbn** is the PBN of the destination and **addr** is the physical address of the source. As in **P_Read**, the value in **pbn** is converted from a PBN to a DBN for the device before calling the device write operation. **Id** and **flag** are parameters that the device write uses to control the type of write performed, and are simply passed uninterpreted to the device write operation. The **flag** indicates whether the write is asynchronous or synchronous and whether the write is performed by an action. The return value for the operation is either success or failure.

3.2.12 Storage Allocation

```
u_int P_GetBlk (pptr, number, parray)
address pptr;
u_int number, *parray;
```

P_GetBlk0 allocate blocks of storage from the partition pptr. Number specifies how many blocks are required. Parray is a pointer to an array where the block numbers of the allocated blocks are place to pass to the caller. Parray is provided by the caller. The blocks contained in parray at the end of the operation are not necessarily contiguous. The operation uses the allocation map semaphores to ensure mutual exclusion on the map, but also uses an inuse flag to avoid waiting if possible. The return value is the number of blocks allocated.

3.2.13 Deallocation of Storage

```
u_int P_ReturnBlk (pptr, number, parray)
address pptr;
u_int number, *parray;
```

This call deallocates storage blocks to the partition. **Parray** contains the PBNs of the blocks to be released. **Number** indicates the length of the array. The operation takes the semaphore of each allocation that contains an entry it must reset. This list should be sorted in ascending order for efficiency, but this is not required. The return value of **P_ReturnBlk** indicates the success or

failure of the operation.

3.2.14 Partition Recovery Support

```
int   P_Flush (actname)
      SYSNAME actname;
```

The **P_Flush** operation allows recovery management to ensure that action write requests are completed on time. The **actname** parameter specifies the action causing the flush. A partition flush table is maintained for all partitions at the site. **P_Flush** locates the entry for the given action and calls the appropriate device flush operation for each partition referenced in the entry. The value returned by **P_Flush** indicates the success or failure of the operation.

3.2.15 Partition Reconstruction

```
u_int P_Restore (pptr)
address pptr;
```

The partition refered to by **pptr** is activated. This includes:

1. a read/write lock for the directory is created, along with a set of semaphores for the allocation map. A buffer for the directory is also created;
2. the operation performs a consistency check on the partition header and directory. The two copies of the partition header are read and compared. The primary copy is used to update the secondary if there is a disagreement. If neither copy can be read, the operation returns a value indicating failure. The same procedure is followed for each bucket of the partition directory;
3. the reconstruction of the partition allocation map. As in **P_Create**, the storage for the partition headers and directories is preallocated. Also, any excess bits at the end of the allocation map are set to prevent their allocation. The other phase of reconstruction involves the examination of each segment on the partition to determine which blocks are allocated for the segments. P_Restore reads the directory and for each sysname encountered, it makes a call to S_Check (described in the previous section dealing with the segment module). S_Check returns a list of the partition blocks in use. P_Restore allocates these blocks; and
4. any action processing remaining to be performed is done. This is actually done through the call to S_Check for each segment. S_Check determines whether or not further action processing is required. If S_Check indicates that a segment is shadowed, then the segment sysname is placed in a table for further processing. For each sysname in this table, P_Restore determines which action caused the shadowed by examining the shadow field of the in-memory version of the segment descriptor. The kernel database is then queried to find the final result of this action (whether it committed or aborted). If the action is found to have committed, P_Restore calls **S_Eoa** with the **flag** parameter set for a commit. If the database indicates that the action aborted, **S_Eoa** is called with its **flag** set for an abort. If the database contains to information for the action, the segment sysname is saved for further processing by the action management system, which has more complete information concerning action events.

The return value for the operation indicates the success or failure of the operation.

3.2.16 Maybe Table Manipulations

```
u_int P_MayEnter(segname)
SYSNAME segname

u_int P_MayTest(segname)
SYSNAME segname
```

These two operations provide the interface to the maybe table. **P_MayEnter** enters the sysname specified into the maybe table. **P_MayTest** queries the maybe table to determine whether the specified sysname is in the maybe table. The implementation is based on the hashing technique discussed in this dissertation. The sysname is hashed to a compact format and enter into the maybe table using a second hashing function. The return value for both operations indicates success or failure. A successful return value from **P_MayTest** indicates only that the sysname is probably contained in the maybe table. A return value of "failure" indicates that the sysname is definitely not in the maybe table.

4. The Clouds Device System

This section presents the interface and structures in the device module for the RL02 disk. The structures and operations defined are found mainly in the files rl_dev.c and rl_dev.c. The definition for RL02 requests is found in buffer.h.

4.1 Device Module Structures

This section describes the major structures used by the RL02 device module. For the most part these structures are defined in rl_dev.c. However, the request packet definition can be found in buffer.h and the active device descriptor definition can be found in devtab.h.

4.1.1 The Medium Header

```
typedef struct rl_header
{
        u_int signature;
        SYSNAME medname, devname;
        u_int storage;
        u_short npart;

        struct rl_index {
                SYSNAME partname;
                u_int start;
                u_int extent;
                u_short type;
        } index[MAX_PART];

        char    filler[406];
} Header;
```

This structure definition is found is the file rl_dev.c. It defines the medium header for a RL02 device. The structure is zero padded to a block size. Most of the information has been described previously. Start, storage, and extent are in terms of device blocks (512 bytes). The **signature** field is currently unused.

4.1.2 RL02 Control Registers

```
static struct rl_regs {
        u_short cs;
        u_short ba;
        union {
                u_short seek;
                u_short rw;
                u_short get_st;
        } da;
        union {
                u_short get_st;
                u_short rhead;
                u_short rw;
        } mp;
} *rl_regs;
```

Rl_regs is a pointer to the control and status registers used by the RL02. The location of the register set for the RL02 is determined by the **RL_Init** and **RL_Mount** operations using the offset specified by the device documentation [DEC82b] and the base address for the device control and register area passed to these operations. **Rl_regs** contains an address inside the device control and status area of the kernel memory. The registers are all 16 bit words. **Cs** is the control register and is used to specify the type of device operation to perform, and also allows the specification of options. Return codes and error codes are passed from the controller and device through this register. The **ba** register indicates the base address in memory for a data transfer. The address is actually a Unibus address. The address is obtained as described in the description of the **rwstart** operation. The **da** register has several functions and format depending on the device operation being performed. For a "get-device-status" operation, the register controls whether a reset is performed. For seek operations, the **da** register indicates the distance and direction in which to seek. For data transfer operations, the base of the area involved in the transfer is specified in the **da** register in cylinder/sector format. The **mp** register is a multipurpose register used for counting the amount of data transferred in read and write operations and as a fault status register during the "get-device-status" operation. The union of various formats for the **da** and **mp** registers was taken from the Unix RL02 driver.

4.1.3 Request Packets

```
typedef struct {
        QE      links;
        QE      thread;
        SYSNAME id;
        address  vma, da;
        u_short reqtype, errcnt;
} buffer;
```

This type defines the structure of a request packet. The structure has linkage fields for two hardware queues. One is for the request queue. The other is there in the event the request is started by an action and the packet is placed in a flush table entry. **Vma** contains a physical address to or from which the data is to be transferred. **Reqtype** is used to encode the operation and write options. **Errcnt** is the number of errors caused by the request.

4.1.4 Bad Sector Table Definition

```
typedef struct badsct {
        u_int csn, filler1;

        struct fields {
                u_int cyl: 9;
                u_int filler1: 7;
                u_int sect: 6;
                u_int filler2: 2;
                u_int hd: 1;
                u_int filler3: 7;
        } bds[125];

        u_int filler2;
} BstEntry;
```

BstEntry is a type for an in-memory version of the bad sector tables residing on the last track of the RL02. Each element of the **bds** array indicates a bad sector. The backup sector is found on the last cylinder of the RL02. The index into the **bds** array is also the index of the backup sector. The format of the structure matches the format of the bad sector file on the RL02 media, so filler fields are used in the structure.

4.1.5 RL02 Flush Table Structures

```
typedef struct flush
{
        QE      links;
        QH      flush_set;
        SYSNAME         id;
        u_short  complete, outstanding, errcnt, fishflag;
} flentry;

static QH flush[FLSHTBSZ];
```

These are the definitions of the flush table for the RL02 device. Entries for the table are allocated from the system heap. **Flush** is the flush table itself. As mentioned previously, it is a hash table. The declaration is static to prevent the table's use outside the module.

4.1.6 The Ready Queue

```
static  QH * request;
```

The request queue for the RL02 is implemented using the VAX hardware queue facility. Again, the static declaration hides the structure.

4.1.7 The Active Device Table

```
typedef struct {
        QE      links;
        SYSNAME         devname, medname;
        u_short  active, errcnt, available;
        u_short  nintr, nreads, nwrites, nerrs;
        GENERIC         *regs;
        void     (*ivector) ();
        u_int    (*enter) ();
        u_int    (*remove) ();
        u_int    (*partitions) ();
        u_int    (*read) ();
        u_int    (*write) ();
        void     (*dispatch) ();
        u_int    (*flush) ();
        u_int    (*init) ();
        u_int    (*mount) ();
        u_int    (*unmount) ();
} ADD
```

The active device descriptor format is defined above. The GENERIC type is a 32 bit structure which is a union of various types.

4.2 The Device Module Interface

The following functions are the operations available through the device module for the RL02.

4.2.1 Device Initialization

```
u_int RL_Init (devname, medname, csroffset)
SYSNAME *devname, *medname;
address csroffset;
```

The third parameter, **csroffset**, contains the base address of the device control and status register area for the kernel. **RL_Init** uses this address to locate the control/status registers for the RL02. These registers are used to initiate device commands. The operation then issues the first device operation to test whether the device is present and ready. If this is so, the operation may continue. The **RL_Init** operation is basically a formatting operation. The call creates the medium header and initializes this structure. It then mounts the device by creating an active device descriptor in the active device table. See **RL_Mount** for more details on device mounting. The sysnames for the device and its medium are created by the operation and returned to the caller. The function value indicates the success or failure of the call.

4.2.2 Device Storage Allocation

```
u_int    RL_Enter(partname, size, type)
SYSNAME         partname;
u_int    size;
u_short  type;
```

RL_Enter provides a mechanism for allocating device storage for partitions. The structures and strategies used are very simplistic. Neither the call nor any other part of the device module attempts to perform block coalescing. A simple allocate at the end strategy is used to allocate storage for a partition. The index field of the medium header is used to manage this simple form of allocation. The parameters passed into the call describe a partition being created. The second parameter, **size**, is used to allocate the correct amount of the storage. All three parameters are placed into the next free index field entry, along with the base address for the new partition. The medium header is then written to disk. The return value of the function indicates the success or failure of the operation. In the case of a successful allocation, the starting address is returned. Failure is indicated by a zero return value.

4.2.3 Device Storage Deallocation

```
u_int    RL_Remove(partname)
SYSNAME      partname;
```

RL_Remove is the complement of **RL_Enter** The entry in the medium table for the referenced partition is cleared. No storage compaction is attempted. Currently, there is no facility for the free storage to be reallocated. The return value indicates success or failure.

4.2.4 Device Allocation Query

```
u_int    RL_Partitions(partarray, offset, size)
SYSNAME      *partarray;
address *offset;
u_int *size;
```

This operation allows the caller to determine what partitions reside on an RL02 device. Each of the parameters is an array. This call is generally used during system startup as part of storage management reconstruction. The return value indicates the number of partitions that reside on the device.

4.2.5 Device Activation

```
u_int RL_Mount (devname, medname, csroffset)
SYSNAME *devname, *medname;
u_int csroffset;
```

RL_Mount activates an RL02 device. The third parameter, **csroffset**, contains the base address of the device control and status register area for the kernel. **RL_Mount** uses this address to locate the control/status registers for the RL02. The operation then issues the first device operation to test whether the device is present and ready. If this is so, the operation may continue. The call creates an active device descriptor for the RL02, places it in the ADT, and then proceeds to initialize the descriptor from the medium header found on the device. The sysnames found in this header must match those passed as parameters to the call. **RL_Mount** also examines the bad sector file (this resides on the last sector of RL02 media) and places this information into the appropriate tables. The call also initializes the request queue and flush table to an empty state. The addresses for the operations described in this section are placed in the device descriptor, and most subsequent references to the operations are made through the device descriptor fields. One of the last acts of **RL_Mount** is to set the available flag in the ADD, so the device is available to the kernel. The value of the function indicates the success or failure of the operation.

4.2.6 Device Deactivation

```
u_int RL_UnMount (devname, medname)
SYSNAME * devname, *medname;
```

The sysnames passed into the operation are compared with those contained in the medium header, and if they match, the device is unmounted. First, the availability flag in the ADD is cleared so that no further requests are accepted. Pending device requests are flushed from the device request queue. The ADD for the RL02 is destroyed. The return value indicates success or failure of the operation.

4.2.7 Device Read Requests

```
u_int RL_Read (addr, lbn)
address addr;
address lbn;
```

RL_Read allows the caller to create a read request on the RL02 device. The call takes a request packet from the pool and fills the fields with the appropriate values. Note that the memory address into which the data is to be transferred is a physical address. A semaphore is created for the call using the request packet address as the semaphore ID. If the device is currently processing the request, the new packet is placed onto the request queue. If the request queue is empty, however, **RL_Read** initiates the read operation itself. In either case, the operation waits on the semaphore it created. When the semaphore is notified, **RL_Read** returns the request packet and destroys the semaphore. The packet returns the result of the read operation (a zero in the **errcnt** field indicates a successful transfer; any positive value means the read failed) and the appropriate value is returned as the function value.

4.2.8 Device Write Requests

```
u_int RL_Write (addr, lbn, id, flags)
address addr;
address lbn;
SYSNAME *id;
short  flags;
```

RL_Write initiates a write request to the RL02 device. As with **RL_Read** the operation takes a request packet from the pool of packets and fills the packet fields with the necessary information. The **flags** parameter is used to control whether a write operation is synchronous or asynchronous. It also indicates whether an action is performing this request. The bits in **flags** are ORed with the operation code for a write request and placed in the **reqtype** field of the request packet. The **id** parameter is placed in the **id** field of the packet if **flags** indicates the write is for an action; the **id** is the action's sysname. A semaphore is created for a synchronous write. If the write is for an action, the request packet is queued on an entry in the RL02 flush table. If no entry exists for the action, one is created. As with **RL_Read** the request either is placed in the request queue if it is non-empty, or started by **RL_Write** if the request is empty. For asynchronous writes, the operation returns immediately. For synchronous writes, **RL_Write** waits on the semaphore as does **RL_Read** and, on the semaphore notify, performs the same cleanup as **RL_Read** The return value for the function is either success or failure.

4.2.9 Flushing Action Writes

```
u_int RL_Flush (id)
SYSNAME *id;
```

Through the **RL_Flush** operation, the caller can tell the RL02 device to notify the caller when all write request performed by an action are complete. The operation first determines if there is a flush table entry for the action. If so, the outstanding and completed field of the table entry are compared and if the two fields are not equal (indicating that some pending requests started by the action have not completed), the **flshflag** field is set, and the call waits on the semaphore created for the table entry. When this semaphore is notified, **RL_Flush** destroys the semaphore, the flush table entry, and returns the number of requests that were completed.

4.2.10 The Device Interrupt Handler

```
void RL_Dispatch ()
```

This operation is not available as a callable operation. It is the interrupt handler for the RL02 device. An RL02 interrupt indicates that the request completed or that there was a device error. In the latter case, **RL_Dispatch** logs the error and restarts the request. A count is kept for the

request and when 15 retries have been made, the operation terminates the request. If the status registers indicate that the errors may be due to a bad sector, bad sector forwarding is attempted and the request is restarted with the new device address.

After a successful request, the operation performs the appropriate cleanup procedures for the request. For device writes, the operation must check whether the write is synchronous or asynchronous, and whether the write was started by an action. For writes started by an action there are several possibilities. After the flush table entry for the action is found, **RL_Dispatch** increments the **completed** field of the table entry. If the **flshflag** field of the entry is set and if the **outstanding** and **completed** fields are equal, the operation notifies the flush table entry semaphore. For asynchronous writes, the operation returns the request packet used for the request. For asynchronous write requests and for read requests, **RL_Dispatch** notifies the semaphore associated with the request packet.

After processing the just completed request, **RL_Dispatch** selects a new request to start. This is done in a first-come-first-serve manner. If the request queue is empty, the **active** flag found in the ADD for the RL02 is cleared, indicating that there are no pending requests. Otherwise, the next packet is dequeued from the request queue and started by the operation.

4.2.11 Debugging Support

```
void RL_Debug()
```

This routine prints the values of the RL02 registers and structures on the console.

4.3 Important Service Utilities

This section describes several routines which, while not part of the device interface, perform important functions.

4.3.1 Initiating Device Requests

```
static u_short rwstart (bf, interrupt)
buffer * bf;
u_short interrupt;
```

Rwstart is used to start requests. The parameters are a pointer to a request packet and a flag indicating whether the request should post an interrupt when it completes. The first step **rwstart** takes is to determine whether a seek to a new cylinder is necessary. If it is, **rwstart** starts the seek and waits (using a tight loop) until the seek completes. **Rwstart** then begins setting the control and status registers to the values indicated in the request packet. The memory address to or from which data is transferred is not given directly to the device. Instead, the address in the request packet is mapped into a Unibus address. This mapping may be done in two ways. If the device is not active (the device has no pending requests), a new Unibus page mapping must be allocated. If the device is active, the Unibus page mapping from the previous request may be reused. The latter alternative is slightly faster. The disk address for the transfer must be converted from a DBN based on 512 byte blocks to a cylinder/sector format based on 256 byte blocks. The command register is set last and this initiates the request. If the **interrupt** parameter is clear (the request should not cause an interrupt), the operation waits on the completion of the request before returning. This option is seldom used and only for requests generated by the device itself for administrative purposes. If the **interrupt** parameter is set, the operation returns immediately after starting the request. The device interrupt handled by **RL_Dispatch** will take care of the request. The return value of the operation indicates success or failure.

4.3.2 Flush Table Manipulations

```
static void    fenter (id, bf)
SYSNAME id;
buffer * bf;
```

This operation enters the request packet into an existing flush table entry (identified by the parameter **id**), or creates such an entry for the action with sysname **id**, and places the request packet to which the parameter **bf** points into the new entry. In the case that a new flush table entry is created, fenter also creates a semaphore using the entry address as the semaphore ID. This semaphore is used by **RL_Flush** to wait for the completion of action requests. The operation increments the field **outstanding** (there is a new request) and enqueues the request packet onto **flush_set**. There is no return value.

5. Glossary

ADD This is an active device descriptor. It contains the information about a device which are in-use by the kernel.

ADT The ADT (active device table) is a structure used to manage descriptors for devices which are in-use by the kernel.

APD The APD (active partition descriptor) is analogous to the device descriptor. Each APD contains a pointer to the descriptor for the device on which it resides.

APT This is the active partition table. Each entry in this table is an APD for a partition being used by the kernel.

ASD ASD stands for active segment descriptor. An ASD is created for each segment which is mapped into virtual memory. The descriptor contains references to virtual memory and permanent storage mapping tables for the segment, in addition to descriptive information about the segment.

AST The AST (active segment table) contains the ASDs for segment which are being used (through an operation call on an object), or which were recently used.

Block The smallest allocatable unit of secondary storage. In the current kernel implementation the block size is 512 bytes. This is also the virtual memory page size, and the terms page and block are frequently interchanged.

DBN A DBN is a device block number. DBNs are offests from the beginning of a device. They provide a way to provide an abstract addressing scheme for all devices no matter what the underlying geometry of the device is. The device objects are responsible for performing the translation of a DBN to address format (sector/cylinder/head, for example) used by the device hardware.

Flush table A structure used at by device objects to associate writes requests being performed during an action commit to the committing action. The table allows device objects to ensure that all writes for an action are complete before the action commits.

Maybe table The maybe table is an approximate membership tester. It provides an efficient means to short circuit object searches initiated by a remote procedure call. When queried, a positive response from the maybe table indicates that the object may be at a site in the system; a more thorough search is required. A negative indicates that the object is definitely not at the site.

Page allocation map This also referred to as simply a page map. The page map is a bit map used to allocate partition storage.

Partition Partitions in the Clouds kernel are logical devices. They are composed of a contiguous collection of secondary storage blocks. Partitions must reside completely on one device. Clouds partitions are used solely to administer secondary; segment membership in a particular partition provides no logical relationship of that segment to others residing on the partition.

Partition directory Each partition maintains a directory of the segments residing on that partition. Each entry is a segment sysname and a PBN for the segment header.

PBN PBN stands for partition block number. PBNs are offsets from the beginning of a partition and are used for addressing purposes by the segment system.

Segment A Clouds segment is a a sequence of bytes which may be manipulated using the set of operations provided by the storage manager and described earlier. Segments are used by the kernel to facilitate the manipulation of Clouds objects.

Segment tree This refers to the lay out of Clouds segments on secondary storage. Each segment has a header which in addition to containing descriptive information about the segment, contains a list of refers to other blocks of storage. Each of these blocks may be a data block (a leaf in the segment tree), or a mapping block (an internal node in the segment tree) which contains references to data blocks or other mapping blocks.

Shadow pages Copies of modified pages in the permanent segment state used for recovery purposes. On commit of an action, the shadow pages are become part of the permanent state of the segment, replacing the old pages.

Shadowing This is the recovery technique used by the storage manager. Updates to recoverable segments are not made to the permanent version of the segment, but to copies of the permanent version. On commit, these copies become part of a new, modified permanent version. Shadowing in the Clouds kernel is done on at a page level.

Window Windows are used in the mapping of segments into virtual memory. They are contiguous chunk of the segment. A segment may have several windows mapped into it, each having different attributes (code windows, permanent data windows, heap storage windows, etc). Windows facilitate sharing of segments (particularly code segments) in the kernel.

6. Storage Management Functional Flow

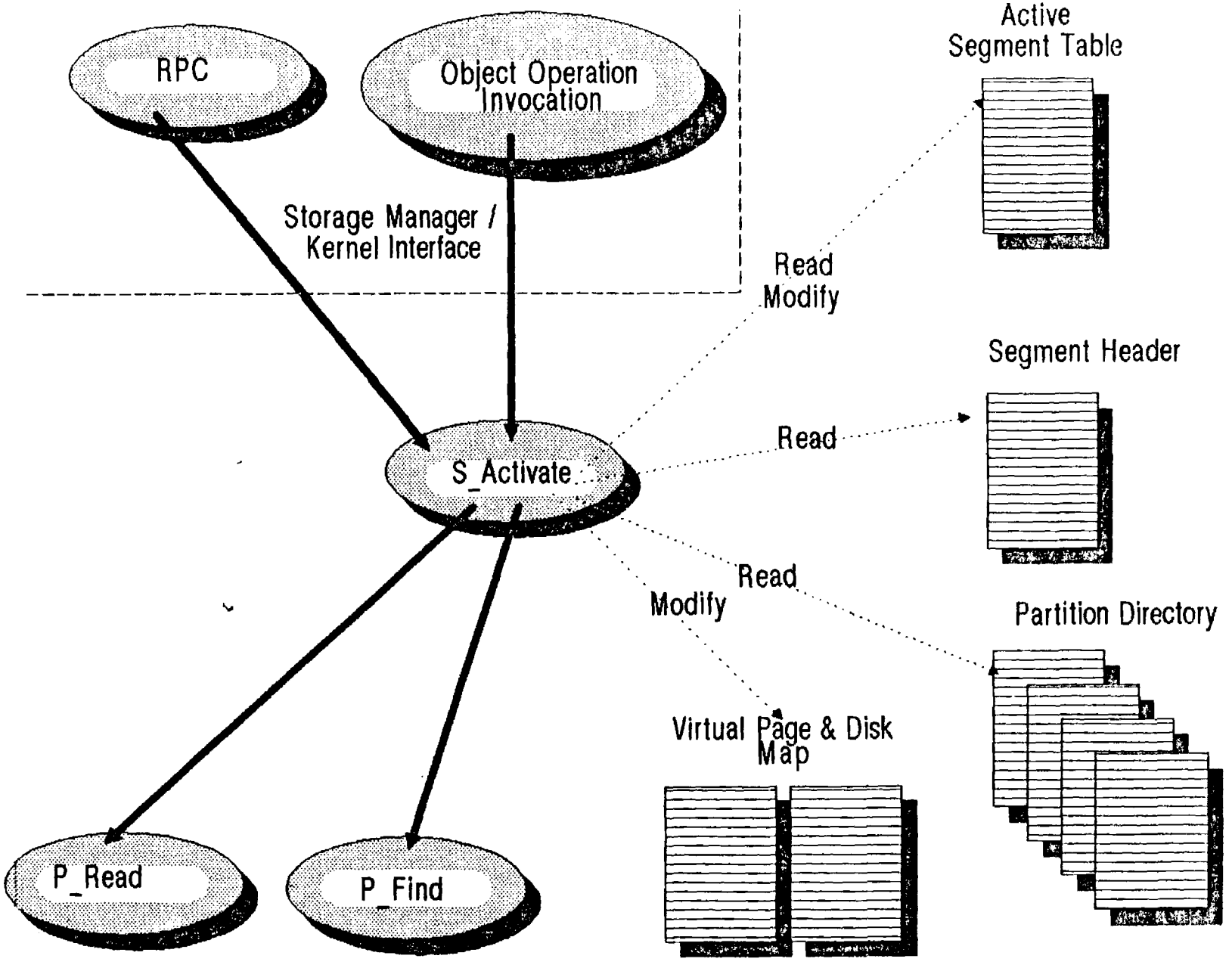
This section contains a series of diagrams illustrating the functional flow of the storage management system of Clouds. The goals of the author is to provide information as to how and by whom storage management routines are used.

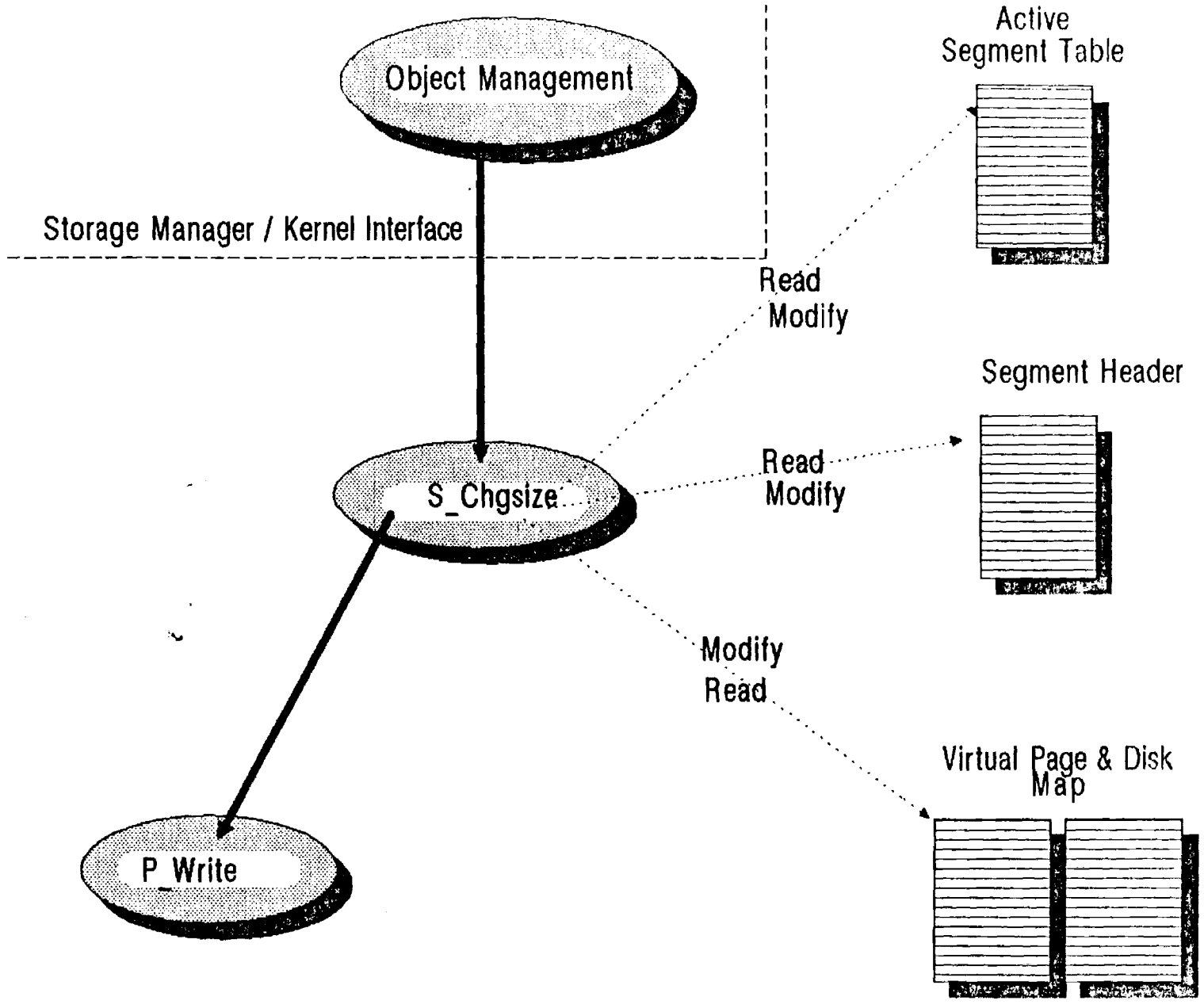
Each routine described in the earlier sections as a diagram in this section. Segment system routines appear first, followed by the partition system routines, and last, the device system routines. Within each group, the illustrations are in alphabetical order. The routine being described appears in the center of each diagram.

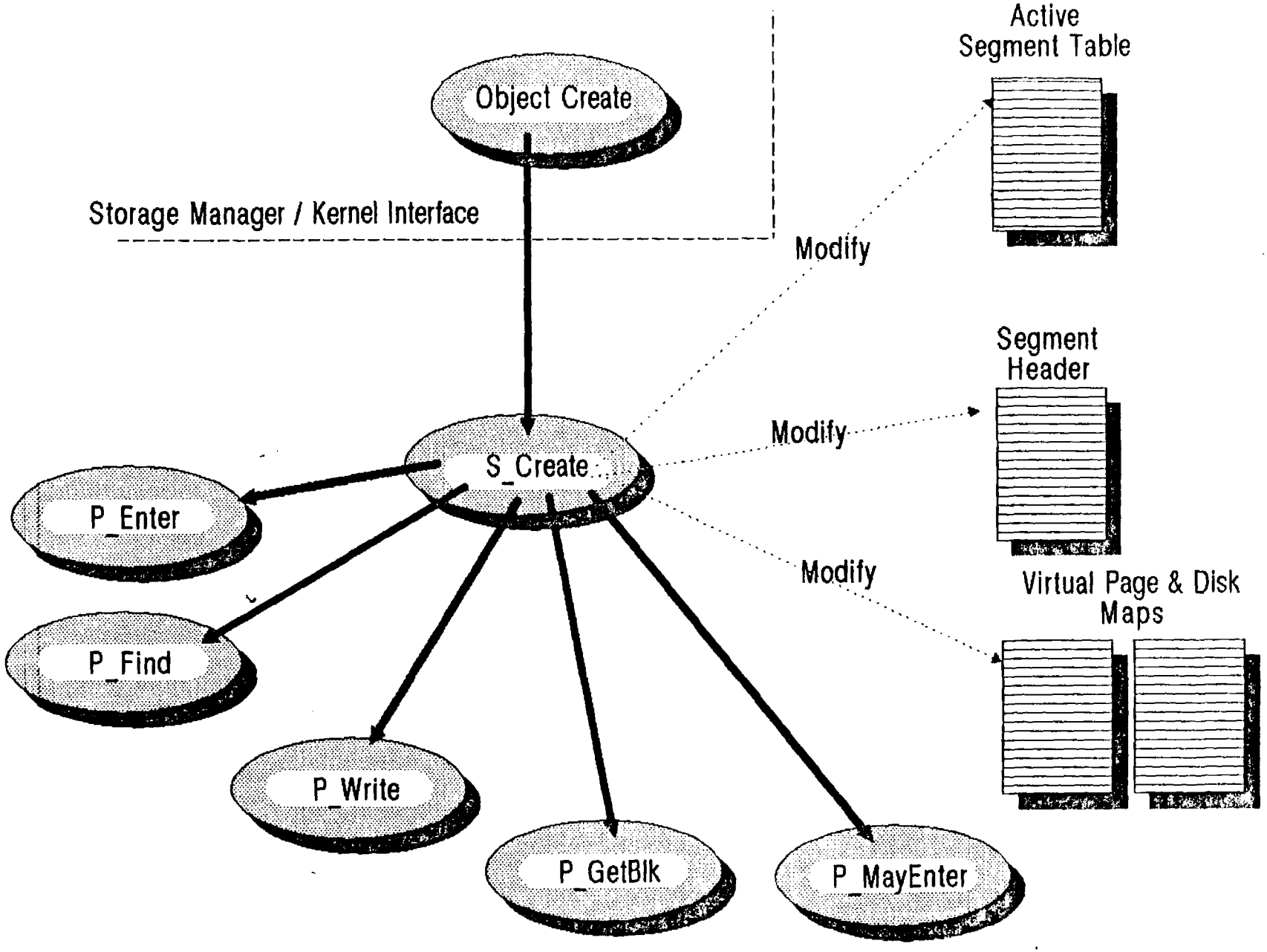
Solid arrowed lines are used to represent a caller/callee relationship. The called routine is at the arrow head and the calling routine is at the arrow tail.

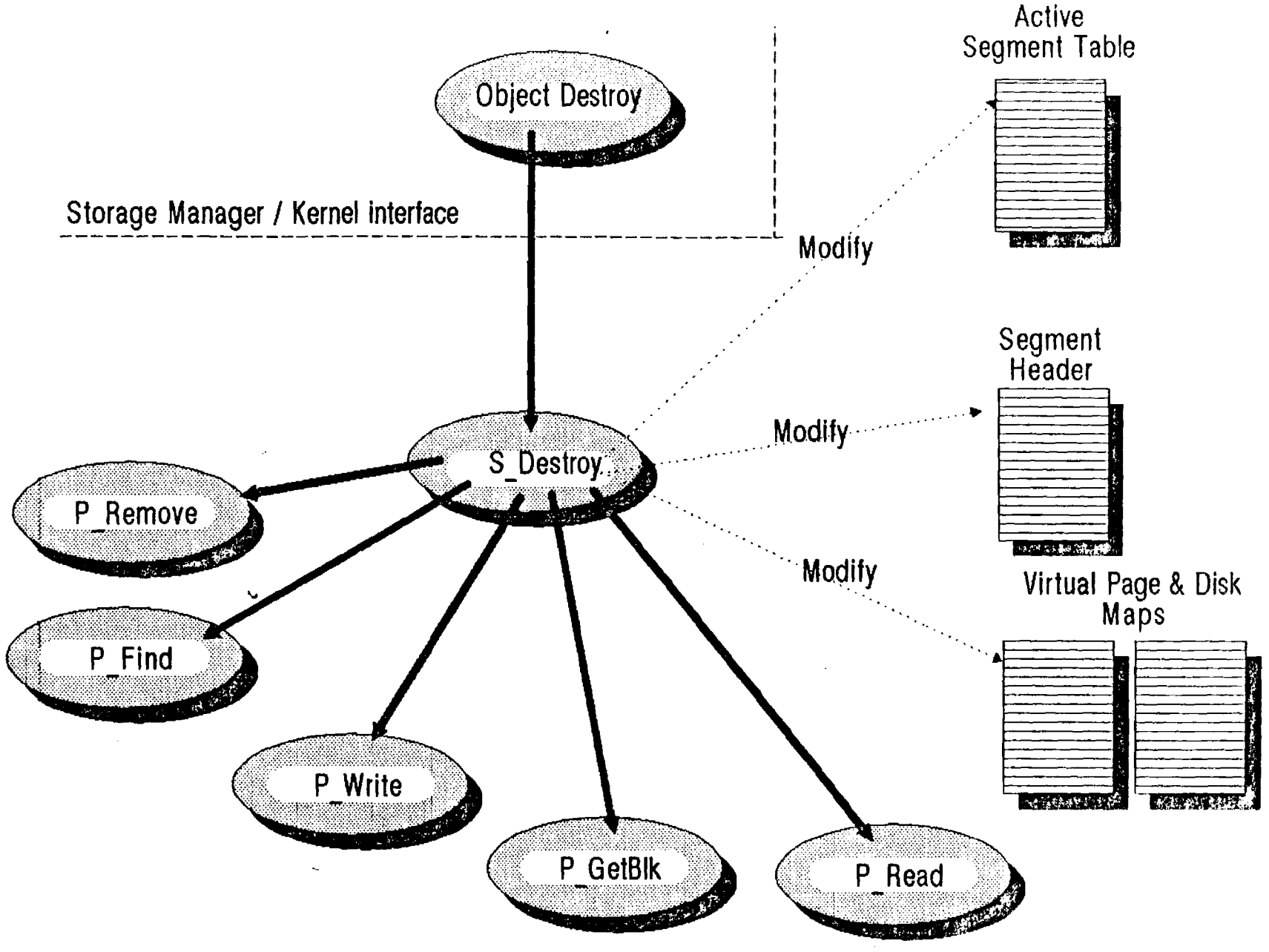
Dotted lines are used to indicate the relationship of the routine to important data structures. Each such relationship is labelled "Read" and/or "Modify", indicating whether the routine simply references the structures, updates, creates, or deletes the structure, or some combination of the above.

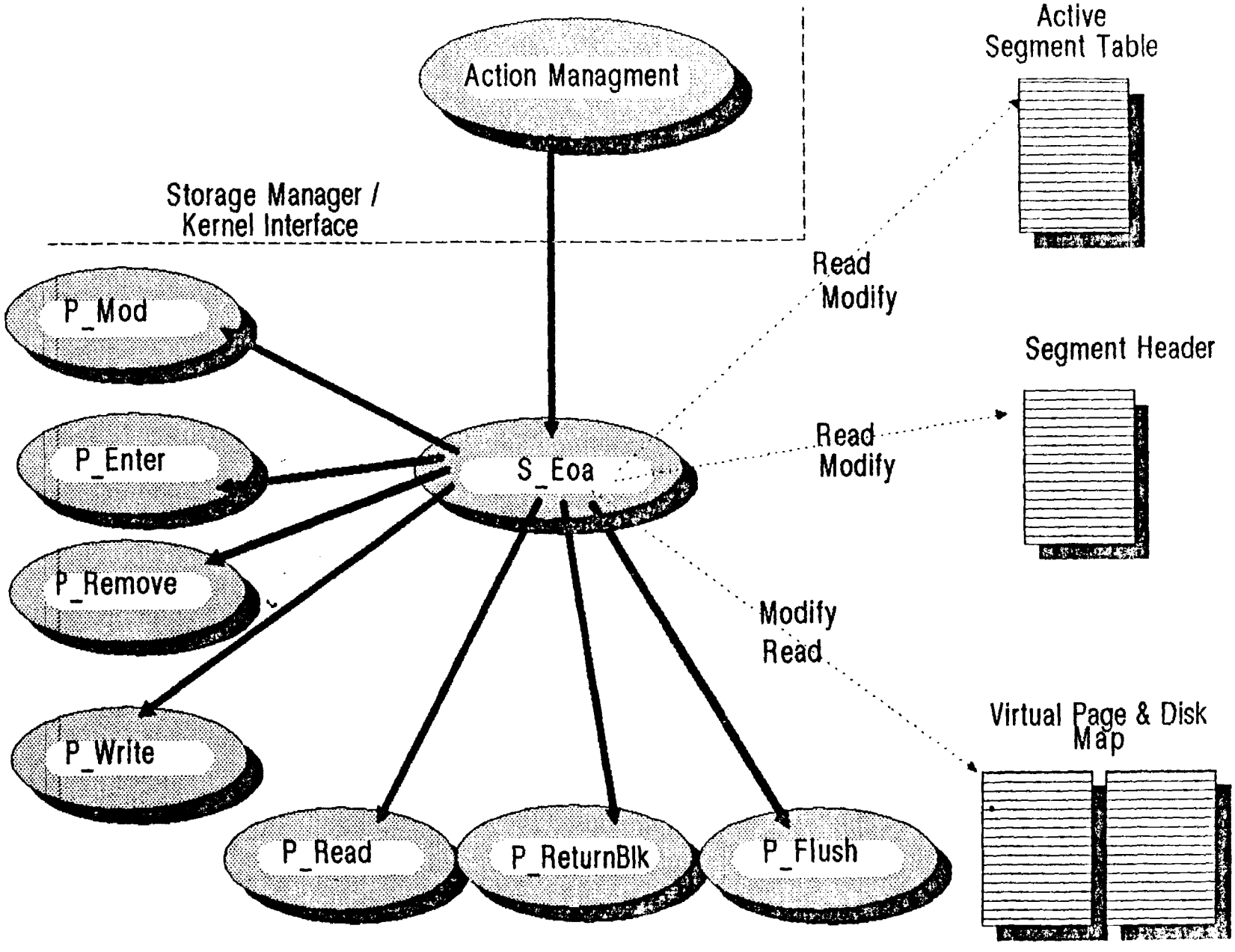
Some of the routines are part of the storage manager/kernel interface. That is, the kernel uses these routines to perform necessary manipulations on secondary storage (usually at the segment level). In these diagrams the boundary is indicated by a dashed line.

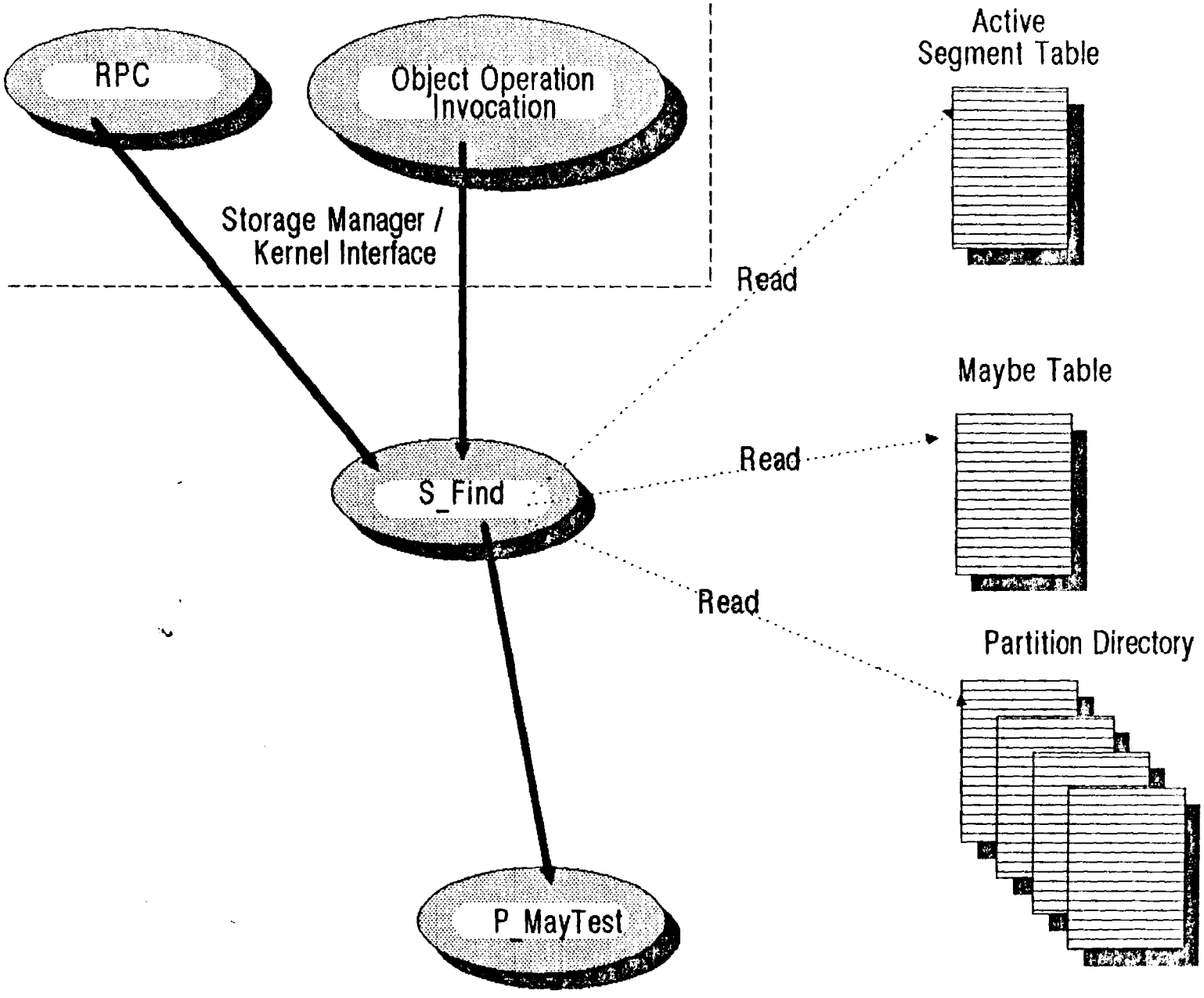

RPC
Object Operation Invocation
Storage Manager / Kernel Interface
S_Activate
P_Read
P_Find
Active Segment Table
Read
Modify
Segment Header
Read
Read
Modify
Partition Directory
Virtual Page & Disk Map

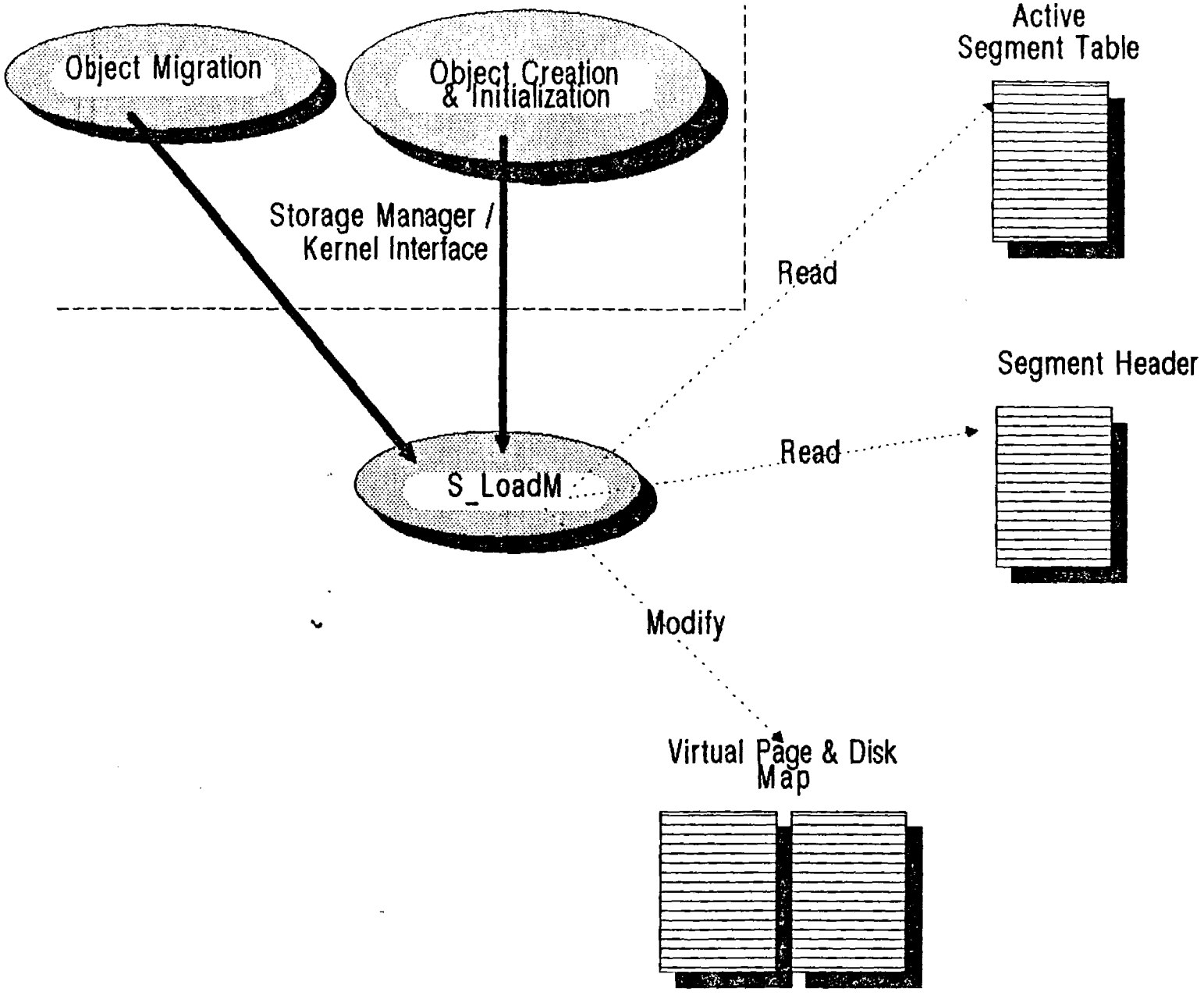

Object Management
Storage Manager / Kernel Interface
S_Chgsize
P_Write
Active
Segment Table
Read
Modify
Segment Header
Read
Modify
Modify
Read
Virtual Page & Disk
Map

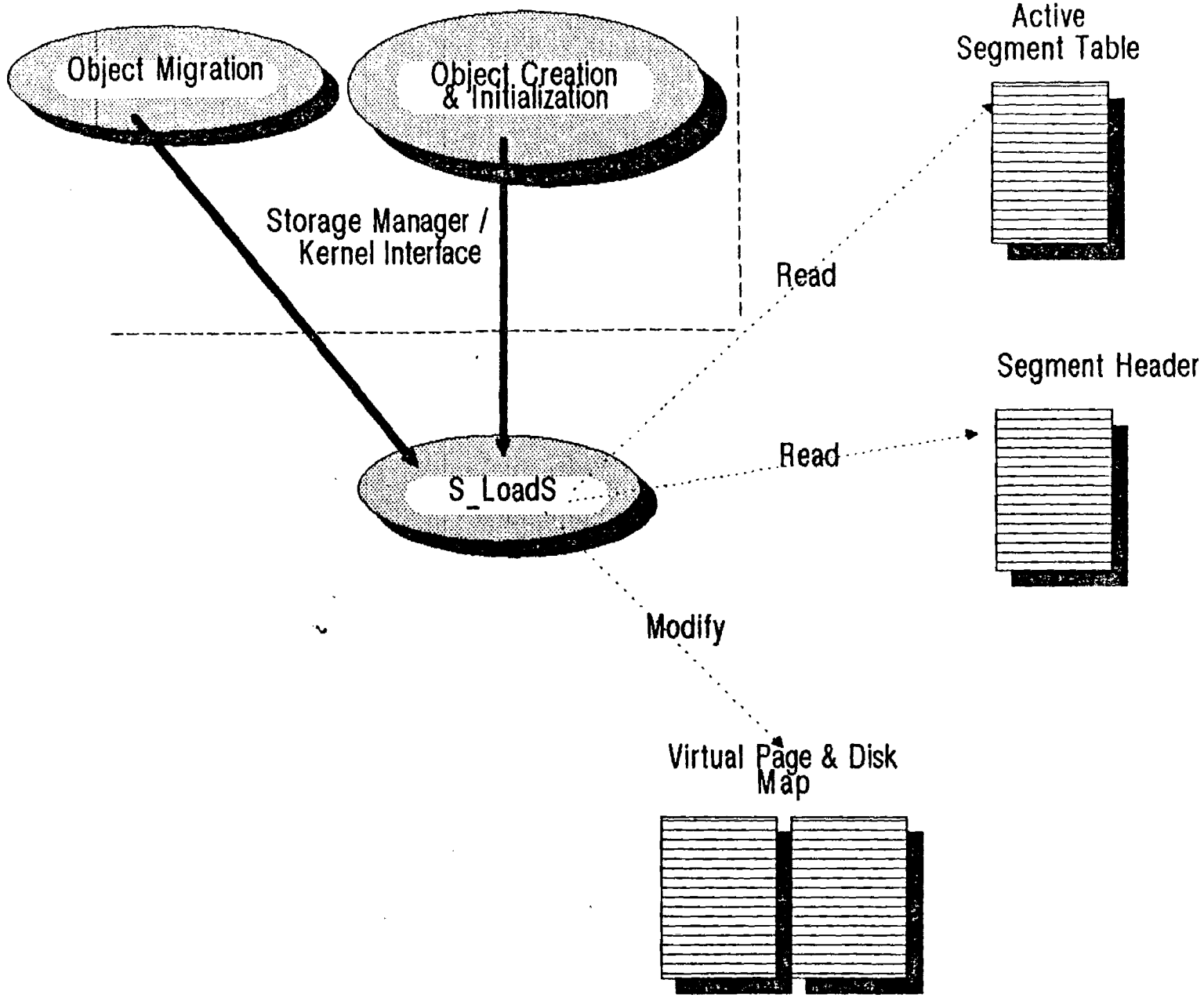

Object Create
Storage Manager / Kernel Interface
Active
Segment Table
Modify
Segment
Header
Modify
S_Create
P_Enter
P_Find
P_Write
P_GetBlk
P_MayEnter
Modify
Virtual Page & Disk
Maps

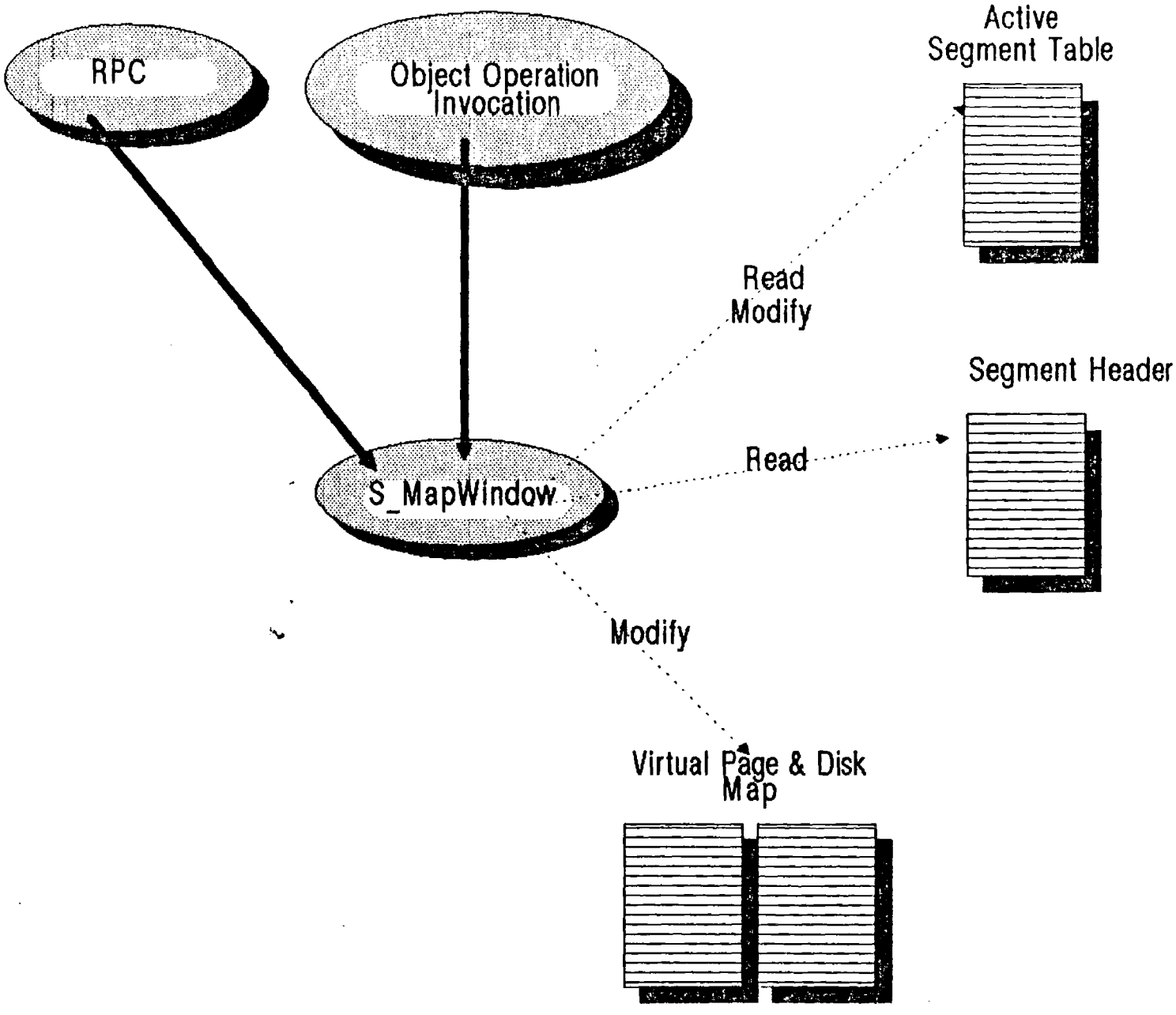

Active
Segment Table
Object Destroy
Storage Manager / Kernel interface
Modify
Segment
Header
Modify
S_Destroy
P_Remove
Modify
Virtual Page & Disk
Maps
P_Find
P_Write
P_GetBlk
P_Read

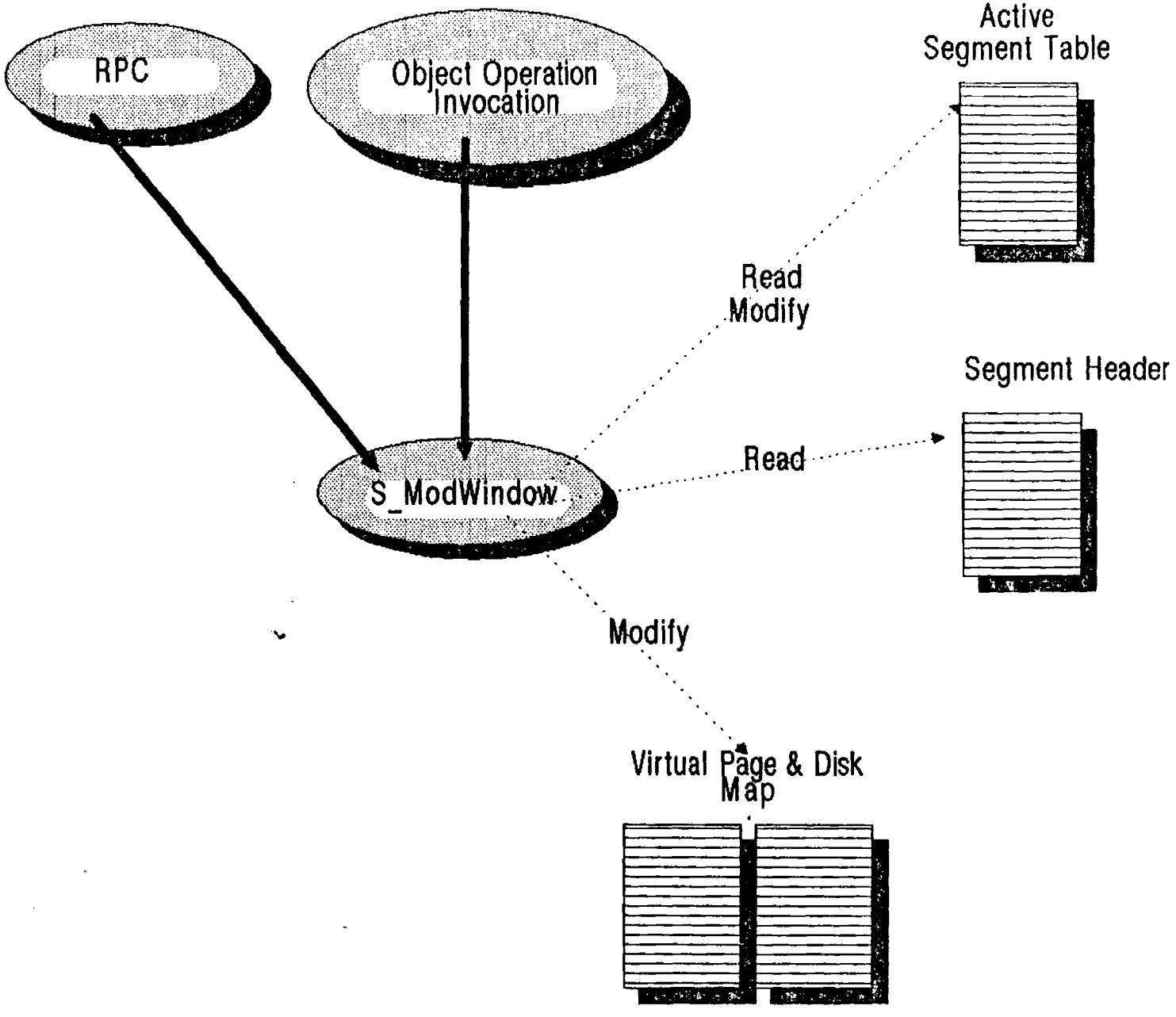

Action Managment
Storage Manager /
Kernel Interface
P_Mod
P_Enter
P_Remove
P_Write
S_Eoa
P_Read
P_ReturnBlk
P_Flush
Active
Segment Table
Read
Modify
Segment Header
Read
Modify
Modify
Read
Virtual Page & Disk
Map

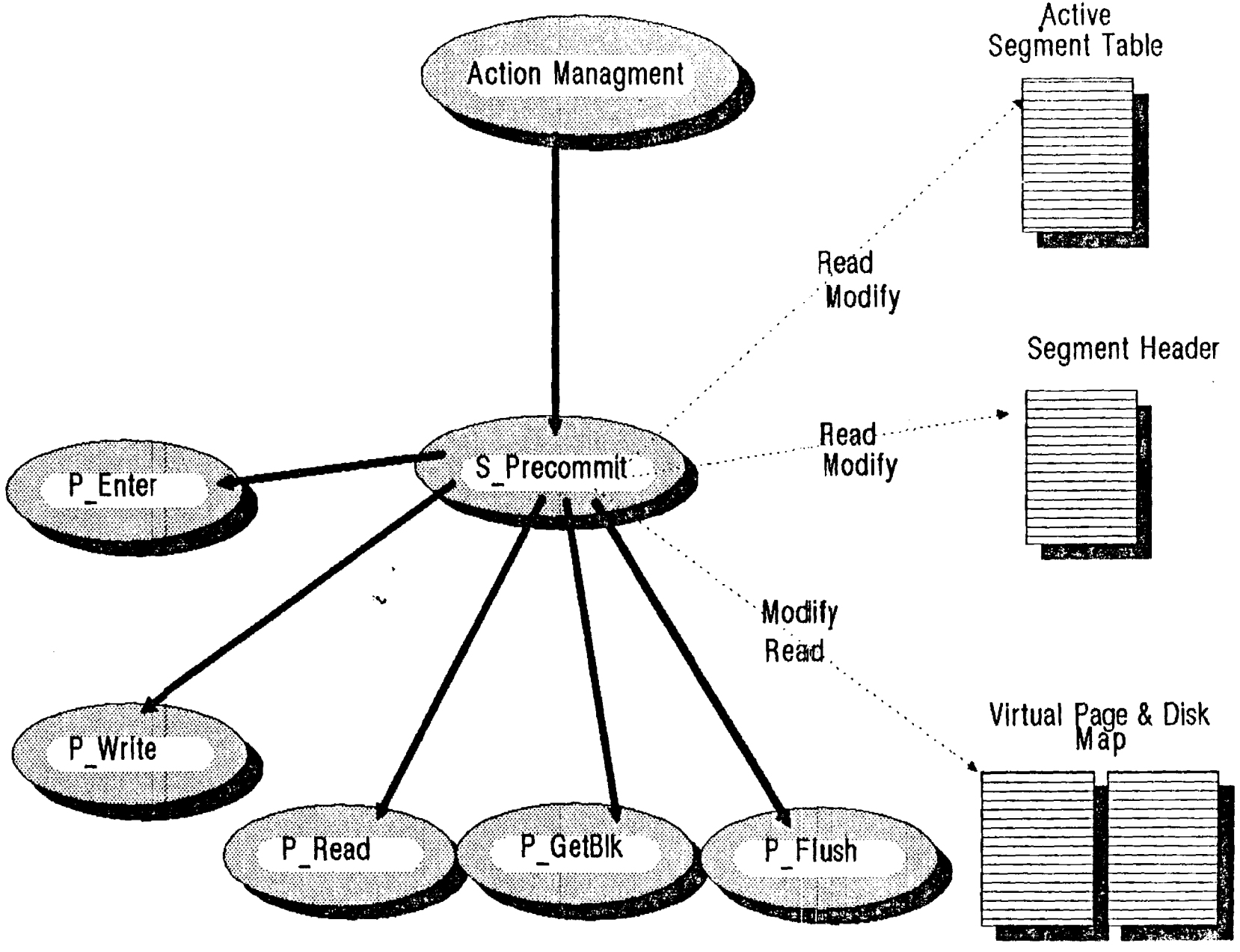

RPC
Object Operation Invocation
Storage Manager / Kernel Interface
S_Find
P_MayTest
Read
Read
Read
Active Segment Table
Maybe Table
Partition Directory

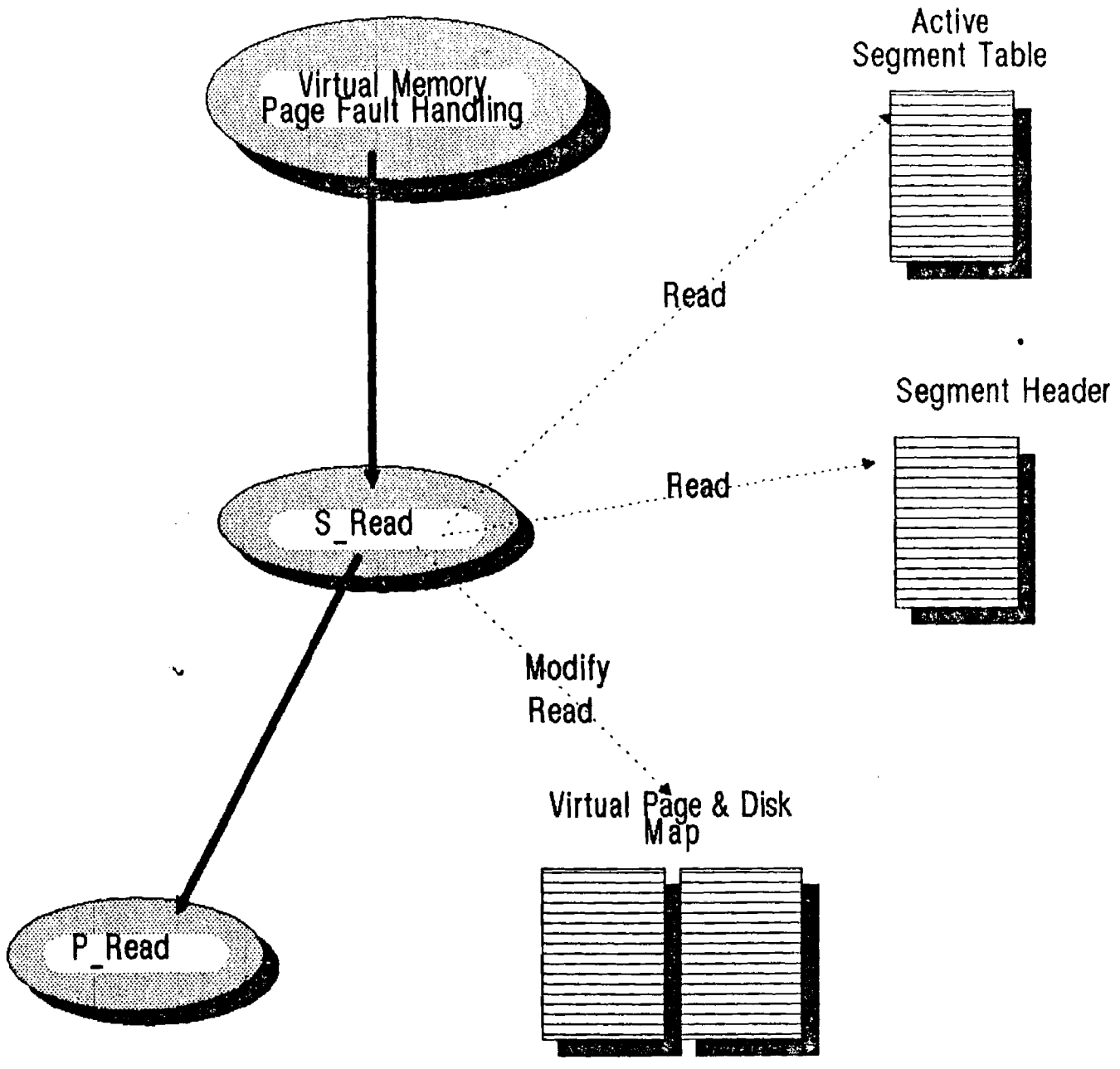

Object Migration
Object Creation & Initialization
Storage Manager / Kernel Interface
S_LoadM
Active Segment Table
Read
Segment Header
Read
Modify
Virtual Page & Disk Map

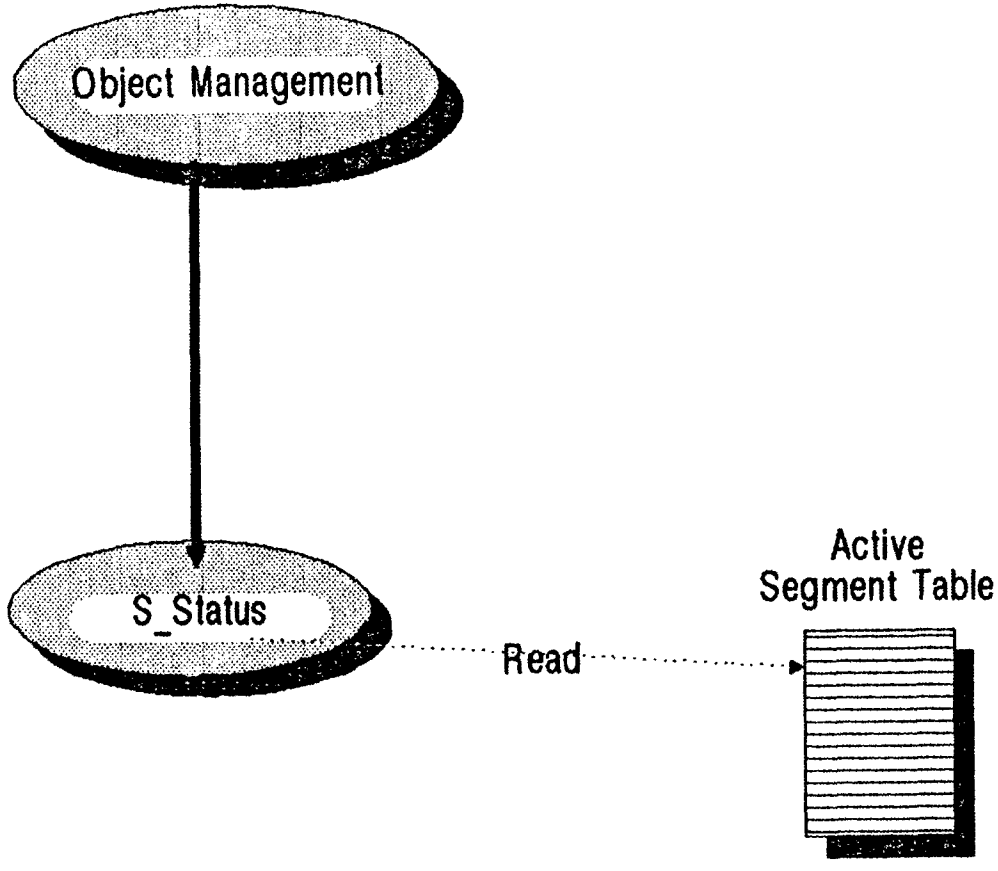

Object Migration
Object Creation & Initialization
Storage Manager / Kernel Interface
S_LoadS
Active Segment Table
Read
Segment Header
Read
Modify
Virtual Page & Disk Map

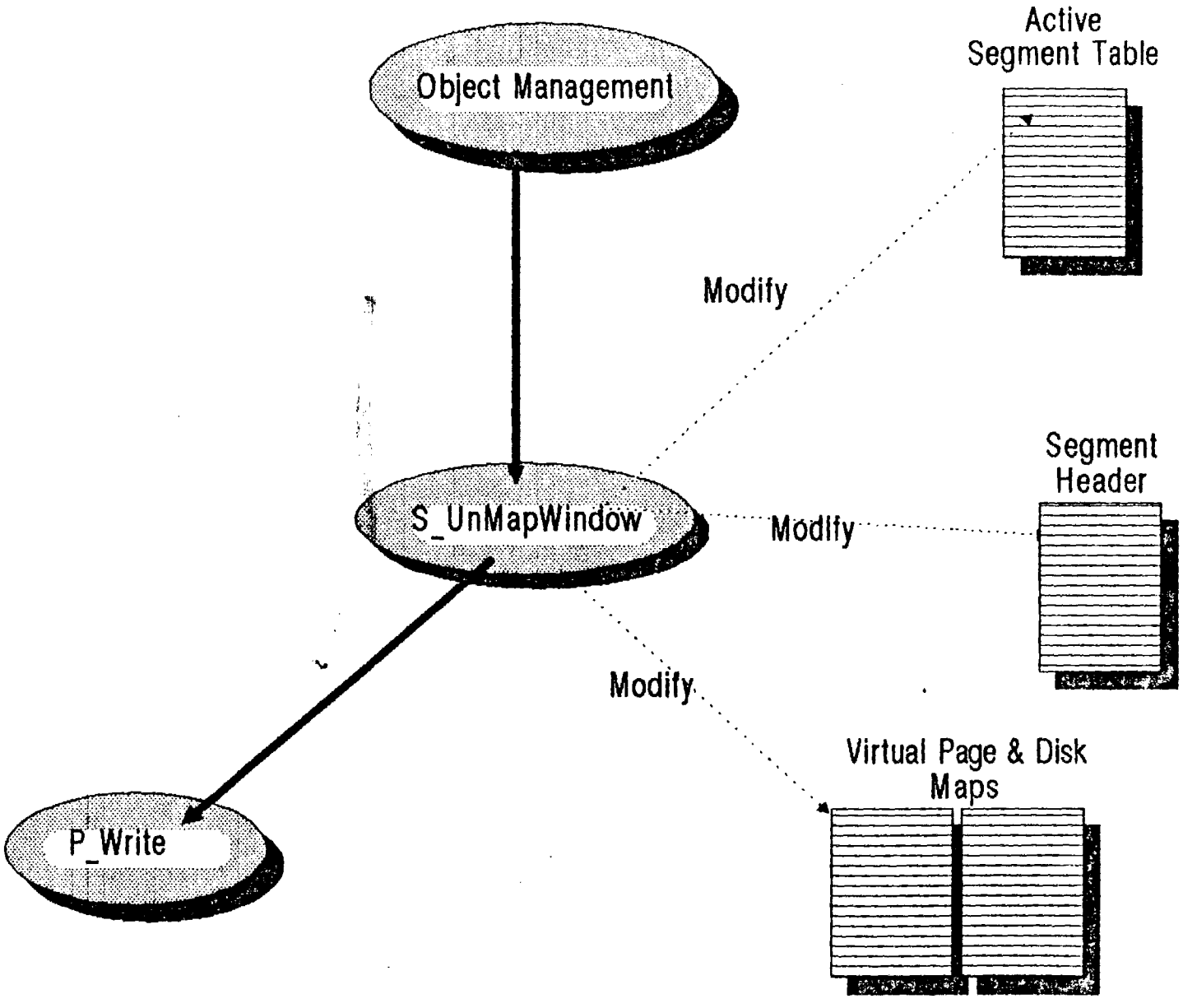

RPC
Object Operation Invocation
S_MapWindow
Active Segment Table
Read
Modify
Segment Header
Read
Modify
Virtual Page & Disk Map

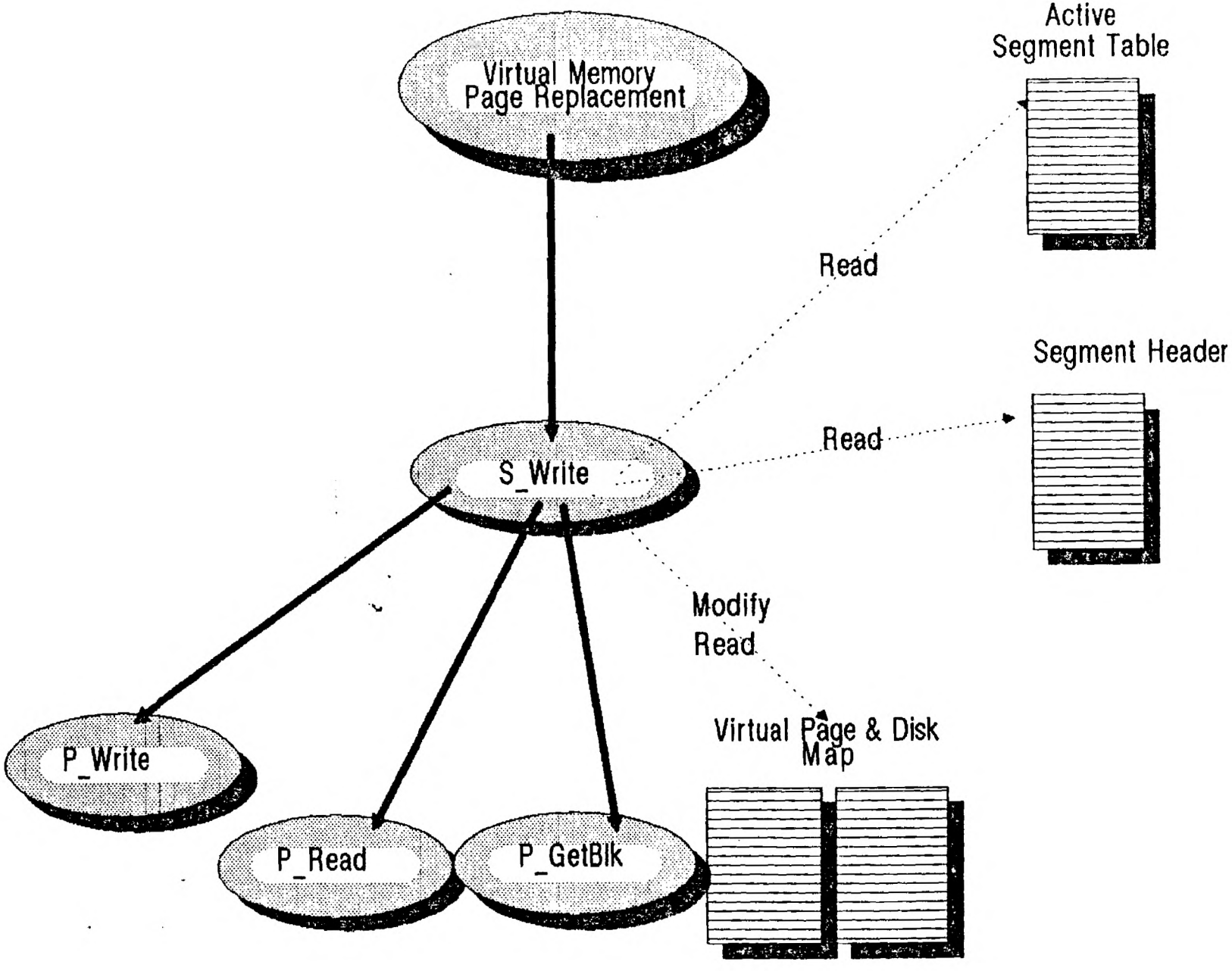

RPC
Object Operation Invocation
S_ModWindow
Active Segment Table
Read Modify
Segment Header
Read
Modify
Virtual Page & Disk Map

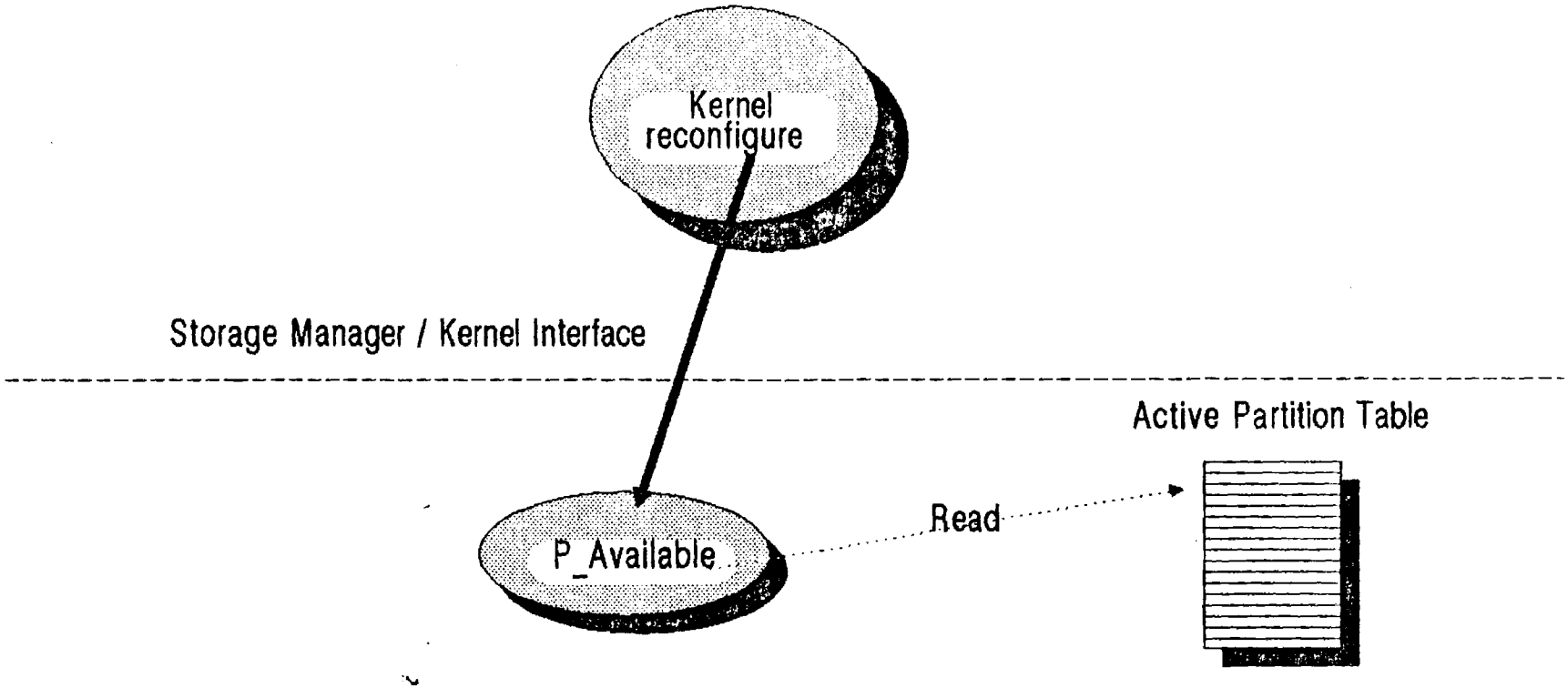

Action Managment
Active
Segment Table
Read
Modify
Segment Header
Read
Modify
S_Precommit
P_Enter
Modify
Read
Virtual Page & Disk
Map
P_Write
P_Read
P_GetBlk
P_Flush

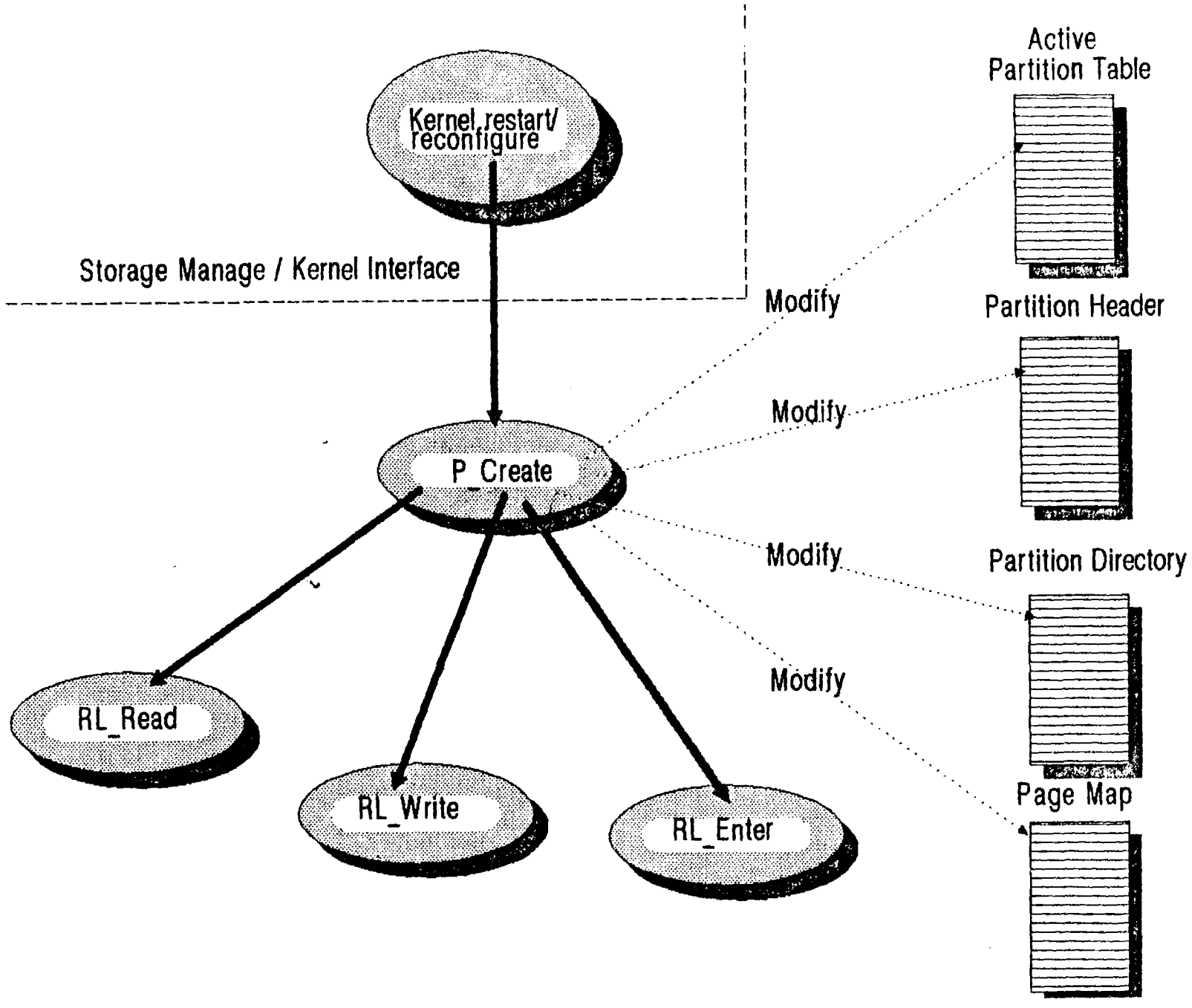

Virtual Memory
Page Fault Handling
S_Read
P_Read
Active
Segment Table
Read
Segment Header
Read
Modify
Read
Virtual Page & Disk
Map

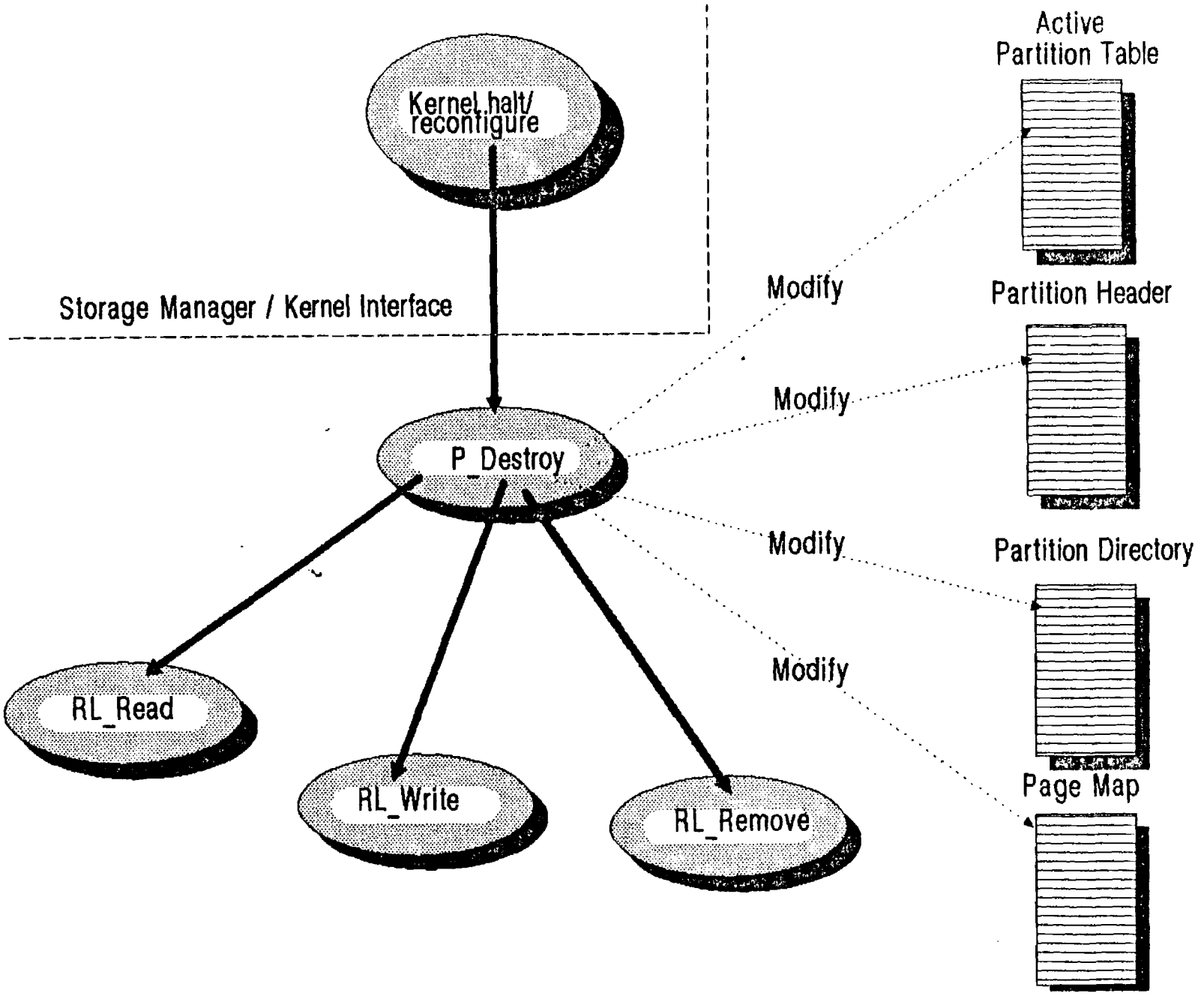

Object Management
S_Status
Read
Active
Segment Table

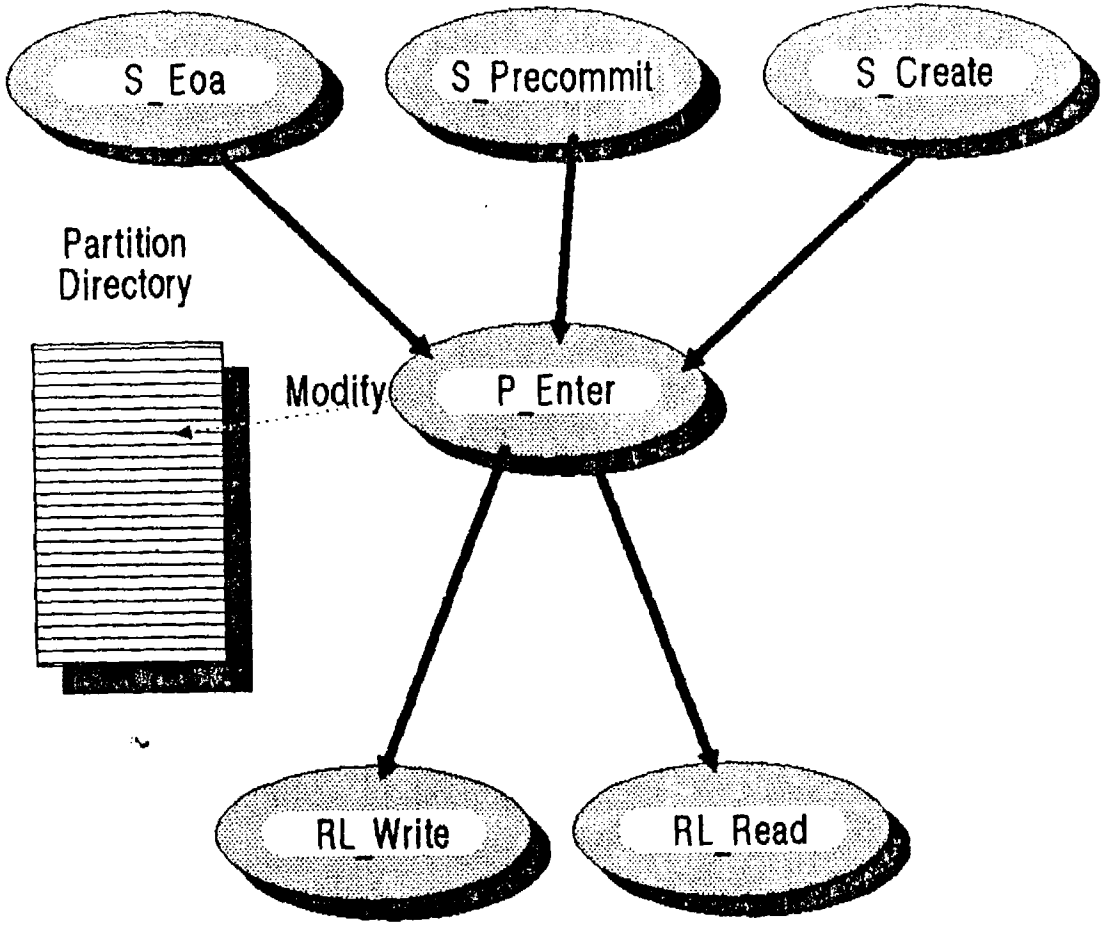

Object Management
S_UnMapWindow
P_Write
Modify
Modify
Modify
Active
Segment Table
Segment
Header
Virtual Page & Disk
Maps

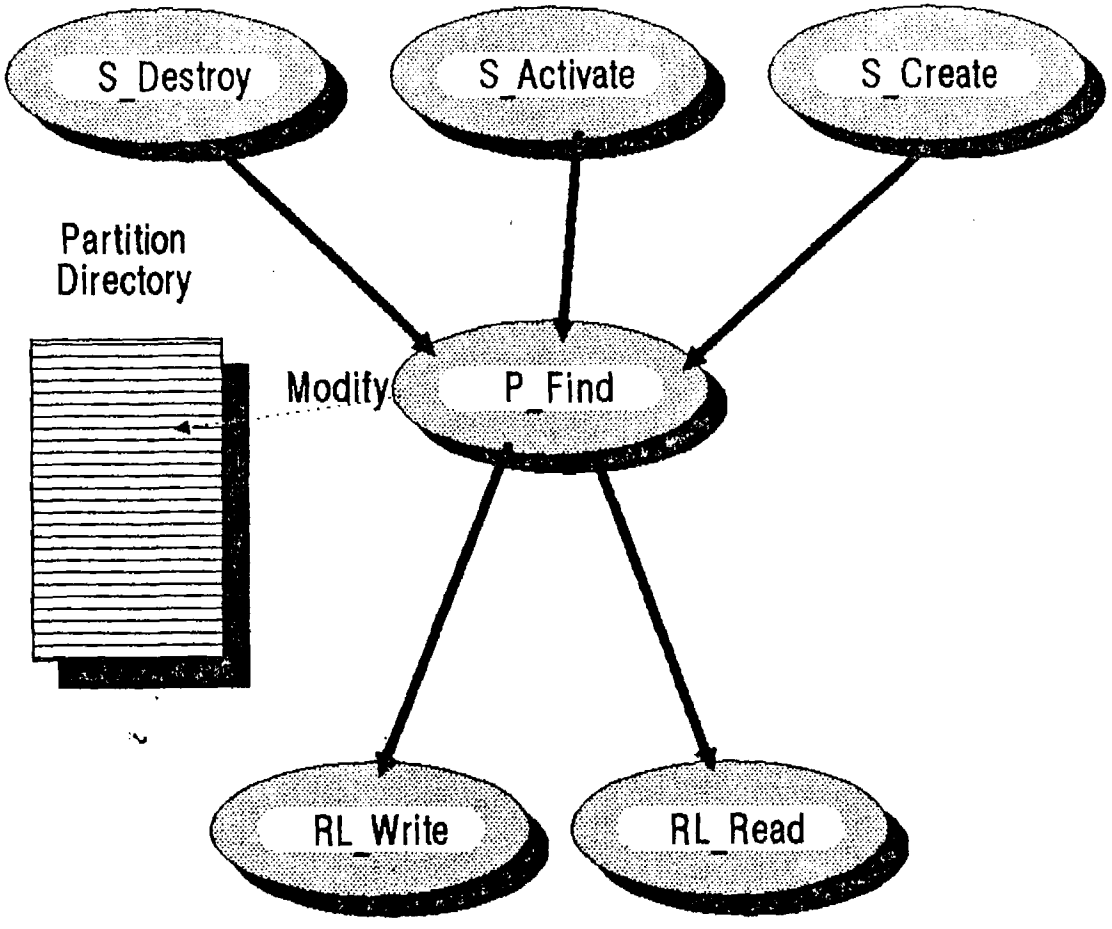

Virtual Memory
Page Replacement
Active
Segment Table
Read
Segment Header
Read
S_Write
Modify
Read
Virtual Page & Disk
Map
P_Write
P_Read
P_GetBlk

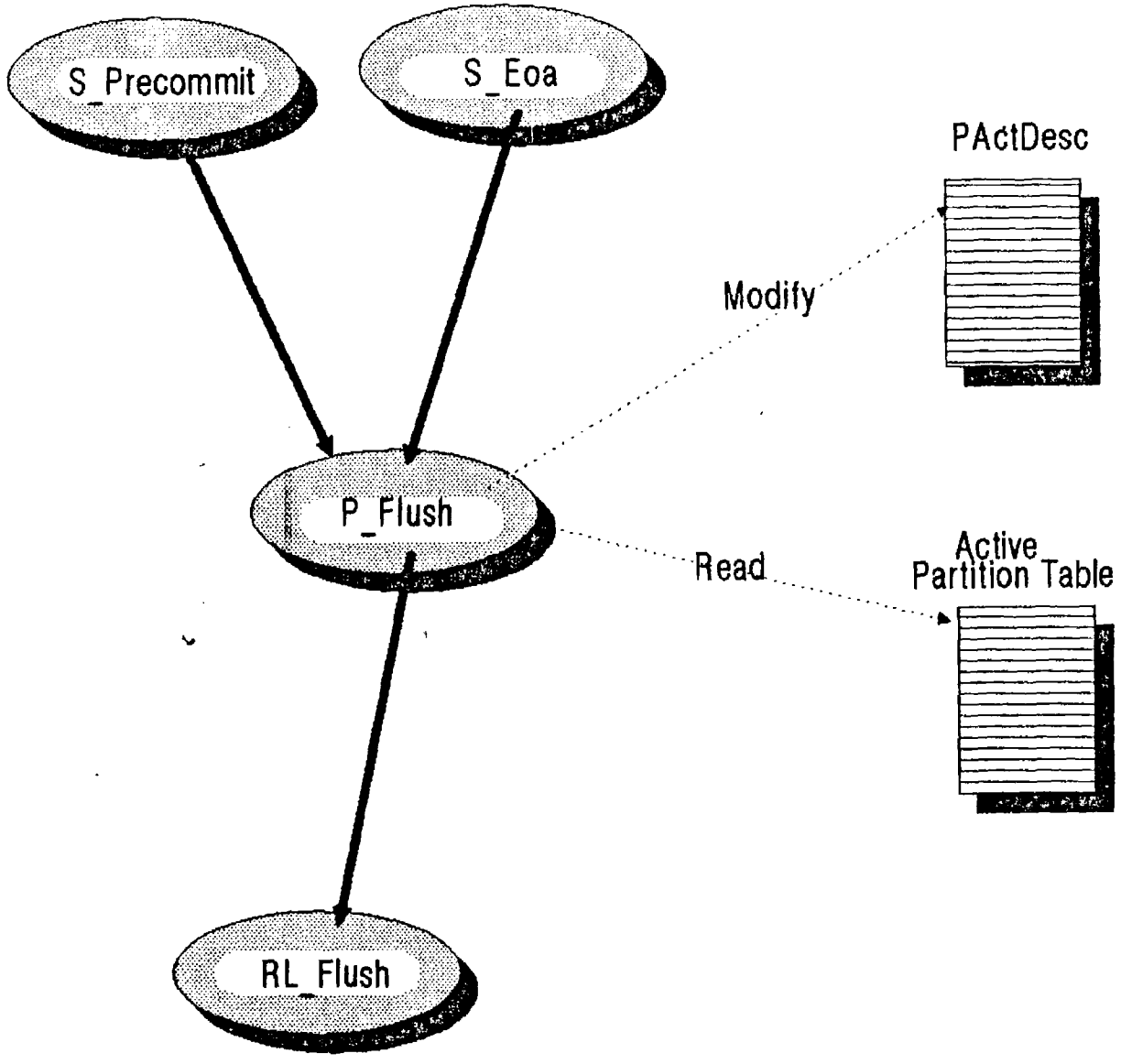

Kernel reconfigure
Storage Manager / Kernel Interface
Active Partition Table
P_Available
Read

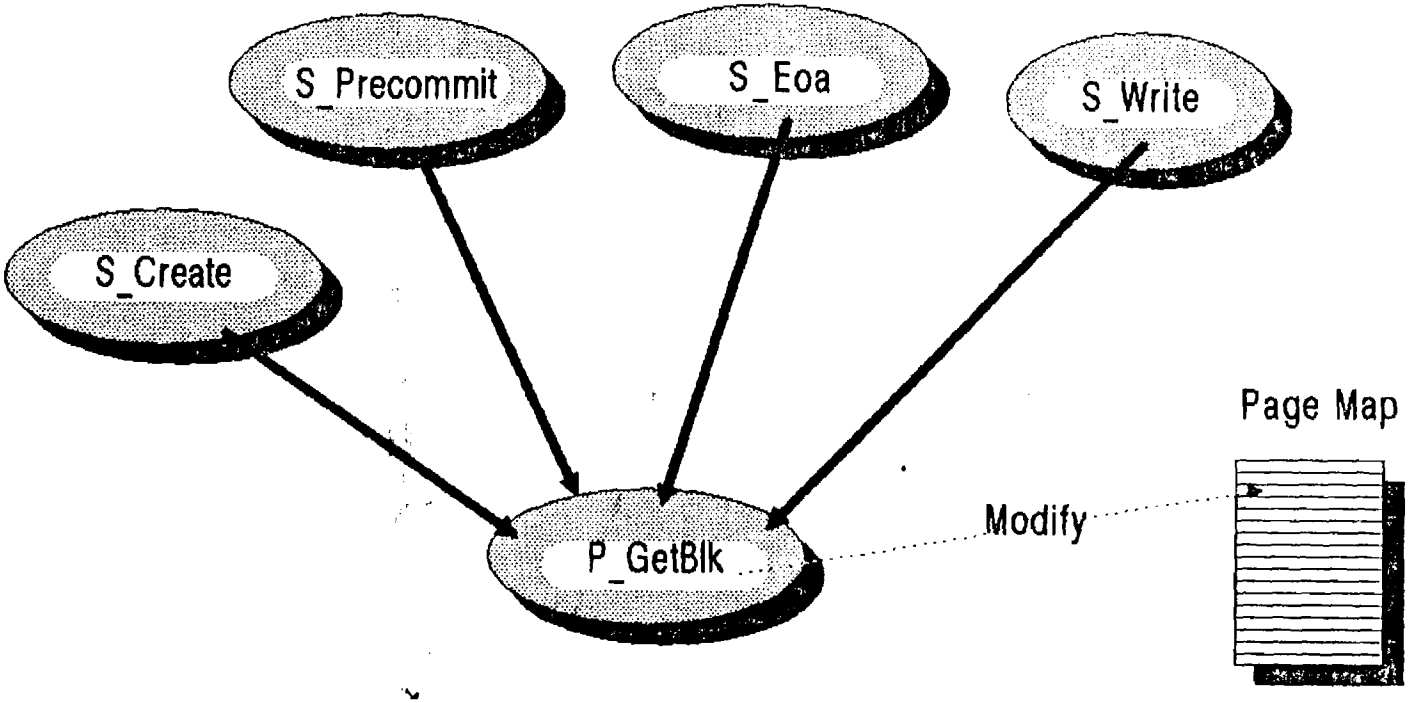

Kernel restart/
reconfigure
Storage Manage / Kernel Interface
P_Create
RL_Read
RL_Write
RL_Enter
Modify
Modify
Modify
Modify
Active
Partition Table
Partition Header
Partition Directory
Page Map

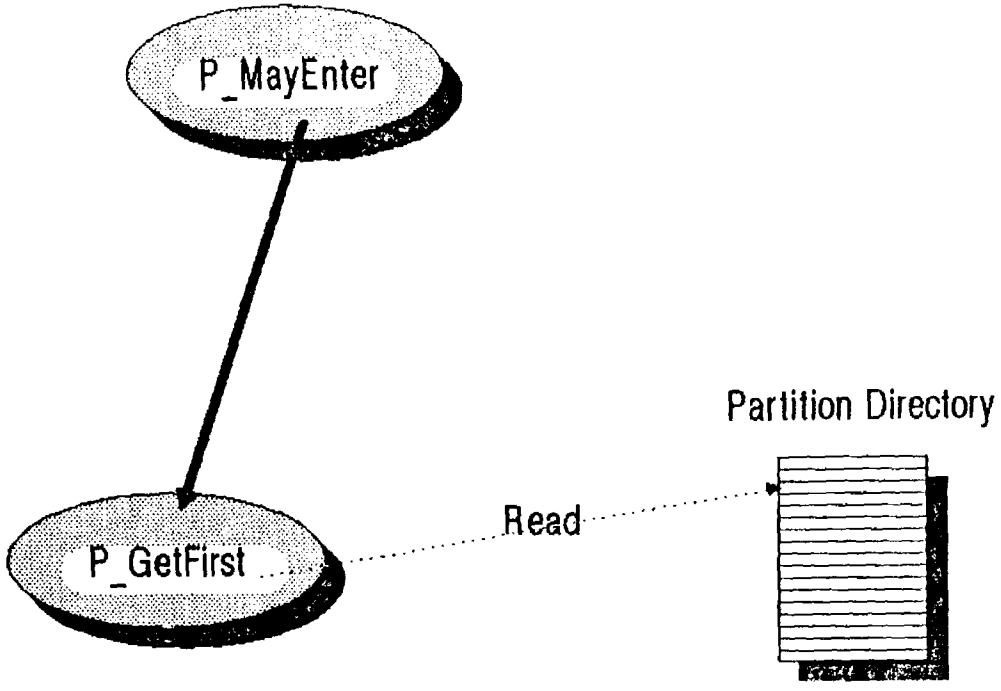

Kernel halt/
reconfigure
Storage Manager / Kernel Interface
P_Destroy
RL_Read
RL_Write
RL_Remove
Modify
Modify
Modify
Modify
Active
Partition Table
Partition Header
Partition Directory
Page Map

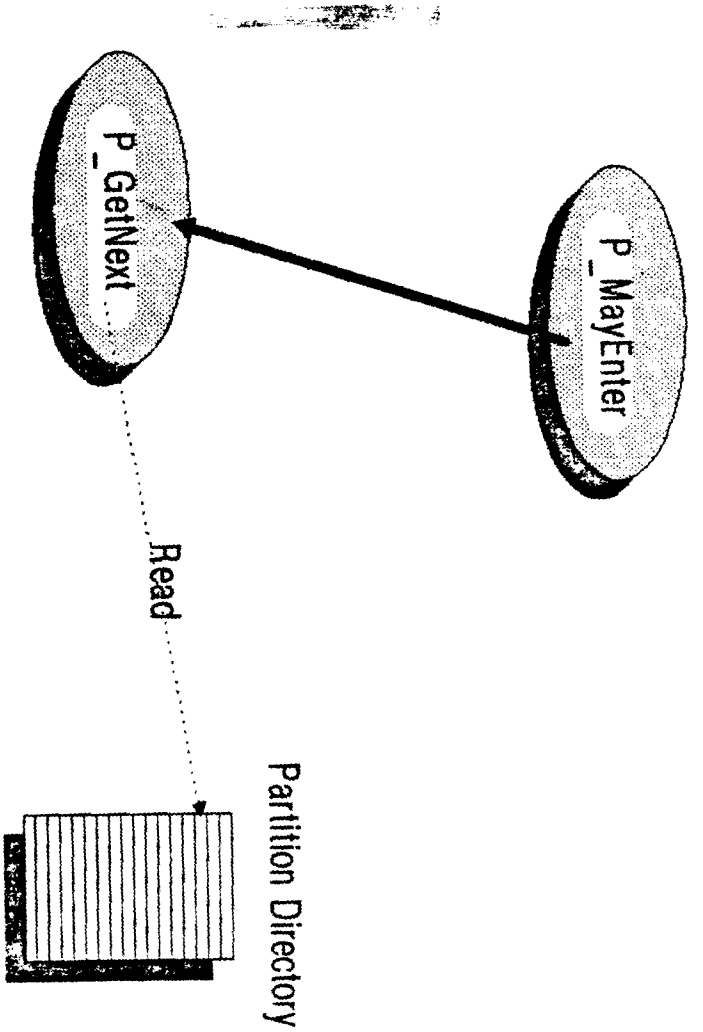

S_Eoa
S_Precommit
S_Create
Partition
Directory
Modify
P_Enter
RL_Write
RL_Read

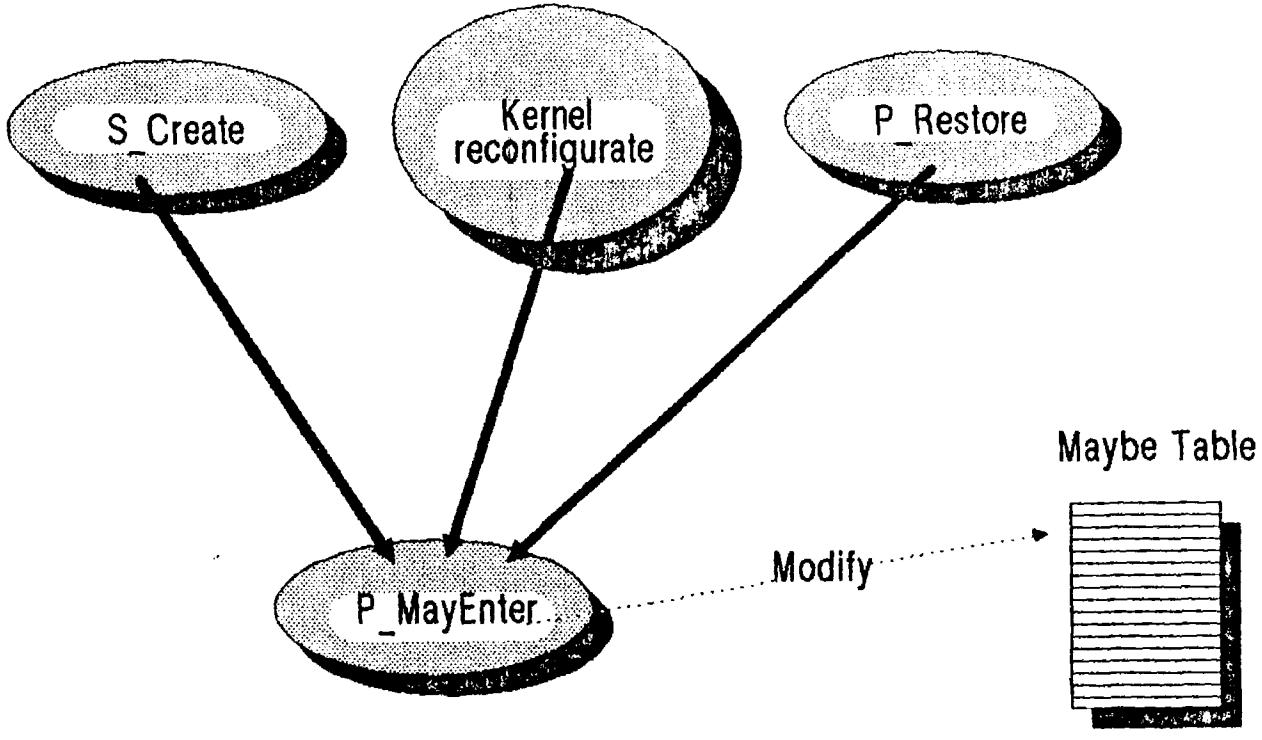

S_Destroy
S_Activate
S_Create
Partition
Directory
Modify
P_Find
RL_Write
RL_Read

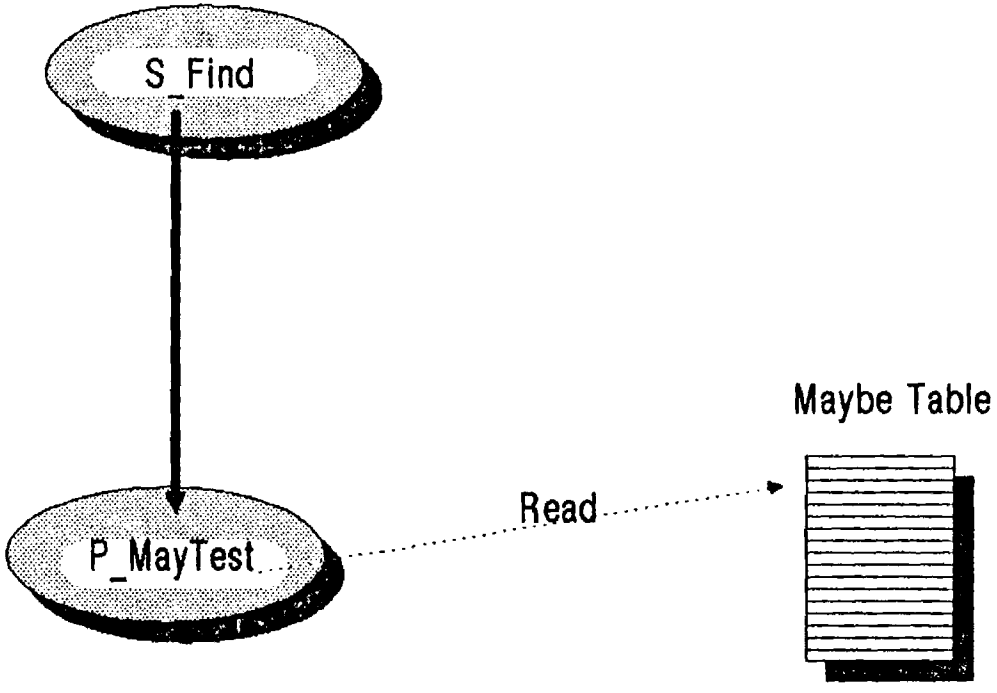

S_Precommit
S_Eoa
P_Flush
RL_Flush
Modify
PActDesc
Read
Active
Partition Table

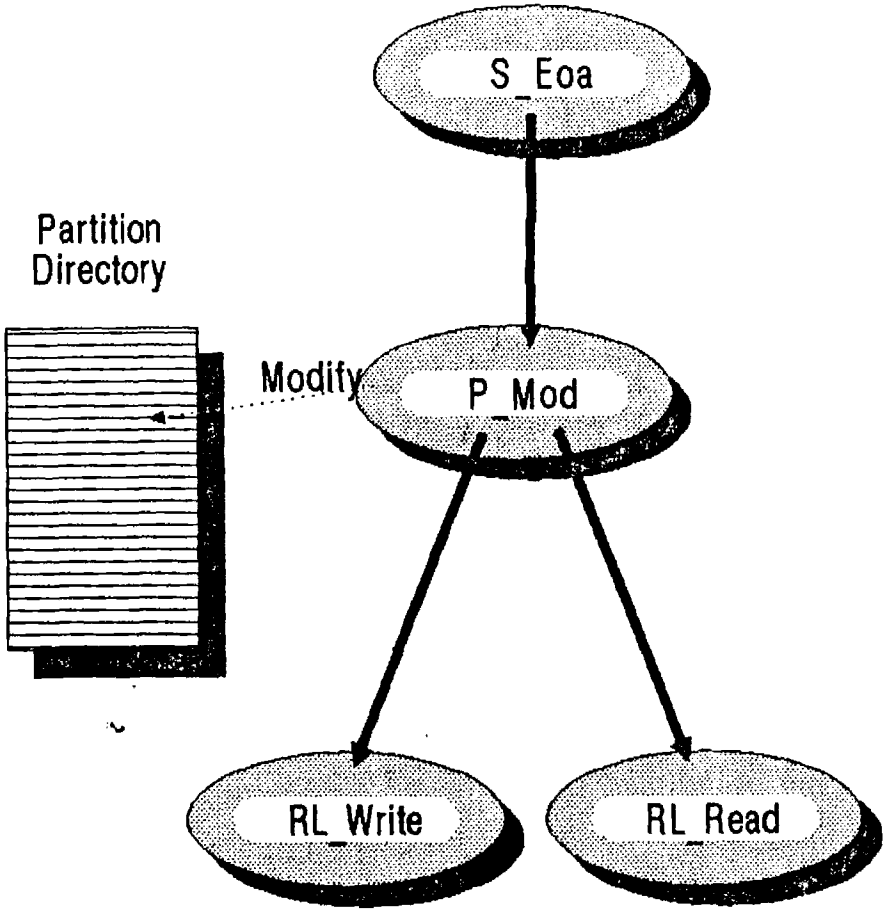

S_Precommit
S_Eoa
S_Write
S_Create
P_GetBlk
Modify
Page Map

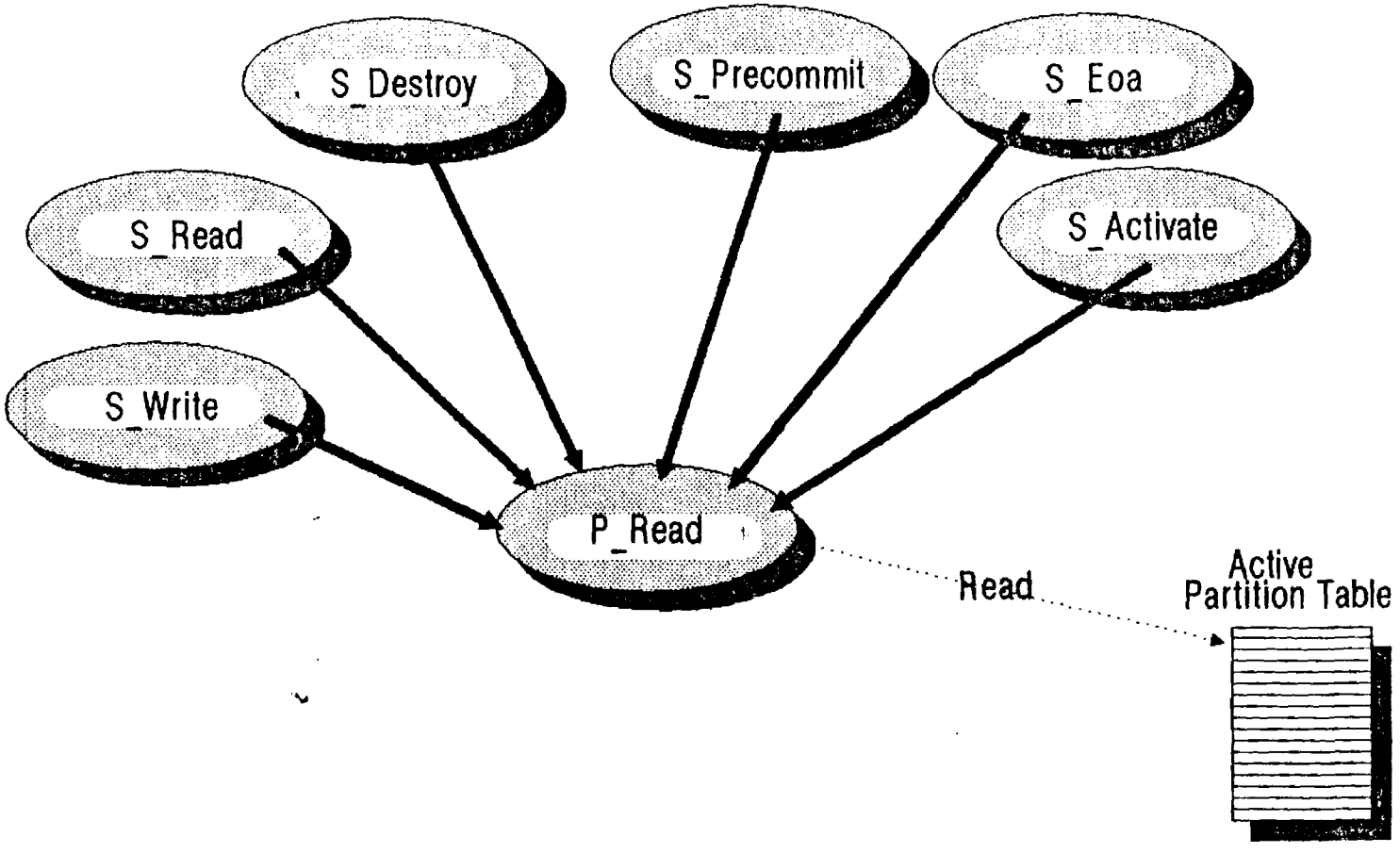

P_MayEnter
P_GetFirst
Read
Partition Directory

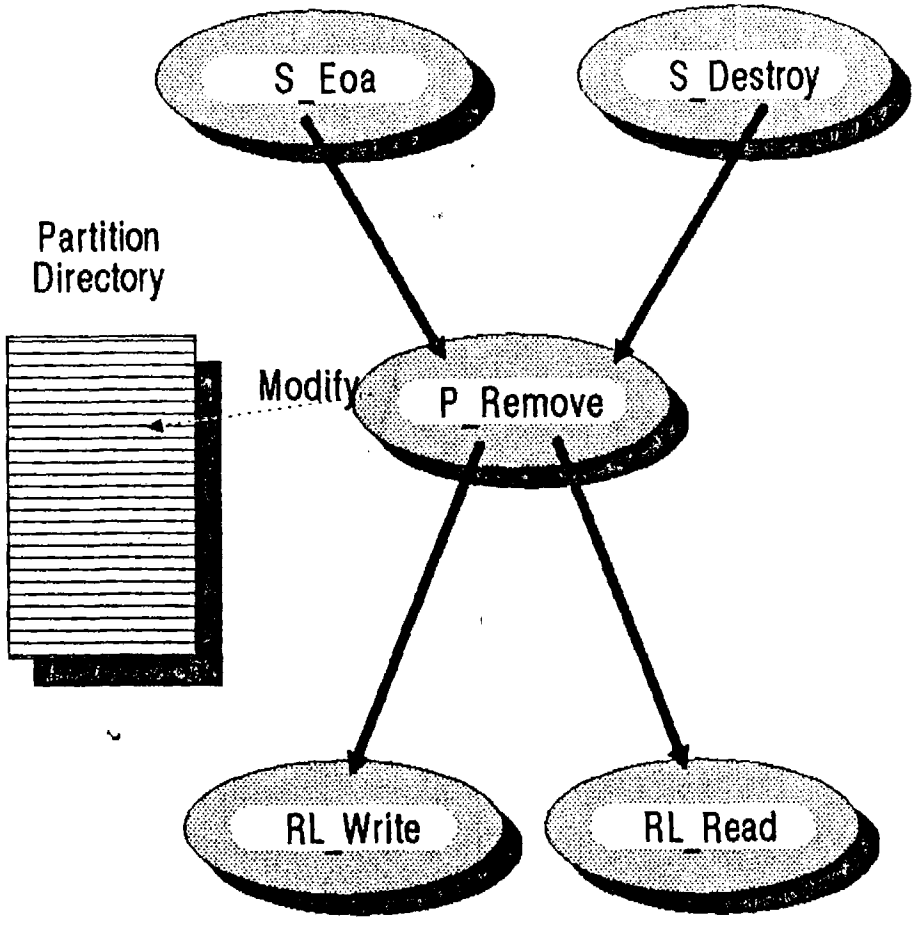

P_GetNext
P_MayEnter
Read
Partition Directory

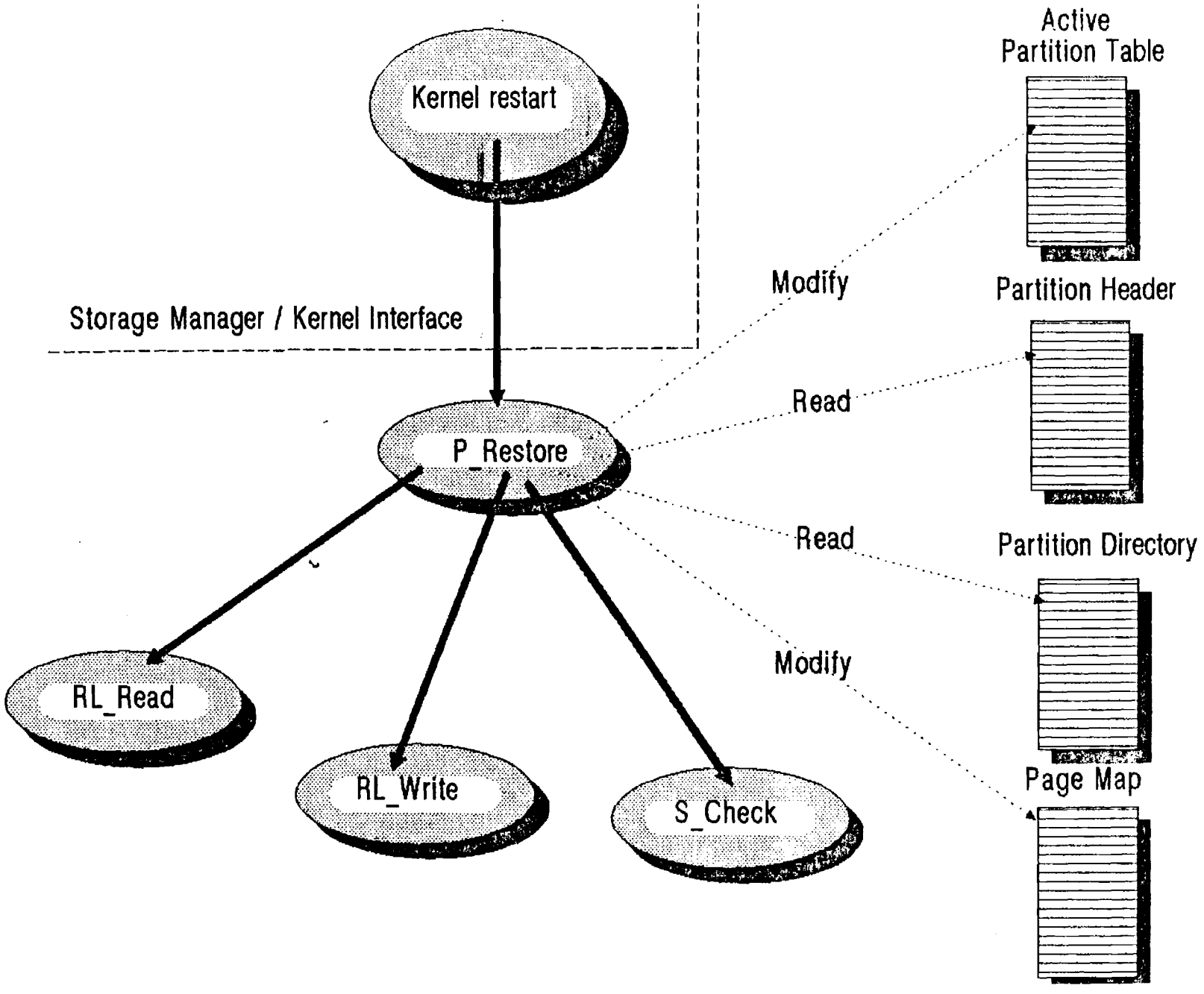

S_Create
Kernel reconfigurate
P_Restore
Maybe Table
P_MayEnter
Modify

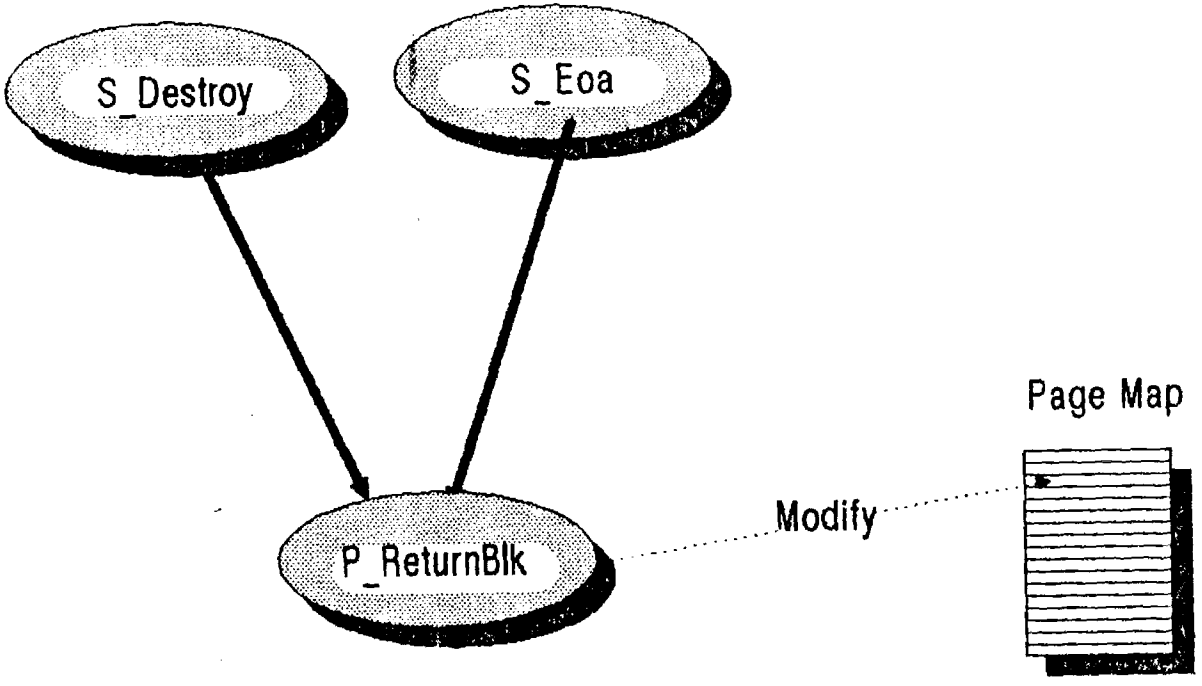

S_Find
P_MayTest
Read
Maybe Table

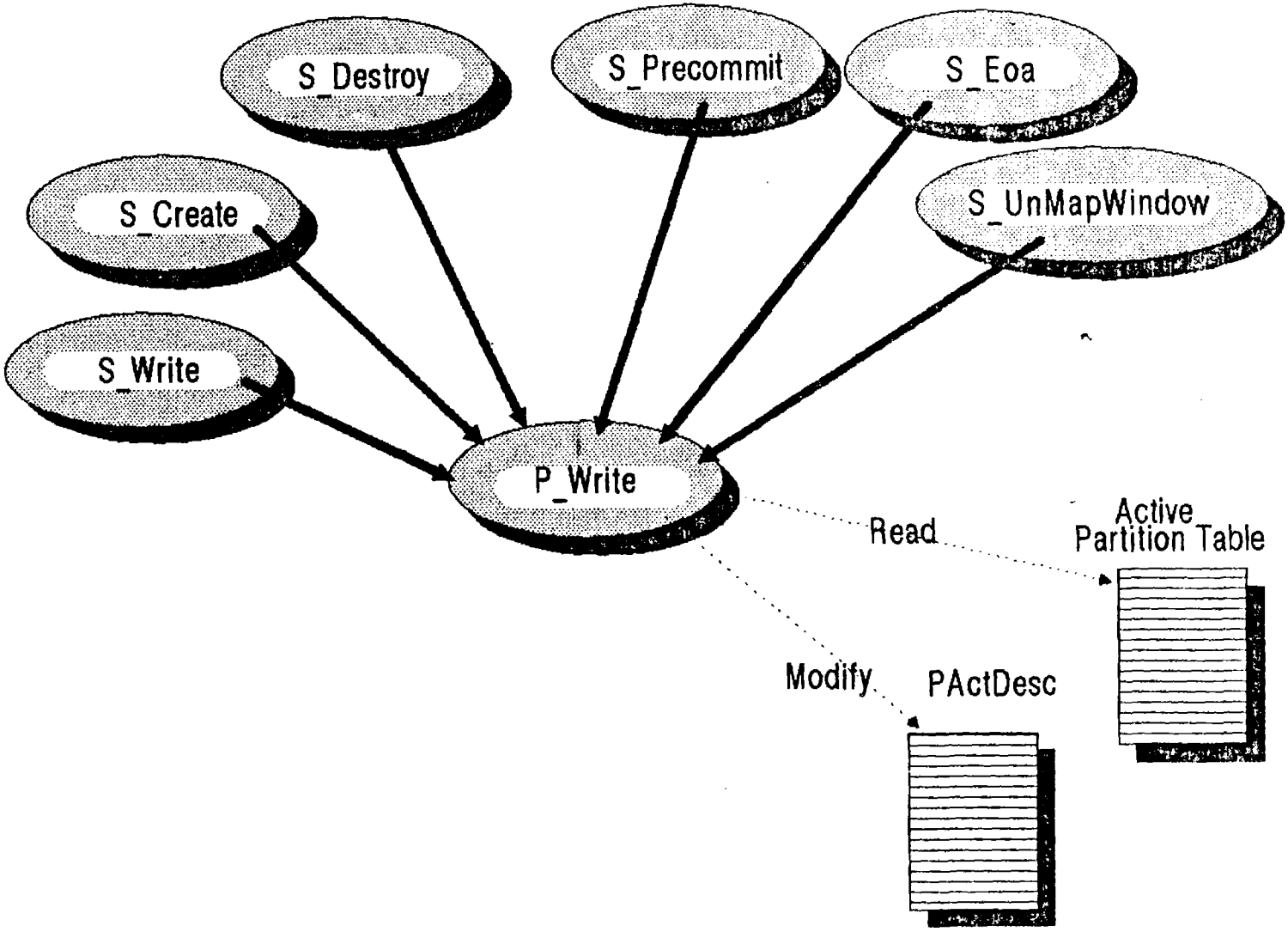

S_Eoa
Partition
Directory
Modify
P_Mod
RL_Write
RL_Read

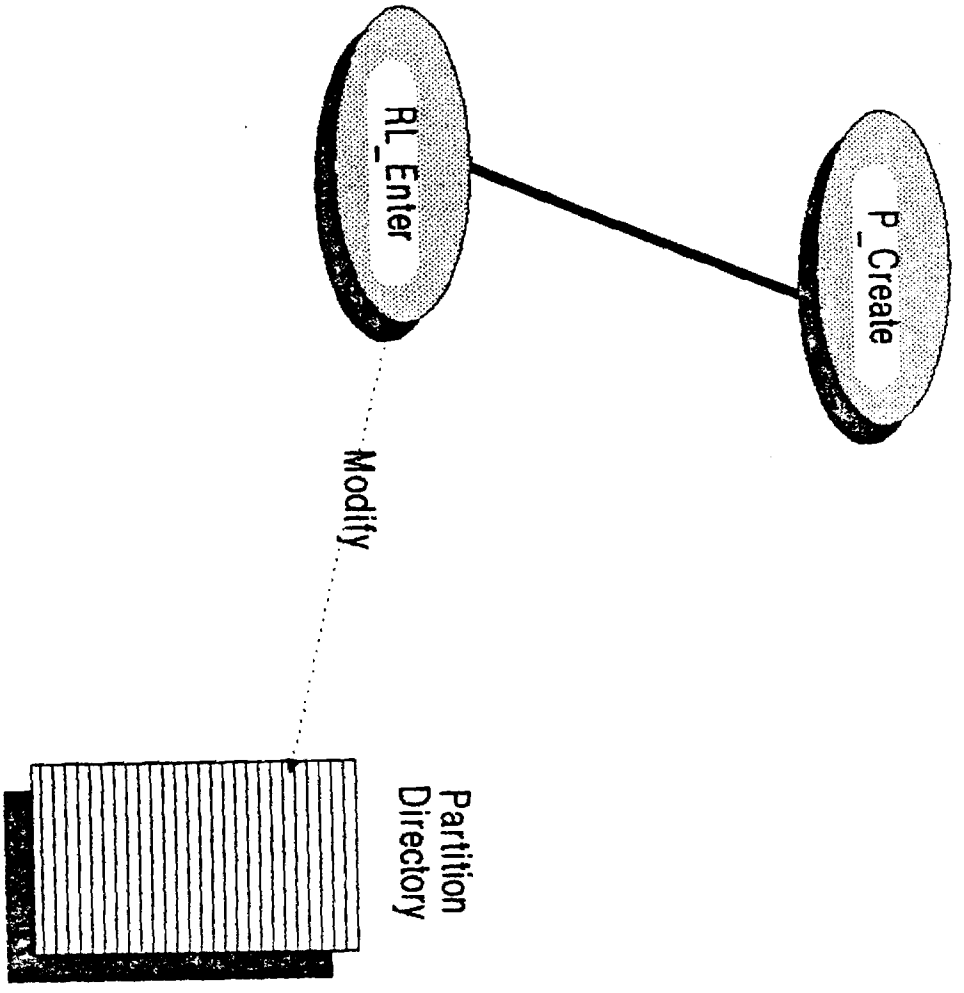

S_Destroy
S_Precommit
S_Eoa
S_Read
S_Activate
S_Write
P_Read
Read
Active
Partition Table

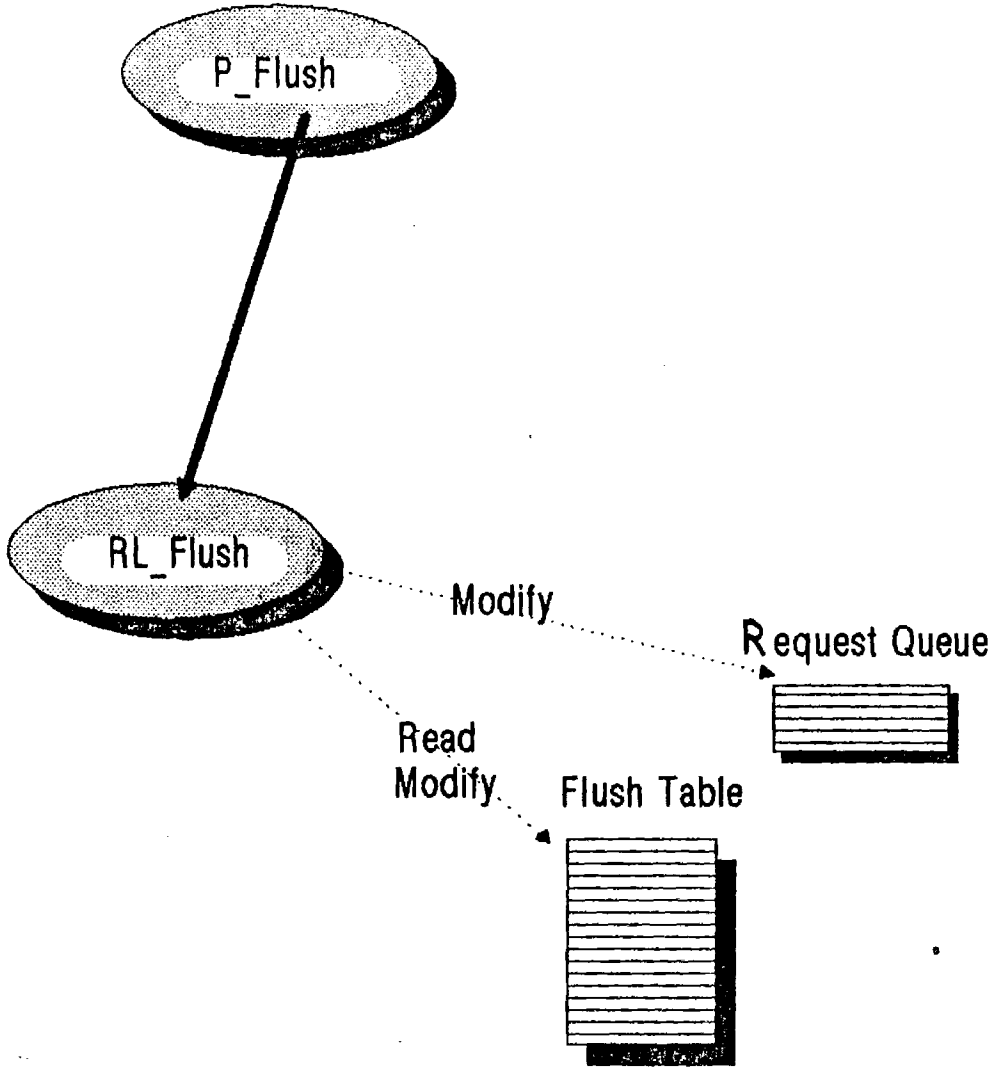

S_Eoa
S_Destroy
Partition
Directory
Modify
P_Remove
RL_Write
RL_Read

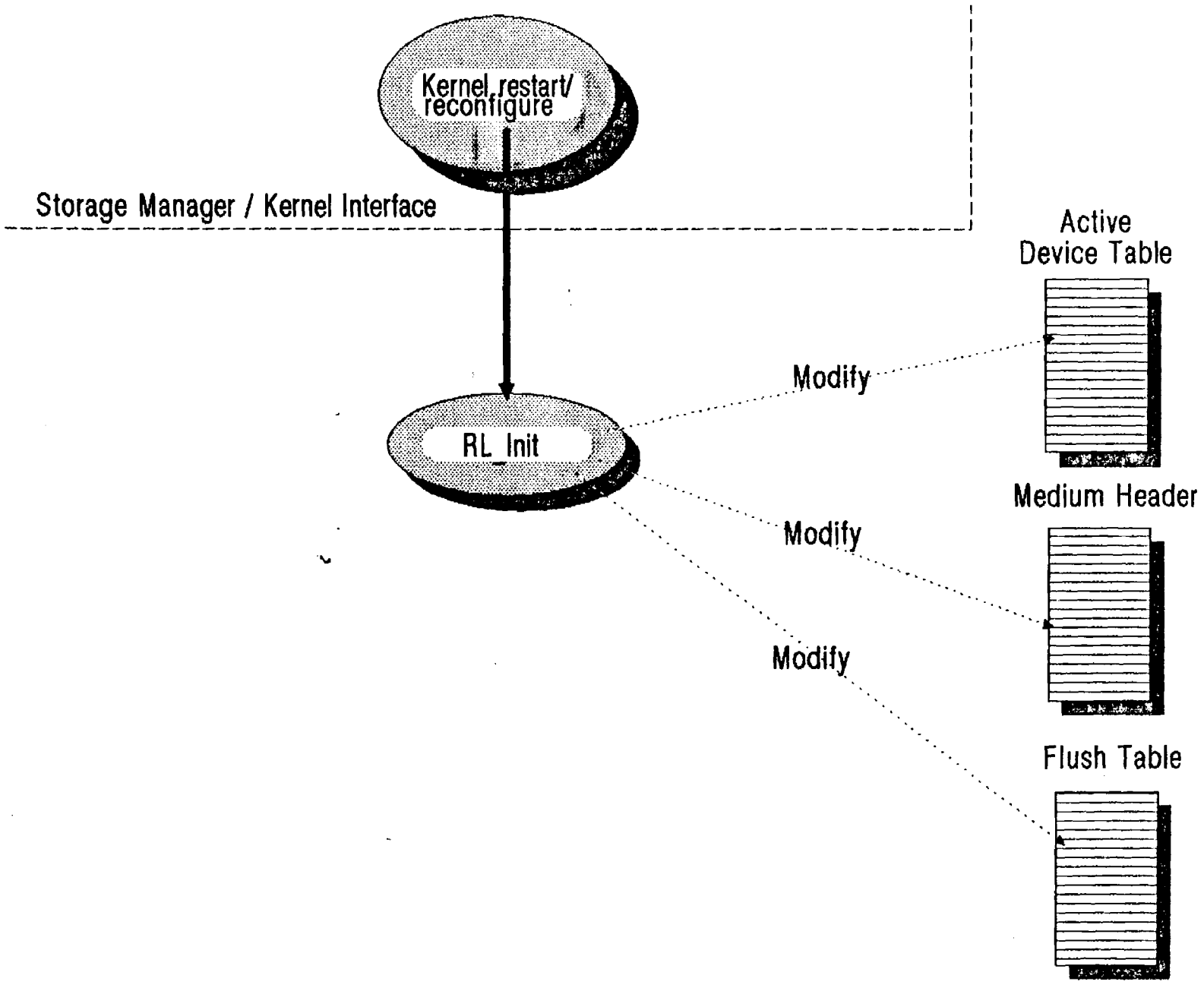

Kernel restart
Storage Manager / Kernel Interface
P_Restore
RL_Read
RL_Write
S_Check
Modify
Read
Read
Modify
Active
Partition Table
Partition Header
Partition Directory
Page Map

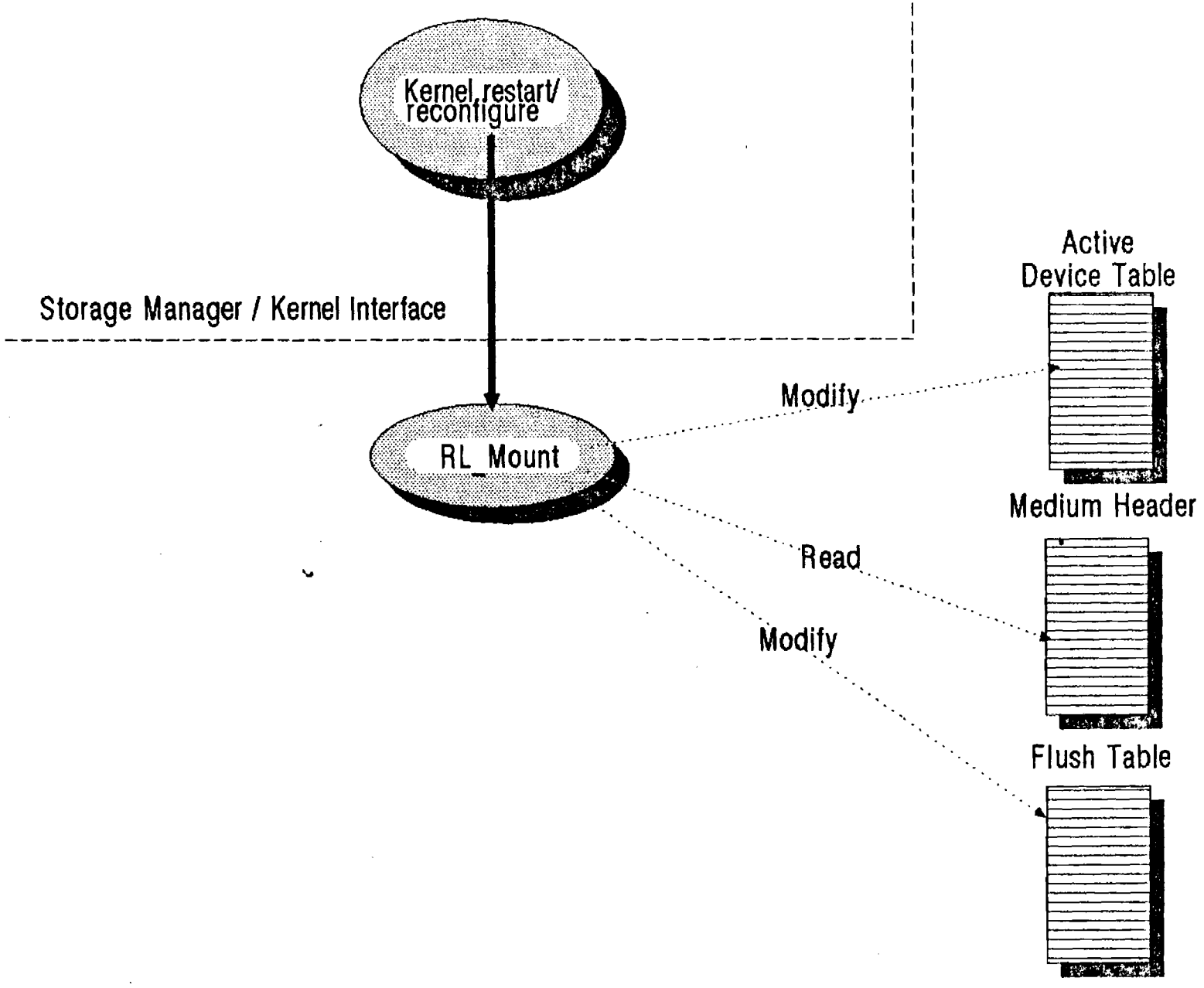

S_Destroy
S_Eoa
P_ReturnBlk
Modify
Page Map

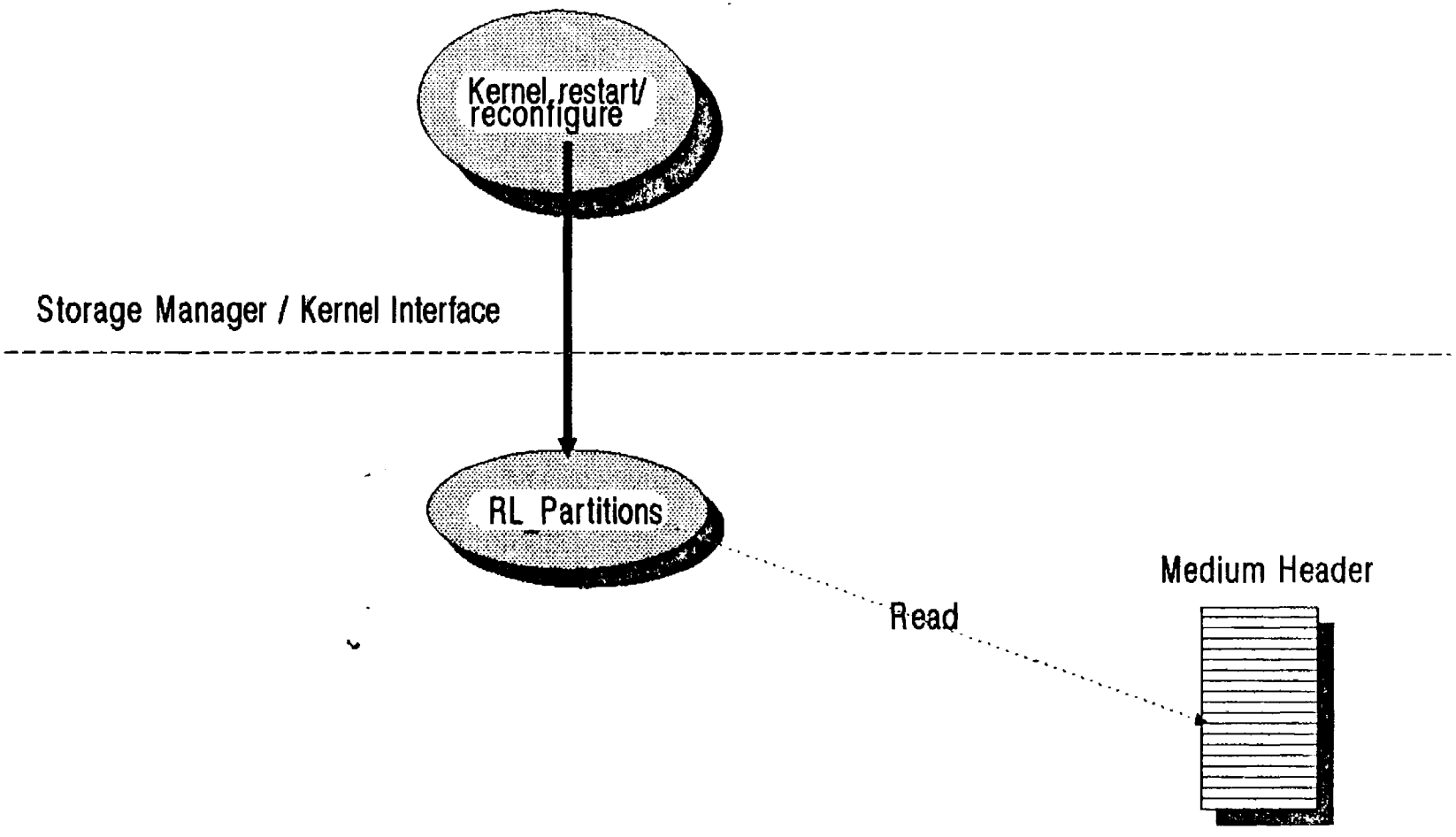

S_Destroy
S_Precommit
S_Eoa
S_Create
S_UnMapWindow
S_Write
P_Write
Read
Active
Partition Table
Modify
PActDesc

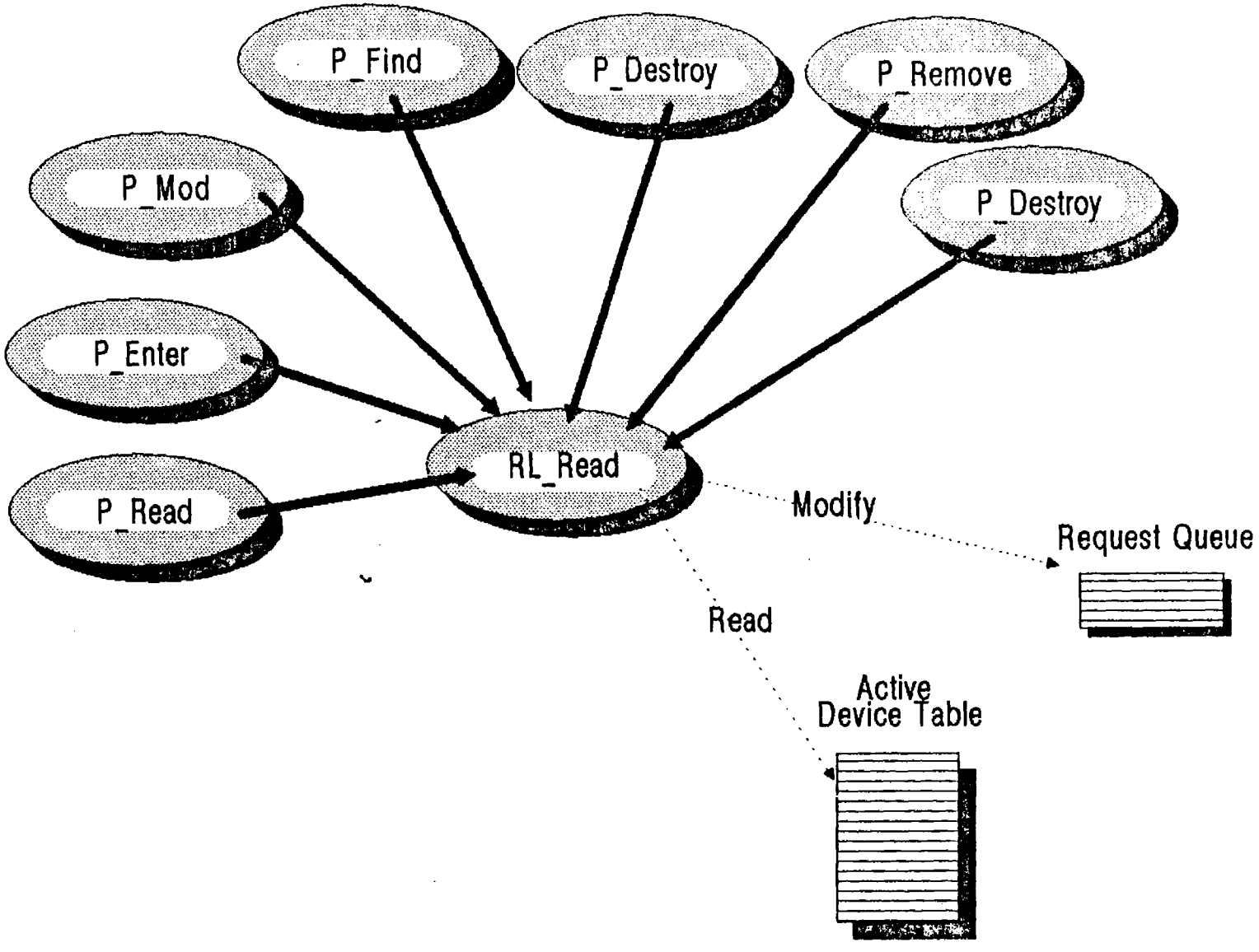

RL_Enter
P_Create
Modify
Partition
Directory

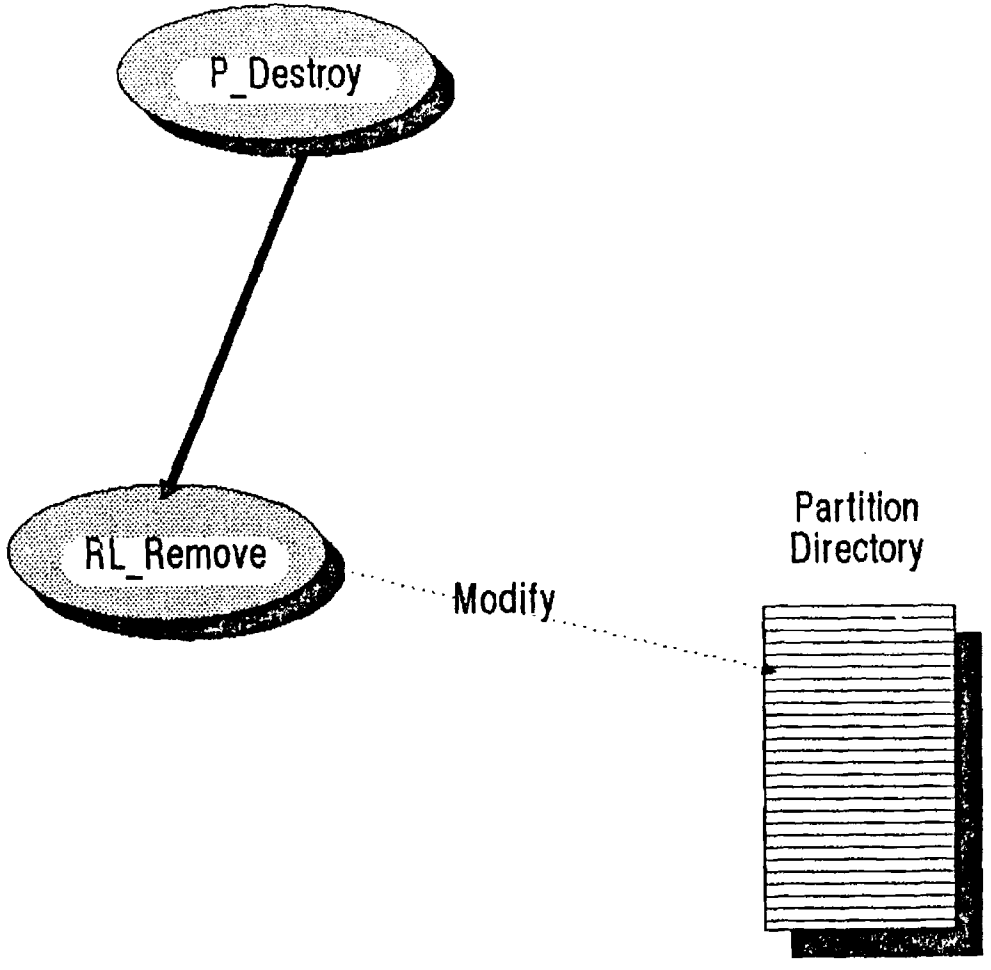

P_Flush
RL_Flush
Modify
Request Queue
Read
Modify
Flush Table

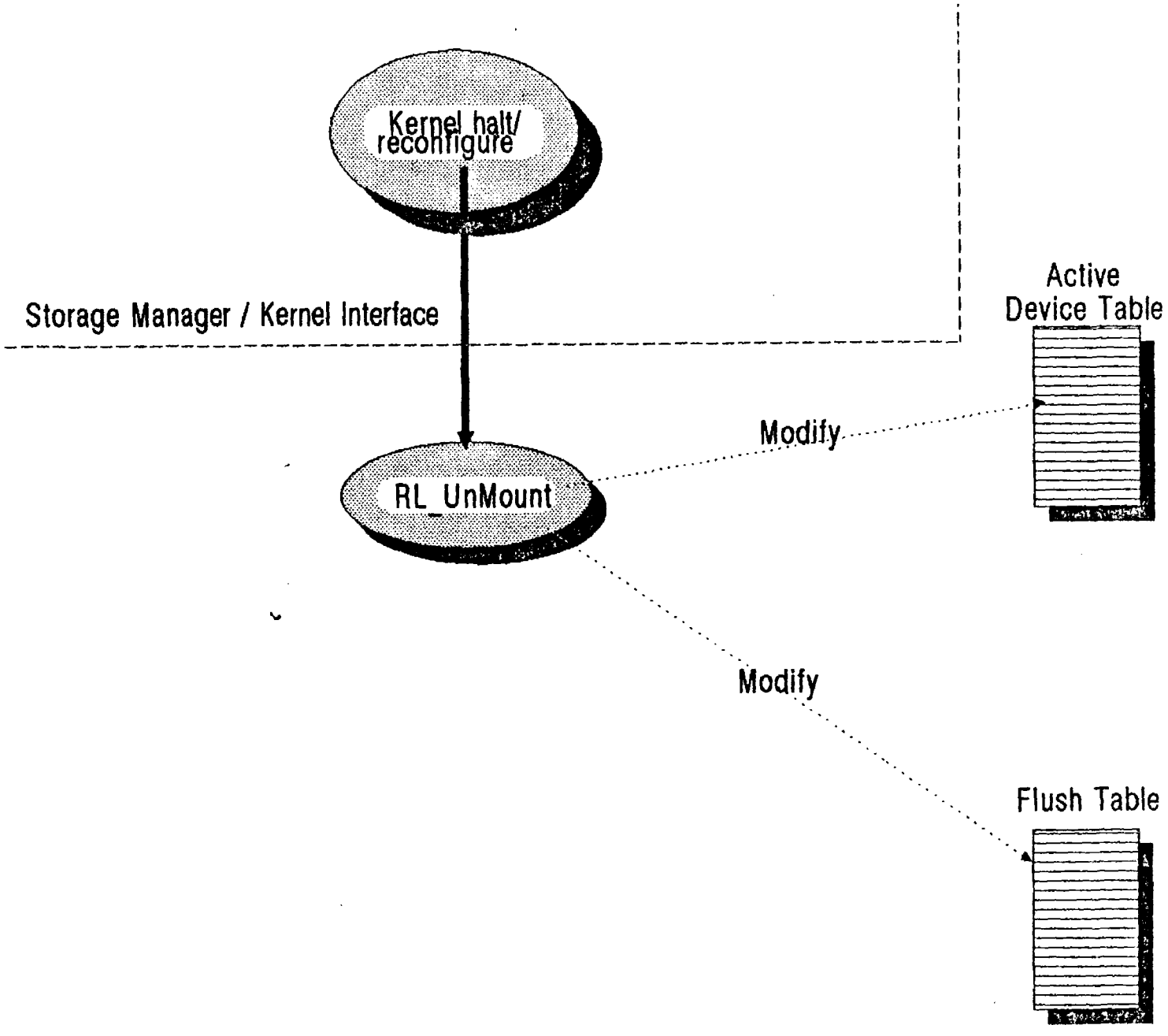

Kernel restart/
reconfigure
Storage Manager / Kernel Interface
RL_Init
Modify
Modify
Modify
Active
Device Table
Medium Header
Flush Table

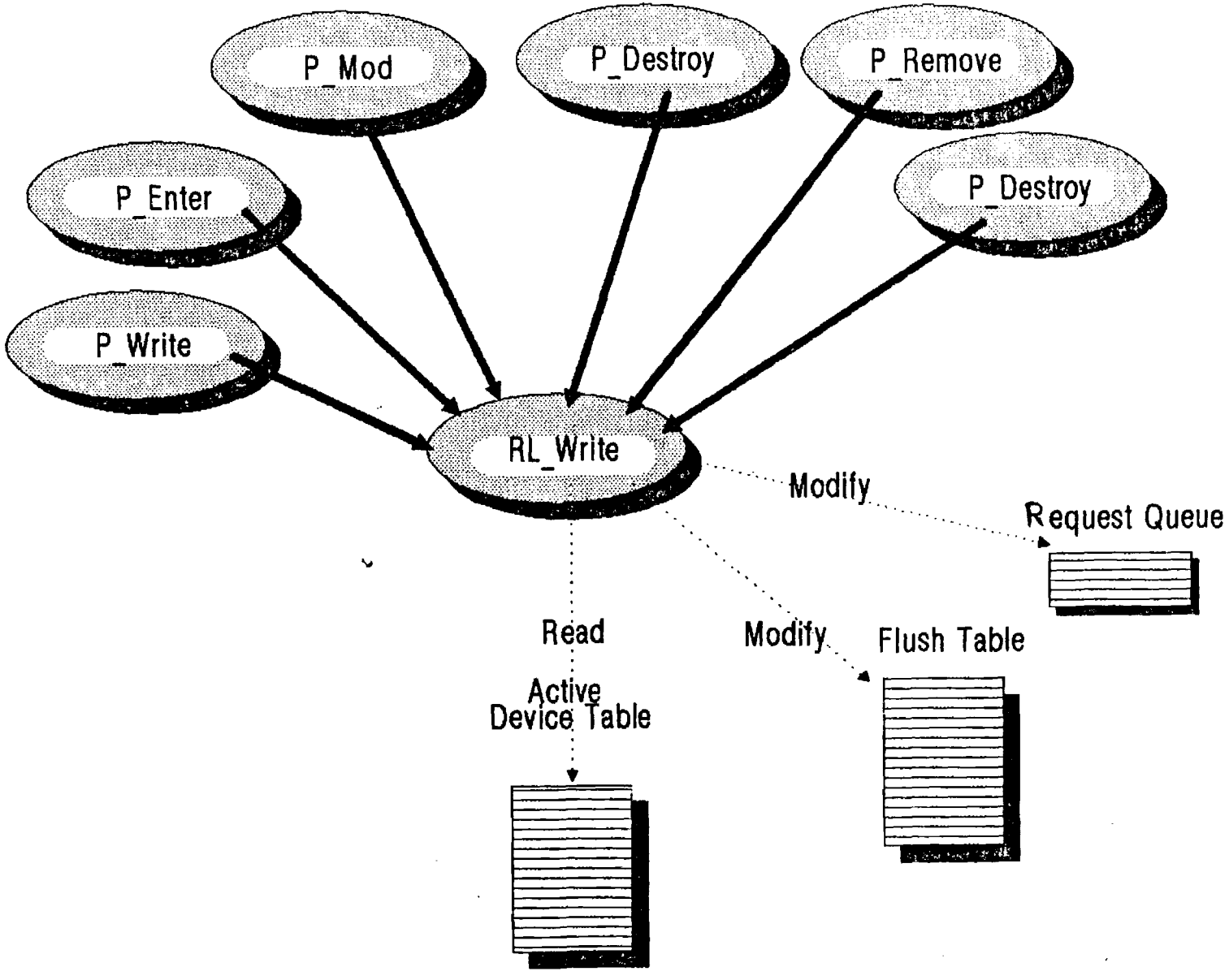

Kernel restart/
reconfigure
Storage Manager / Kernel Interface
RL_Mount
Active
Device Table
Modify
Medium Header
Read
Modify
Flush Table


Kernel restart/
reconfigure
Storage Manager / Kernel Interface
RL_Partitions
Read
Medium Header


P_Find
P_Destroy
P_Remove
P_Mod
P_Destroy
P_Enter
RL_Read
P_Read
Modify
Request Queue
Read
Active
Device Table


P_Destroy
RL_Remove
Modify
Partition
Directory


Kernel halt/
reconfigure
Storage Manager / Kernel Interface
Active
Device Table
Modify
RL_UnMount
Modify
Flush Table


P_Mod
P_Destroy
P_Remove
P_Enter
P_Destroy
P_Write
RL_Write
Modify
Request Queue
Read
Active
Device Table
Modify
Flush Table

References

[DEC82] DEC, *VAX Hardware Handbook*. Maynard, MA: Digital Equipment Corporation, 1982.

[DEC82b] DEC, *RL01/RL02 User Guide*. Maynard, MA: Digital Equipment Corporation, 1982.

[Kenl86] Kenley, G. G. "An Action Management System for a Distributed Operating System." M.S. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01.)

[Spaf86] Spafford, E. H. "Kernel Structures for a Distributed Operating System." Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/16.)

The Clouds Distributed Operating System.

Functional Description,
Implementation Details
and
Related Work.

William Appelbe, Partha Dasgupta & Rich LeBlanc

Abstract

Clouds is a distributed operating system supporting objects, actions, location independence, reliability and integration. We present a functional description of the system attributes and the impact this has on the users of distributed systems. We describe the various design and implementation decisions and how they were affected by the goals of the *Clouds* project. We also compare and contrast several well known operating systems project and their approaches.

The Clouds Distributed Operating System.

Functional Description,
Implementation Details
and
Related Work.

William Appelbe, Partha Dasgupta & Richard LeBlanc.

1. Introduction

Clouds is an operating system designed to be the forerunner of a novel class of distributed operating systems that provide the integration, reliability and structure that makes a distributed system generally usable.

Clouds is designed to run on a set of general purpose computers (uniprocessors or multiprocessors) that are connected via a medium-to-high speed local area network. The structure of *Clouds* promotes transparency, support for advanced programming paradigms, and decentralized yet integrated control. The major design objectives for *Clouds* are:

- Integration of resources through cooperations and location transparency.
- Support for robust transaction processing, and the ability to achieve fault tolerance.
- Efficient design and implementation.
- Simple and uniform interfaces for distributed processing.

The system structuring paradigm chosen after substantial research for the *Clouds* operating system is an object/process/action model. All instances of services, programs and data in *Clouds* are objects. Processing is done by atomic actions. Provision is made for processing that must execute outside the constraints of atomicity [Al83, DaLe85, McAl83]. In the next few pages we provide a functional description of the system, and some implementational details.

2. Objects

All data, programs, devices and resources on *Clouds* are objects. The only entities that are not objects are processes and actions. A *Clouds* object at the lowest level of conception is a virtual address space. Unlike conventional virtual address spaces, a *Clouds* object is neither tied to any process nor is volatile. A *Clouds* object exists forever (like files) unless if it is explicitly deleted.

Every *Clouds* object is named. The name of an object, also known as the capability, is unique over the entire distributed system and does not include the location of the object. That is, the capability based naming scheme in *Clouds* creates a uniform, flat system name space for objects. The capabilities not only provide the naming mechanism, they can also be used for access control and protection.

Since an object consists of a named address space (and its contents), it is completely passive. Unlike those in some object based systems, a *Clouds* object is not associated with any server process. Processes are allowed to execute within the context of objects. A process executes in an object by entering it through one of several defined entry points, and after the execution is complete the process leaves the object. Several processes can simultaneously enter an object and execute in parallel.

Objects have structure. They contain, minimally, a code segment, a data segment and a heap for local storage allocation. Processes that enter an object execute in the code segment. The data segment is accessible to the code in the code segment, but not to any other object. Thus the object has a wall around it which has some well defined gateways, through which activity can come in. Data cannot be transmitted in or out of the object freely, but can be moved as parameters to the code segment entry points (see discussion on processes).

Clouds objects are user-defined or system-defined. Most objects are user-defined. Some examples of system defined objects are device drivers, name-service handlers, and the *Clouds* kernel[†] itself.

A complete *Clouds* object can contain user-defined code and data, system-defined code and data that handle synchronization, recovery and commit, a volatile heap for temporary memory allocation, a permanent heap for allocating memory that will remain permanent as a part of the data structures in the object, locks and capabilities to other objects.

3. Processes

The only form of activity in the *Clouds* system is the process. *Clouds* processes are lightweight workers. A process is composed of a process control block (PCB) and and a virtual space containing the stack. Thus, a process can be viewed as a program counter, stack pointer and stack. Upon creation a process starts up at an entry point of an object. As the process executes, it executes code inside an object and manipulates the data inside this object. The code in the object can contain a procedure call to an operation of another object. When a process executes this, it temporarily leaves the caller object and enters the called object, and commences execution there. The process returns to the caller object after the execution in the called object terminates. The calls to the entry point of objects are called *object invocation*. Object invocations can be nested. The code that is accessible by each entry point is known as an *operation* of the object.

Thus a process executes by processing operations defined inside many objects. Unlike processes in conventional operating systems, the process thus often cross boundaries of virtual address spaces. Addressing in an address space is however limited to that address space, and thus the process cannot access any data outside an address space. Control transfer between address spaces occurs though object invocation, and data transfer between address spaces

[†]Although the specification define the kernel to be an object, the implementation treats it as a special case (or pseudo-object) for

occurs through parameters to object invocation (which may be capabilities for other objects).

When a process executing in an object (or address space) executes a call to another object, it can provide the called operation with arguments. When the called operation terminates, it can send back result arguments. Since the address spaces of the two objects are disjoint, all arguments are passed by value. This argument passing mechanism is identical to copy-in copy-out semantics of parameter passing supported by many programming languages.

4. The Object/Process Paradigm

The structure created by a system composed of objects and processes has several interesting properties. First, all interfaces are procedural. Object invocations are equivalent to procedure calls on modules not sharing global data. The modules are permanent. The procedure calls work across machine boundaries. Since the objects exists in a global name space, there is no concept of machine boundaries. At the system level local invocations and remote invocations (RPC) are differentiated, however this is transparent to the user and the programmer.

Since every entity is an object and objects are permanent, there is no need for a file system. A conventional file is a special case of an object, an object with a read, a write, a seek and some other file operations defined in it to transport data in and out of the object through parameters provided to the calls.

Though we can simulate files by using objects of type *file*, the need for files disappear in most situations. Programs, do not need to store data in files, they can keep the data in the data space, since the data space does not disappear when the controlling process terminates. The need for user-level naming of files transforms to the need for user-level naming for objects.

The *Clouds* operating system does not provide any support for I/O operations, except for terminal I/O. (Terminal I/O is achieved by invoking the read and write operations on a terminal object, dispensing most concepts about I/O streams).

Just as I/O is eliminated, so is the need for messages. Processes do not communicate through messages. There are no ports. This allows a simplified system management strategy as the system does not have to maintain linkage information between processes and ports. Just as files can be simulated for those in need for them, messages and ports can be easily simulated by an object consisting of a bounded buffer that implements the send and receive operations on the buffer. However, we feel that the need for files and messages are the product of the programming paradigms designed for systems supporting these features, and these are not necessary structuring tools for programming environments.

The view of the computing environment created by *Clouds* is apparent. It is a simple world of named address spaces (or objects). These object live in computing systems on a

efficiency reasons.

LAN, but the machine boundaries are made transparent, creating a unified object space. Activity is provided by processes moving around amongst the population of objects through invocation; and data flow is implemented by parameter passing.

This view of a distributed system, does have some pitfalls. Processes aborting due to errors will leave permanent faulty data in objects they modified. Failure of computers will result in similar mishaps. Multiple processes invoking the same object will cause errors due to race conditions and conflicts. More involved consistency violations may be the results of non-serializable executions. In a large distributed system, having thousands of objects and dozens of machines, corruption due to failure cannot be tolerated or easily repaired. The prevention of such situations is achieved through the use of the atomic actions paradigm, discussed below.

5. Actions

Actions are units of work that are defined by the programmer to be atomic. The work done by an action either gets done in its entirety or does not happen at all. Failures, errors, and aborts thus do not leave a trace on the data stored in the objects [Ke86, Mc85].

An action is an abstraction. It is neither an object nor a process. It is a high level concept that exists as information in the action management system. An action starts as one process. Anything this process does, until the action commits or aborts, is in the context of the action. If the process creates more processes, these too are part of the same action.

All the activity of the set of processes (or one process) in the context of an action, consists of *touching* objects. A object is considered *touched* by an action if a process executing on behalf of the action executes one or more object invocations on the object. A touched object is not necessarily modified. All objects modified by an action exist in a volatile form that may be different from their permanent representations.

When an action terminates, all the objects touched by the action are *committed*. Commitment of an object is achieved by updating its permanent representation by replacing any data that have been modified by this action. Since the updates by an uncommitted action are never made permanent, an aborted action is rolled back by default.

Though the updates by an uncommitted action are not written on permanent (secondary) storage, the updates of an uncommitted action may be seen by another uncommitted action accessing the same object, depending upon the synchronization method used by the object. We distinguish between two kinds of atomicities of actions, namely *failure atomicity* and *view atomicity*. Failure atomicity dictates that either the updates performed by an action are made permanent after the action runs to completion, or nothing occurs. View atomicity dictates that the action is insulated from seeing any results from other concurrently executing actions. *Clouds* can provides failure atomicity and, if needed, view atomicity. Note that not providing view atomicity can lead to errors (an action A makes updates based on some results of an uncommitted B action and A commits while B aborts). The differences between the atomicity

requirements and the rationale for providing failure atomicity will be clearer after the discussion of synchronization methods.

5.1. Nested Actions

Actions, as units of work, are too large grained for many applications, especially in an large distributed environment where failures are relatively common. Any error or failure during the execution of an action requires that the entire action be aborted. An action often needs the ability to recover from errors or failures. Finer grained atomicity and failure recovery capabilities are provided by nested actions.

A top-level action is the conventional action. The-top level action can delegate sub-tasks to subordinated actions or sub-actions, which in turn can spawn sub-actions, giving rise to a tree of nested actions. A child action executes in the context of its parent, but the failure of the child does not imply the failure of the parent, the parent may choose to retry the sub-task or respond to the failure in some other way. The commit of a sub-action is conditional upon the commit of the parent, and by transition, the commit of all nested actions are conditional upon the commit of the top-level action. Thus a top-level action makes the final commit decision, based on the commit and abort status of all the nested action it gave rise to.

The nested action semantics of *Clouds* is identical to the semantics defined by Moss in [Mo81]. Nested actions thus provide a action programmer with failure containment firewalls and the ability to try alternate methods to make progress.

6. Synchronization

The synchronization scheme decides how (if at all) concurrent processes execute in the same object. The synchronization scheme used also dictates whether action using the object are view atomic or not. Both the synchronization techniques used and the recovery techniques used affect the semantics of action atomicity. We discuss synchronization in this section. Recovery will be discussed in the next section and the effects of both on actions will be briefly considered [McAlMc82].

Clouds offers two basic types of synchronization: *custom* and *automatic*. Custom synchronization allows the programmer of an object to define and implement the synchronization rules. For this purpose, the programmer has access to locks and semaphores that can be defined and used inside the object. For example, setting a lock on a variable when entering an operation and releasing it upon exit causes processes that execute this operation to run in mutual exclusion. The object programmer can thus customize the synchronization scheme to the needs of the object.

Though custom synchronization can be correct and useful for many applications, it is possible to allow non-serializable execution in custom synchronization schemes. Allowing various unconventional schemes is the power of custom synchronization. However in cases where serializability is necessary, the programmer need not implement any synchronization;

automatic synchronization is available for this purpose.

With automatic synchronization, each entry point in an object is marked as a read entry or a write entry. When an action touches an object for the first time, a read or write lock is obtained on the entire object (as appropriate). Conversions from a read lock to a write lock is allowed. Locks are held until the action commits, implementing a two-phase locking protocol and guaranteeing serializable execution of the action with respect to all data touched by the action (provided all objects it touched were using automatic synchronization). This scheme also provides view atomicity of actions.

7. Recovery

Recovery is managed by shadowing, providing failure atomicity for actions. Objects are classified as recoverable or non-recoverable. Non-recoverable objects are somewhat cheaper to handle and can be used by non-critical system tasks, but usage of non-recoverable objects by actions can lead to lapse of consistency. Note that all Unix files are a special case of non-recoverable objects in *Clouds*.

When an action invokes an operation in a recoverable object, a *shadow* version and a *core* version of the object is created. The shadow version is the original permanent version, and the core version is the possibly updated version. If several actions invoke an object in parallel, there is still only ONE shadow and ONE core version. If the synchronization is not automatic, there are possibilities that one uncommitted action will see updates from another uncommitted action, violating the view atomicity requirements (if any). But this is left to the programmer who chose the synchronization strategy.

Every recoverable object has two default entry points called *pre-commit* and *commit*. When the pre-commit entry point is invoked, the object flushes all the updated data in the core version to stable storage, and the commit operation copies the updates to the shadow version and makes the shadow version the permanent version. These entry points can be used by any 2-phase commit protocol.

Like synchronization, recovery comes in two flavors, namely custom and automatic. When an action is run with automatic synchronization, the action management keeps track of all the objects the action touched. When the action terminates successfully, the action management system creates a commit co-ordinator process, that uses the pre-commit and commit entry points of all the objects touched by the action to perform an atomic commit, using the two-phase commit protocol.

Custom recovery is nearly identical, except that the programmer has the ability to redefine the default pre-commit and commit routines in the objects; the user defined routines will be used by the action manager at commit time. The user also has access to the commit routines during normal execution and thus can perform intermediate check points, partial commits and customized features like flushing only certain pages of the object.

Automatic recovery and automatic synchronization guarantee serializability, failure atomicity and view atomicity. Automatic recovery and custom synchronization guarantees failure atomicity and allows the user to use some concurrency control semantically consistent with the application. Custom recovery and synchronization allows the programmed full control of the execution strategy, and the system does not guarantee anything.

8. Programming Support

Systems and application programming for *Clouds* involves programming objects that implement the desired functionality. These object can be expressed in any programming language. The compiler for the language, however, must be modified to generate the stubs for the various entry points, invocation handler, system call interfaces and the inclusion of default systems function handling code (such as synchronization and recovery.)

The language Aeolus has been designed to provide programmers with the full set of powerful features that the *Clouds* kernel supports. Aeolus provides linguistic support for programming *Clouds* objects and allow the composition of objects from sub-objects. Aeolus provides access to the synchronization features (both custom and automatic) and the recovery features of *Clouds*. Though the *Clouds* programmer is not tied to Aeolus, the language is most suited for systems programming as it has been tailored to match the kernel features [LeWi85, Wi85, WiLe86].

Aeolus is the first generation language for *Clouds*. It does not support some of the features found in object-oriented programming systems such as extensive inheritance and subclassing. Providing support for these features at the language level is currently under consideration.

9. Enhancements and Planned Features

The above description of *Clouds* documents the basic features of the distributed kernel for *Clouds*. Presently the following enhancement, applications and features are at various stages of design, implementation and planning.

- An object naming scheme is being developed that creates a hierarchical user naming strategy (like Unix) that is also highly available and robust (through replicated directories).
- Unix and *Clouds* will be inter-operable providing Unix programmers and user with access to *Clouds* features and *Clouds* programmers to use Unix services. Unix machines will be able to execute remote procedure calls to *Clouds* object thus gaining access to all the functionality that *Clouds* provides. In fact the user interface to *Clouds* will be through Unix shells and tools. Similarly *Clouds* applications will make use of the wide variety of programming support tools that are supported by Unix through a mechanisms that provides Unix service for *Clouds* computations. In addition, *Clouds* services will be directly accessible through *Clouds* libraries for other programming languages, such as C++ and

ADA.

- As mentioned earlier, mechanisms for providing object-oriented programming methodology will be provided at the linguistic level, with enhancements in the kernel that will provide performance advantages (such as sharing of code in the classes with its instances).
- Debugging support at the object level, process level and the invocation level will be provided. Techniques that allow the programmer to get a comprehensive view of the distributed and concurrent execution environment are under development.
- A probe system that can track object and process status in the system can provide information about failures, loading, deadlocks and software problems is being developed. This will be used to develop a distributed system monitoring system that will help in reconfiguration of failure and aid in providing fault tolerance. The probe system will also be useful in distributed object level debugging [Da86].
- A distributed database that utilizes the object structure of *Clouds* for elegance and the synchronization and recovery support for concurrency control and reliability is being developed [DaMo86].
- *Clouds* has been designed as a base layer for fault tolerance computing. The systems that will provide fault tolerance and guarantee progress of computation and system response in face of partial system failures are being developed. The techniques include replicated objects, multi-threaded actions, the coupling of the reconfiguration systems and monitoring systems, and usage of dual-ported hardware.

10. Implementation Notes

The implementation of the *Clouds* operating systems has been based on the following guidelines:

- The implementation of the system should be suitable for general purpose computers, connected through widely used networking. Non-homogeneous machines, though not crucial, should be allowed.
- Since the *Clouds* functionality is largely based on object invocation, support for objects should be efficient in order to make the system usable. Also, the synchronization and recovery systems should be efficient.
- Since one of the primary aims of *Clouds* is to provide the substrate for reliable, fault tolerant computing, the base system design should be tolerant to failures and provide adequate support for implementing fault tolerance.
- The system design should be simple to comprehend and implement.

10.1. Hardware Configuration

The hardware chosen for the prototype is commonplace: three VAX-11/750's connected by an Ethernet. The disk units are dual ported, allowing access to the units from two machines, which provides the ability to remount the data from one machine to the other in case of site failures thus increasing availability.

The user interface is not through terminals, but over the Ethernet from Unix mainframes or workstations. This allows easy (software based) reassignment of users in case of site failures.

10.2. Software Configuration

The kernel is implemented in C for portability, and because the availability of C source for the UNIX kernel simplified the task of developing hardware interfaces such as device drivers. Aeolus has been used as the implementation language for *Clouds* utilities.

10.3. Kernel Structure

The kernel is a replicated resident kernel, replicated at all the sites. Logically, the kernel is distributed over several sites and the machine boundaries are invisible. This is achieved by the communication system that provides the low level messaging interface between the replicated kernels. The system control however is completely decentralized, so that failure of individual kernels do not affect the rest of the system [Sp86].

For efficiency considerations, the kernel runs on the native machine and not on top of any conventional operating system. As *Clouds* does not use most of the functionality of conventional operating systems (such as Unix), building *Clouds* on top of a Unix like kernel would have several unacceptable deficiencies, mainly leading to bad performance. Some of the *negative* aspects of using Unix as the base layer would be:

- Unix processes are heavyweight processes: thus process creation and RPC would be expensive.
- An Unix process is tied to one address space. Making a process cross address spaces would involve simulating it through multiple processes and the Unix IPC mechanism, which would involve multiple context switches and other message layer overheads.
- Only one process can execute in an address space, providing serious limitations to intra object concurrency. There are methods that get around this problem, but they are generally complex to implement, unreliable and require substantial overhead.

To avoid these problems the *Clouds* kernel is designed to support the *Clouds* functions on the native VAX and all the performance critical support is implemented at the lowest level in the kernel.

10.4. Object Naming and Invocation

The two basic activities inside the *Clouds* kernel are system call handling and object invocations System call handling is done locally, as in any operating system. Object invocation is a service provided by the kernel for user processes. The attributes that object invocation must satisfy are:

- Location independence.
- Fast, for both local and remote invocations.
- Failed machines should not hamper availability of objects on working sites, from working sites. Also, moving objects between sites, reassigning disk units and so on should be simple.

Location independence is achieved through a capability based naming system. Availability is obtained through decentralization of directory information and a unique search-and-invoke strategy. Speed is achieved by implementing the invocation handlers at the lowest level of the kernel, on the native machine.

When a process invokes an object, it first places the arguments on the stack and executes an *invoke* system call, with the called object capability as the parameter. The capability of the object is unique systemwide, but has no site information. The kernel searches the local object directory to determine if the object is available locally. If it is, then the process address space is switched and the process starts executing in the object that it invoked. (This is achieved by changing the P0 region of the VAX address space by updating the P0 page table registers. The stack of the process is in P1 region, and this space remains the same.)

If the object does not exist locally, the kernel broadcasts a search-and-invoke request. All participating kernels then attempt to locate the object. The successful kernel dispatches a slave process, which copies the arguments from the invoke request to its stack and performs a local invocation on the object. Upon termination, the arguments are send back to the invocation requester, which causes the invocation request to return.

Hash tables, caches, and hint databases are used to add speed both the local searches for objects as well as avoiding the need for all sites to search for objects at each broadcast search-and-invoke request.

10.5. Storage Mangement

The storage management system handles the function required to provide the reliable, permanent object address spaces. As mentioned earlier, unlike conventional systems, where virtual address spaces are volatile and short lived, *Clouds* virtual spaces contain objects and are permanent and long lived [Pi86].

The storage management system stores the object representations on disk, as an image of the object space. When an object is invoked, the object is demand paged into its virtual space as and when necessary. As the invocation updates the object, the updated pages do not

replace the original copy, but have shadow copies on the disk. The permanent copy is updated only when a commit operation is performed on the object. The storage manager provides the support to commit an object using the two-phase commit protocol.

10.6. Action Management

The storage management implements the virtual memory system and the commit protocols, providing the mechanisms for handling the object storage needs. The policies of the action management are not implemented in the storage manager, but rather in the action management system. The action management system implements nested actions for the *Clouds* system by keeping track of the objects touched by an action as well as the success and failure of each action and its subactions [Ke86].

The action manager primarily keeps track of information regarding actions. The action manager is distributed, with the manager at each site keeping information about each action that was started as a top level action at that site. Although an action can span several sites, the action commit is coordinated through the action manager at the site where the action started. As is apparent if the site starting the action fails, the action is doomed to abort, and hence the failure of the coordinating action manager does not hamper the progress of this action.

As discussed previously, when an action terminates, the coordinating manager invokes commit operations on all touched objects, in order to make all updates by the action permanent in an atomic step.

11. Comparisons with Related Systems

Clouds is one of the several research projects that are actively building distributed object based environments. There are similarities and differences between all the approaches, and the area of distributed operating systems are in general not mature enough to conclusively argue the superiority of one approach over the other. In the following paragraphs we document some of the major differences between *Clouds* and some of the better know projects in distributed systems.

One of the major difference between *Clouds* and most of the systems mentioned below is in the implementation of the kernel. Most of the systems implement the kernel as a Unix process[†], while *Clouds* is implemented as a native operating system. In addition, no attempt has been made to build a UNIX interface (e.g., SVID) 'on top of' *Clouds*. *Clouds* is not intended to be an enhancement, or replacement of, the UNIX kernel. Instead, *Clouds* provides a different paradigm from that supported by UNIX (e.g., the UNIX paradigms of 'devices as files', unstructured files, etc,)

[†]The term *kernel* has been used quite frequently to describe the core service center of a system. However when this service is provided by a Unix process rather than a resident, interrupt driven monitor, the usage of the term is somewhat counter-intuitive.

11.1. Argus

Argus is a system developed at MIT, that supports the Argus programming language. The language defines a distributed system to be a set of guardians, each containing a set of handlers. Guardians are logical sites, and each guardian is located at one site, though a site may contain several guardians. The handlers are operations that can access data stored in the guardian. The data types in Argus can be defined to be atomic, and any operation on atomic data types by actions are updated atomically when the action terminates [WeLi83, LiSc83].

Some of the similarities between Argus and *Clouds* are the semantics of nested actions. Both use the nested action semantics and locking semantics described by Moss. This includes conditional commit by subactions and lock inheritance by subactions from the parents as well as lock inheritance by the parents from a committed child. Also the guardians and handlers in Argus have somewhat more than cosmetic similarities to objects in *Clouds*.

The differences include the implementation strategies, programming support and reliability. As mentioned earlier, Argus is implemented on top of a modified Unix environment. This is one of the reasons for the somewhat marginal performance of the Argus system. The programming support provided by Argus is for the Argus language. *Clouds* on the other hand is a general purpose operating system, not tied to any language. Though Aeolus is the preferred language at present, we have used C extensibly for object programming. Any language can be used to program object, after some modifications and patches to the procedures to make them invokable. We have plans to implement more object-oriented languages for the the *Clouds* system. Unlike Argus, *Clouds* is designed to form the base layer for fault tolerant computing, and hence the design decisions of transaction support, search and invoke strategies and so on.

11.2. Eden

Eden is a object based distributed operating system, implemented on the Unix operating system at the University of Washington. Eden objects (called Ejects) use the active object paradigm, that is each object consists of a process and an address space. An invocation of the object consists of sending a message to the (server) process in the object, which executes the requested routine, and returns the results in a reply. The messages use the Berkeley Unix IPC mechanism [Alm83, AlBl83, NoPr85].

Since every object in the system needs to have a process servicing it, this could lead to too many processes. Thus Eden has an *active* and a *passive* representation of objects. The passive representation is the core image of the object stored on the disk. When an object is invoked, it must be active, thus invoking a passive object involves activating it. A process is created and it reads in the passive representation into its virtual space and then performs the required operation. The activation of passive objects is an expensive operation. Also concurrent invocations of objects are difficult and is handled through multithreaded processes or coroutines.

The active object paradigm and the Unix based implementation are the major differences between Eden and *Clouds*. This is also the reason for the performance problems in Eden. Eden also provides support for transaction and replication objects (called Replects). The transaction support and replication was added after the basic Eden system was designed and have some misgivings especially due to manner Unix handles disk I/O. Eden was not designed for fault tolerant applications.

11.3. Cronus

Cronus is an operating system designed and implemented at BBN Laboratories. Some of the salient points of Cronus is the intergration of Cronus functions with Unix functions, the ability of Cronus to handle a wide variety of hardware and the coexistence of Cronus on a distributed set of machines running Unix [BeRe85, GuDe86, ScTh86].

Like Eden, Cronus uses the active objects. This is necessary to be able to make Cronus run on top of Unix, and be an added function to Unix programs. Cronus objects are handled by managers. Often a single manager can handle several objects, by mapping the objects into its address space. The managers are servers and receive invocation requests through catalogued ports. Any Unix process on any machine on the network can avail of Cronus services from any manager, by sending a message to the appropriate manager. By use of canonical data forms, the machine dependencies of data representations are made transparent. Irrespective of the machine types, any unix machine can invoke Cronus objects in a location independent fashion.

11.4. ISIS

ISIS is a distributed operating system, developed at Cornell University, to support fault tolerant computing. ISIS has been implemented on top of Unix. It uses replication and checkpointing to achieve failure resilience. If data object is declared to be k-resilient the system creates k+1 copies of the object. The replicated object invocation is handled by invoking one replica and transmitting the state updates to all replicas. Checkpointing at each invocation is used to recover from failures [Bi85A, Bi85B].

11.5. ArchOS and Alpha

Alpha is the kernel for the ArchOS operating system developed by the Archons project at Carnegie Mellon University. Like *Clouds*, the Alpha kernel is a native operating system kernel designed to run on the Sun-3 computers, networked over Ethernets. The Alpha kernel uses passive objects residing in their own virtual spaces, similar to *Clouds*. ArchOS is designed for real time applications supporting specialized defense related systems and applications [Je85].

The key design criteria for ArchOS and Alpha are time critical computations and not reliability. Fault tolerance is not an issue, as the operating conditions are more susceptible to total failure rather than partial failure. Although the basic system structure resembles *Clouds*,

the different goals have led to significant difference in the implementation techniques and algorithms used in ArchOS.

11.6. V-System

The V operating system has been developed at Stanford University. V is a compromise between message based systems and object based systems. The basic core of V provides light-weight processes and a fast communications (message) system. V messages are similar to object in the sense that the messages are synchronous. The relationship between processes confirm to the client server paradigm. A client sends a request to the server, and the client blocks untill the server replies. In a sense this is similar to a object invocation, as the invocation remains outstanding until the reply is received [ChZw83].

V allows multiple processes to reside in the same address space. Data sharing is through message passing, though shared memory can be implemented through servers managing bounded buffers. The design goals of V are primarily speed and simplicity. V does not provide transaction and replication support, these can be implemented, if necessary at the application level.

11.7. Mach

Mach has been developed at Carnegie Mellon, and looks like a Unix extension. Though Mach is not implemented "on top of Unix" it is implemented to look like distributed Unix. Mach is compatible with Unix at the object code level, that is Mach support all system calls supported by Unix, and hence compiled Unix code can run on Mach. Mach uses the Accent message operating system as its base layer, and Accent provides the communication support. In addition Mach provides support for multiprocessors and distributed systems, memory mapped files, processing abstractions called *tasks* and *threads* [Ac86].

The activity in Mach is carried by tasks and threads. A task is similar to a unix process. It is an address space and an execution environment. A task may be composed of several threads. A thread is a thread of control that can concurrently execute with other threads as a part of the same task, in the tasks address space. Messages are typed data that can be used by threads to communicate, and messages are routed through ports. Ports are addressable through capabilities.

The approaches used by Mach and *Clouds* are conceptually different and it is hard to draw conclusions about the differences in capabilities and usabilities at this stage. Mach however does not provide transaction support.

11.8. LOCUS

LOCUS is a Unix compatible, distributed operating system, operating on SUNs and VAX, connected via an Ethernet. The system supports a high degree of network transparency, permits automatic replication of storage, supports transparent distributed process

execution, and supports nested transactions. LOCUS's primary design goals are transparency, Unix compatability, and high reliability. By contrast, *Clouds* provides Unix interoperability only, and mechanisms for high reliability (rather than integrating high reliability into the kernel). LOCUS's two primary disadvantages are its size, and the performance penalty of an ultra-high reliability kernel. While the overhead for replicated files is relatively low, the overhead caused by system reconfiguration (e.g., when a host is 'powered down') is high [WaPo83, MuMo83].

12. Concluding Remarks

Clouds provides an ideal environment for research in distributed applications. By focusing on support for advanced programming paradigms, and decentralized, yet integrated, control, *Clouds* offers more than 'yet another Unix extension/look-alike'. By providing mechanisms, rather than policies, for advanced programming paradigms, *Clouds* provides systems researchers a adaptable, high-performance, 'workbench' for experimentation in areas such as distributed databases, distributed computation, and network applications. By adopting 'off the shelf' hardware, the portability and robustness of *Clouds* are enhanced. By providing a 'Unix gateway', users can make use of established tools, without the performance penalty of running *Clouds* 'on top of' Unix (or conversely). The gateway also relieves *Clouds* from the necessity of providing emulating services such as provided by Unix mail and text processing.

13. References

[Ac86] Accetta M, et. al. *Mach: A New Kernel Foundation for Unix Development*, Technical Report, Carnegie Mellon University.

[Alm83] G. T. Almes, *The Evolution of the Eden Invocation Mechanism*, Technical Report 83-01-03, Department of Computer Science, University of Washington, 1983.

[Al83] J. E. Allchin, *An Architecture for Reliable Decentralized Systems*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, (Also released as technical report GIT-ICS-83/23,) 1983.

[AlBl83] G. T. Almes, A. P. Black and 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.

[AlMc82] J. E. Allchin 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, Atlanta, GA, 1982.

[BeRe85] J. C. Berets, R. A. Mucci and R. E. Schantz, *Cronus: A Testbed for Developing Distributed Systems*, October 1985 IEEE Communications Society, IEEE Military Communications Conference.

[Bi85A] K. P. Birman and others, *An Overview of the ISIS Project*, Distributed Processing Technical Committee Newsletter, IEEE Computer Society (7,2) October 1985

(Special issue on Reliable Distributed Systems).

[Bi85B] K. P. Birman, *Replication and Fault-Tolerance in the ISIS System*, ACM SIGOPS, Proceedings of the Tenth Symposium on Operating Systems Principles, December 1985 Orcas Island, Washington, (Also released as technical report TR 85-668).

[ChZw83] D. R. Cheriton and W. Zwaenepoel, *The Distributed V Kernel and its Performance for Diskless Workstations*, Proceedings of the Ninth Symposium on Operating Systems Principles, ACM SIGOPS, Bretton Woods, NH, October 1983.

[Da86] P. Dasgupta, *A Probe-Based Fault Tolerant Scheme for the Clouds Operating System*, Technical Report GIT-ICS-86/05 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[DaLe85] P. Dasgupta, R. LeBlanc and E. Spafford, *The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System*, Technical Report GIT-ICS-85/29, 1985 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.

[DaMo86] P. Dasgupta and M. Morsi, *An Object-Based Distributed Database System Supported on the Clouds Operating System*, Technical Report GIT-ICS-86/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[GuDe86] R. F. Gurwitz, M. A. Dean and R. E. Schantz, *Programming Support in the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.

[Je85] E. D. Jensen et. al. *Decentralized System Control*, Technical Report RADC-TR-85-199, Carnegie Mellon University and Rome Air Development Center, April 1985.

[Ke86] G. G. Kenley, *An Action Management System for a Distributed Operating System*, M.S. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01).

[LeWi85] R. J. LeBlanc and C. T. Wilkes, *Systems Programming with Objects and Actions*, Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, July 1985. (Also released, in expanded form, as technical report GIT-ICS-85/03)

[LiSc83] B. Liskov and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust Distributed Programs*, ACM, Transactions on Programming Languages and Systems (53) July 1983.

[Mc84A] M. S. McKendry, *Clouds: A Fault-Tolerant Distributed Operating System*, Distributed Processing Technical Committee Newsletter, IEEE, 1984, (Also issued as

Clouds Technical Memo No:42).

[Mc84B] M. S. McKendry, *Fault-Tolerant Scheduling Mechanisms*, (Unpublished Technical Report), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, May 1984, (Draft only).

[Mc85] M. S. McKendry, *Ordering Actions for Visibility*, Transactions on Software Engineering, IEEE (11,6) June 1985 (Also released as technical report GIT-ICS-84/05).

[McA183] M. S. McKendry, J. E. Allchin and W. C. Thibault, *Architecture for a Global Operating System*, IEEE Infocom, April 1983.

[Mo81] J. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[MuMo83] E. T. Mueller, J. D. Moore and G. J. Popek, *A Nested Transaction Mechanism for LOCUS*, Proceedings of the Ninth Symposium on Operating Systems Principles, ACM SIGOPS, Bretton Woods, NH, October 1983.

[NoPr85] J. D. Noe, A. B. Proudfoot and C. Pu, *Replication in Distributed Systems: The Eden Experience*, Department of Computer Science, University of Washington, Seattle, WA, September 1985 Technical Report TR-85-08-06.

[Pi86] D. V. Pitts, *Storage Management for a Reliable Decentralized Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as Technical Report GIT-ICS-86/21).

[ScTh86] R. E. Schantz, R. H. Thomas and G. Bono, *The Architecture of the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.

[Sp86] E. H. Spafford, *Kernel Structures for a Distributed Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as technical report GIT-ICS-86/16).

[SpBu84] A. Z. Spector, J. Butcher, D. S. Daniels and others, *Support for Distributed Transactions in the TABS Prototype*, July 1984, Technical Report CMU-CS-84-132, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

[WaPo83] B. Walker, G. Popek, R. English, C. Kline and G. Thiel, *The LOCUS Distributed Operating System*, Proceedings of the Ninth Symposium on Operating Systems Principles, Bretton Woods, NH, ACM SIGOPS, pp. 49-70, October 1983. (Available as *Operating Systems Review* **17**, no. 5)

[WeLi83] W. Weihl and B. Liskov, *Specification and Implementation of Resilient Atomic Data Types*, Symposium on Programming Language Issues in Software Systems, June

1983.

[Wi85] C. T. Wilkes, *Preliminary Aeolus Reference Manual,* Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986)

[WiLe86] C. T. Wilkes and R. J. LeBlanc, *Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System,* Technical Report GIT-ICS-86/12, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (To be presented at the IEEE Computer Society 1986 International Conference on Computer Languages).

[Wu74] W. A. Wulf and others, *HYDRA: The Kernel of a Multiprocessor Operating System,* Communications of the ACM, (17,6) June 1974.

[WuLe81] W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/C.mmp, An Experimental Computer System,* McGraw-Hill, Inc., 1981.