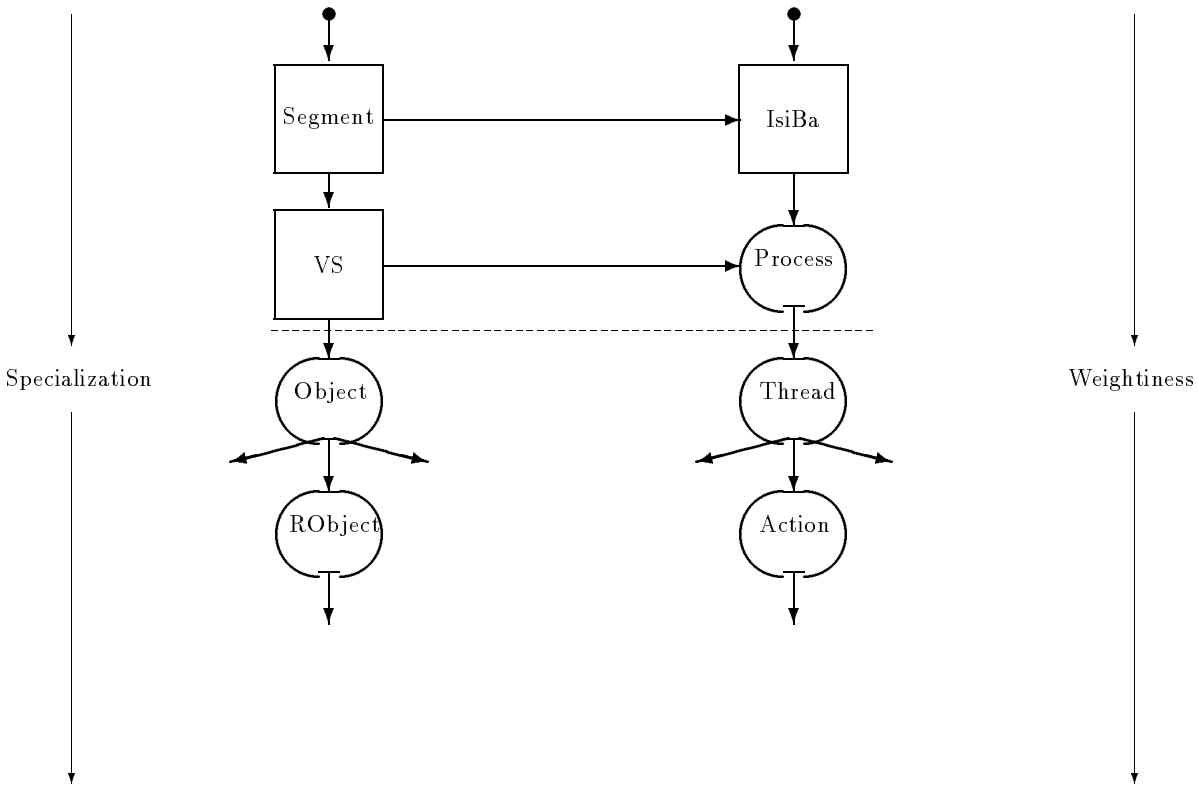

Figure 6: Ra Primitive Composition Spectrums

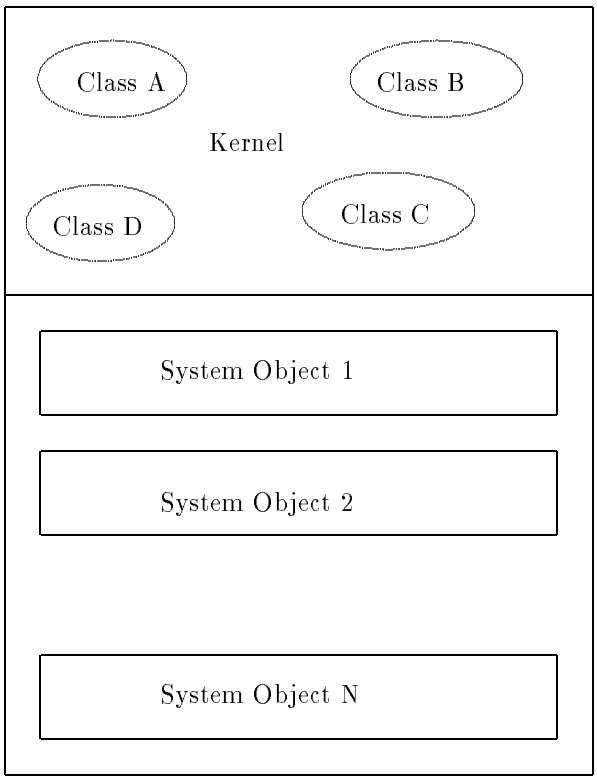

Figure 5: Kernel virtual space

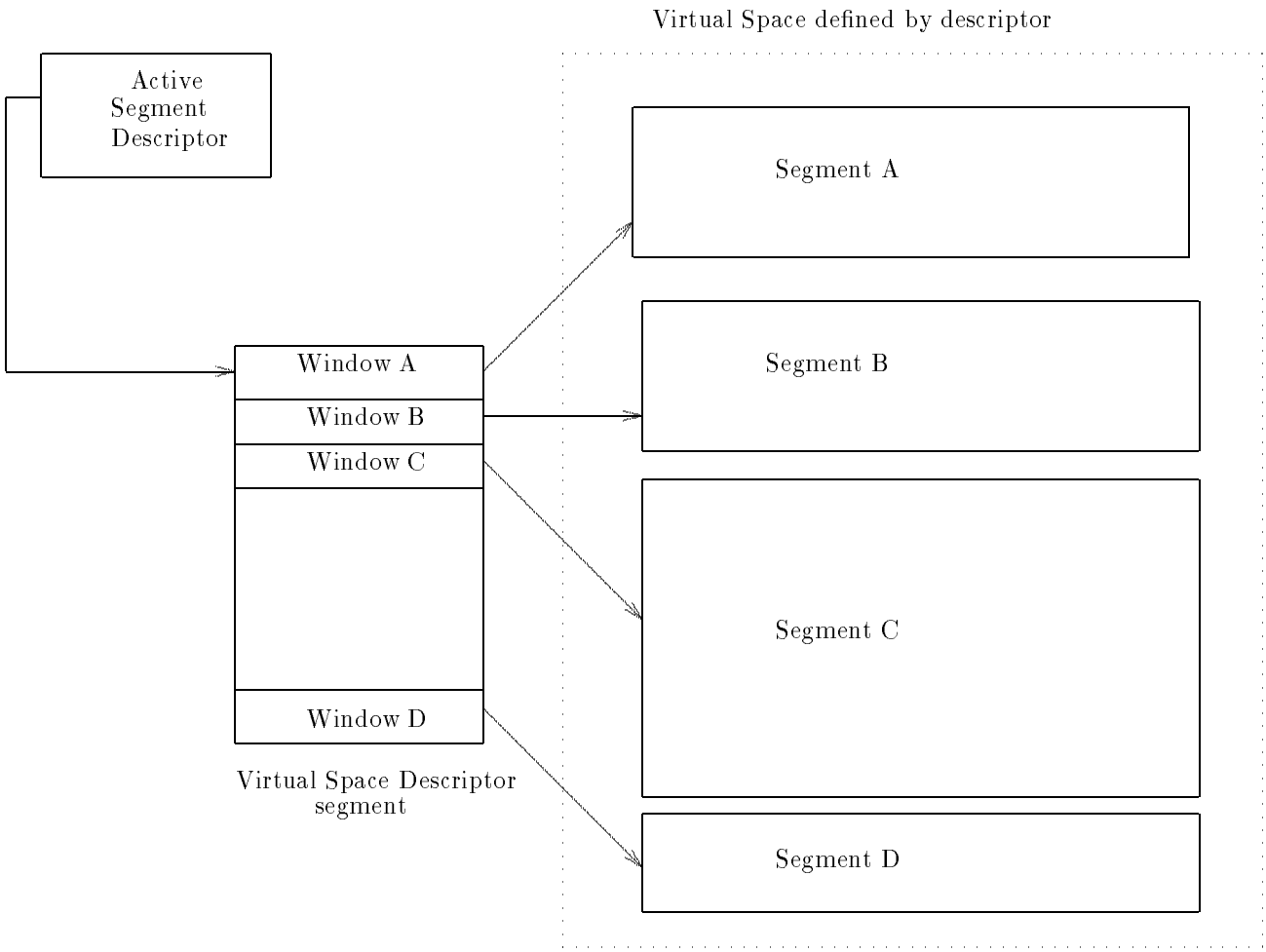

Figure 4: Segment describing a virtual space

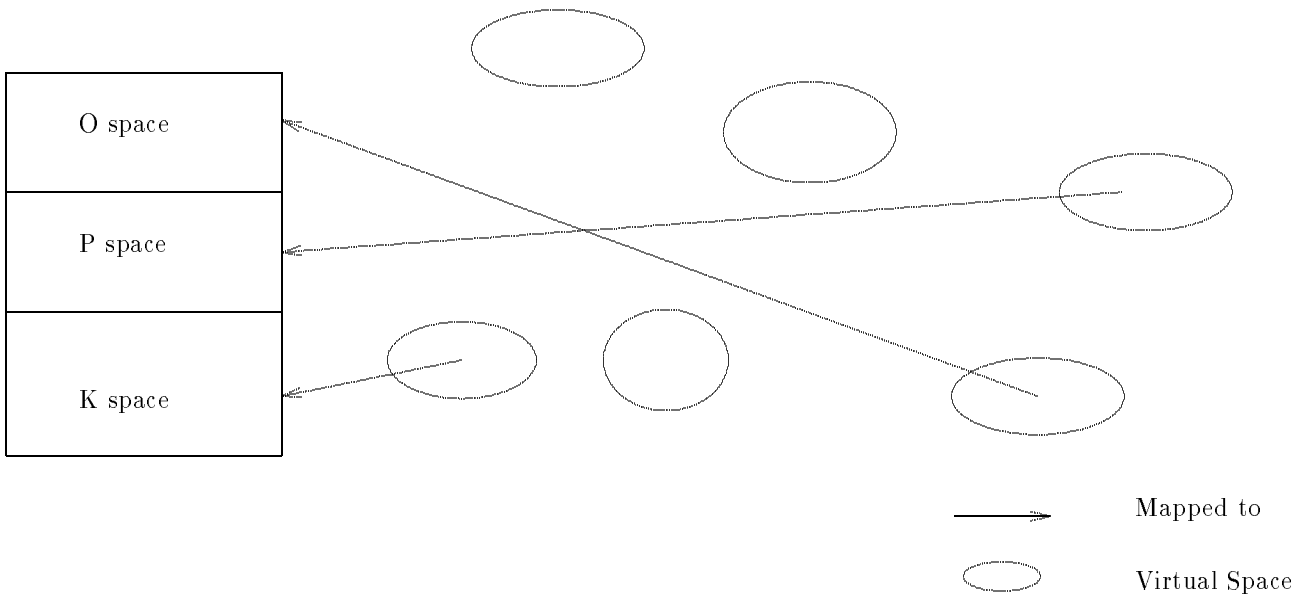

Figure 3: Mapping virtual spaces into hardware spaces

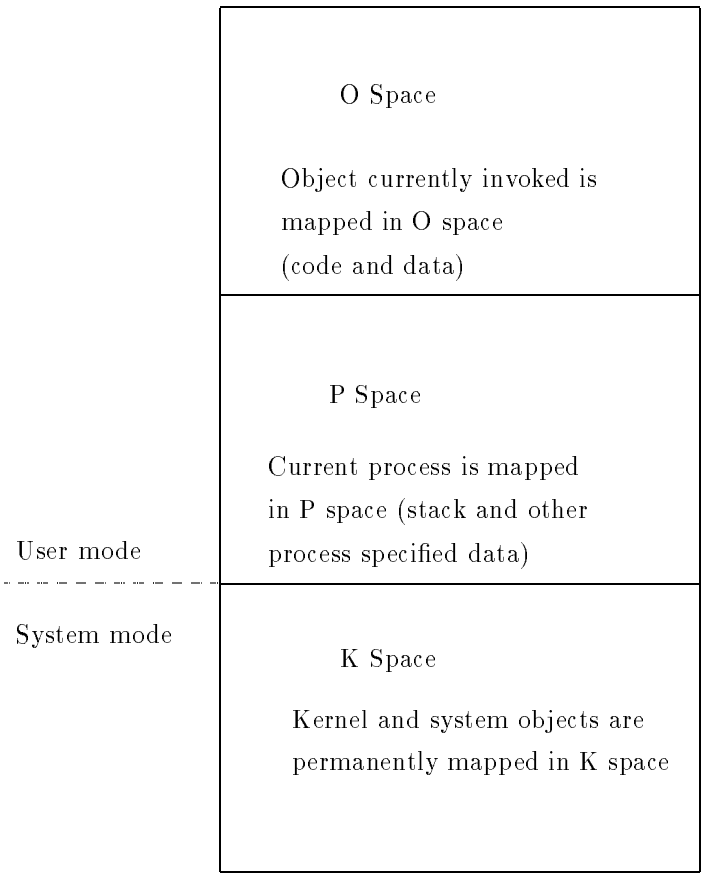

Figure 2: Hardware virtual memory structure

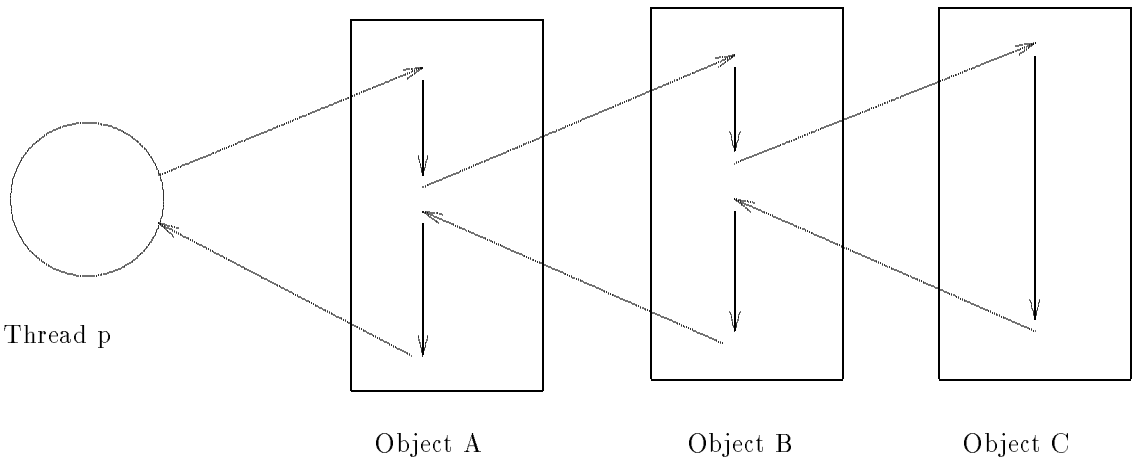

Figure 1: Clouds invocation model

A.5 System Object Class

Load This operation causes a window of the virtual space installed in K space to be mapped to the segment containing the system object code. After the mapping is done, it will usually be necessary to link the system object's code to the rest of the kernel, performing any needed relocation of the object's code. After the above is done, the specified system object is registered in the system object directory thereby making it available for user-level invocations. Register the system object's dispatch table thereby making it available for inter-system object invocations. The registration step may involve the replacement of a previous system object which was already performing the function the new object is to perform. When this is the case, it will be necessary to provide a way of transferring information from the old object to the new one, before the old object is unloaded.

Unload Remove the specified system object from the system object directory and remove its dispatch table. Deallocate the memory occupied by the system object. The unload operation will be performed only if the object being unloaded is not the only one containing information which is vital for the orderly operation of the system.

Query-Update Read or modify current values in the system object directory (used for user-level invocations of system objects) or the system object dispatch tables (used for inter-system object invocations).

time is reached or until it is removed. The associated P and O spaces may be installed repeatedly as a result of this operation.

Remove Remove the specified isiba from the ready list and place it at the specified destination (typically a queue associated with a synchronization variable), or, if the operation is requested by the target isiba itself, remove the isiba from the processor and dispatch a new isiba. The associated P and O spaces may remain mapped if the next isiba to be dispatched has the same attributes (that is, if more than one isiba is executing in the specified virtual spaces).

Query-Update Read or modify current values in isiba context segment or stack segment.

A.3 Synchronization Class

These operations are intended to be used only for synchronizing system objects. See section 8.1.3 for a description of how inter-object synchronization can be achieved in Ra.

Acquire Request control of synchronization variable. Parameters include type of synchronization variable, instance of type, and action desired (such as 'block if unavailable' or 'return immediately'). Examples of possible synchronization variable types include counting semaphores, spin locks, read-write locks, events, etc.

Release Return previously acquired control of a synchronization variable. Parameters are as for the acquire operation. May cause blocked activities to awaiting the synchronization variable to be resumed.

A.4 Device Class

Mount Associate a controlling system object (device driver) with a given piece of hardware. Involves installing the appropriate interrupt handlers and initializing the device.

Unmount Bring the specified device to quiescence and replace the associated interrupt handlers with null handlers.

Query-Update Read or modify current values in the device registers, interrupt vectors, or other device-related data structures.

A Kernel Classes

The kernel classes are introduced in section 6.2. The following sections outlines the operations associated with each class.

A.1 Virtual Space Class

Activate Attach the segment containing a description of the specified virtual space (the virtual space descriptor) into K space and prepare to map the virtual space into the P or O space. Allocate segment and page mapping structures.

Deactivate Deallocate segment and page mapping structures. Detach the virtual space descriptor from K space.

Install Map the specified virtual space into P or O space. Entails constructing the appropriate segment and page tables and, possibly, replacing the currently installed virtual space. May require activating the virtual space.

Attach Segment Form an association between a specified segment and a range of addresses in a specified virtual space. Update the virtual space descriptor to reflect this association. May be performed either dynamically when the specified virtual space is either mapped in P or O space, or passively when the virtual space is not mapped. The specified segment may be either activated or deactivated.

Detach Segment Dissociate the specified segment with the range of addresses in the specified virtual space. Note that a single segment may be mapped into a virtual space more than once so that detaching a segment from one location may not completely remove the association of the segment and virtual space. May be performed either dynamically or passively.

Query-Update Read or modify current values in virtual space descriptor.

A.2 Isiba Class

Install Request that the specified isiba be assigned a physical processor. If none are currently available, place the isiba on the ready queue. A time quantum, a maximum execution time, and a priority level are specified with this call. The ready list is maintained as a priority queue. The isiba must have associated P and O space attributes, a scheduler attribute, and a privilege level. The isiba will receive quantum units of processor time until the maximum execution

[RAK87] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Hardware support for distributed shared memory. Technical Report GIT-ICS-87/41, School of ICS, Georgia Institute of Technology, November 1987.

[RR81] R. F. Rashid and G. G. Robertson. Accent: A communication oriented network operating system kernel. In *Proc. of the Eighth Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[Spa86] E. H. Spafford. *Kernel Structures for a Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Tech, Atlanta, Ga, 1986. (Technical Report GIT-ICS-86/16).

[STB86] R. E. Schantz, R. H. Thomas, and G. Bono. The architecture of the Cronus distributed operating system. In *Proc. of the 6th Int'l. Conf. on Distr. Computing Sys.*, May 1986.

[TM81] A. S. Tanenbaum and S. J. Mullender. An overview of the Amoeba distributed operating system. *Operating System Review*, 13(3):51–64, July 1981.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Commun. of the ACM*, 17(6):337–345, June 1974.

[WL86] C. T. Wilkes and R. J. LeBlanc. Rationale for the design of Aeolus: A systems programming language for an action/object system. In *IEEE Computer Society 1986 International Conference on Computer Languages*, 1986.

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Trans. on Software Eng.*, SE-11(1):43–58, Jan. 1985.

[Ber87] José M. Bernabéu Aubán. *Locating Resources in Distributed Systems.* PhD thesis, School of Information and Computer Sciences, 1987. PhD Research Proposal.

[CD87] Raymond Chen and Partha Dasgupta. Consistency preserving threads: An aprroach to atomic programming. Technical Report GIT-ICS-87/43, School of ICS, Georgia Institute of Technology, December 1987.

[CZ83] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. *Operating Systems Review*, 17(5):128–140, October 1983.

[DLA88] P. Dasgupta, R. J. LeBlanc, and William F. Appelbe. The Clouds distributed operating system: Functional description, implementation details and related work. In *Proc. of the 8th Intl. Conference on Distributed Computing Systems*, number GIT-ICS-87/42. IEEE, June 1988.

[Gif79] D. Gifford. Weighted voting for replicated data. In *Proceedings of 7th Symposium on Operating Systems (Pacific Grove, California).* ACM, Dec 1979.

[HMSC87] Roger Haskin, Y. Malachi, Wayne Sawdon, and Greg Chan. Recovery management in Quicksilver (extended abstract). In *Proc. Eleventh ACM Symp. on Operating Systems Principles*, pages 107–108, November 1987.

[J+85] E. D. Jensen et al. Decentralized system control. Technical Report RADC-TR-85-199, Carnegie Mellon University and RADC, Pittsburgh, PA, April 1985.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proc. 5th ACM Symp. Principles of Distributed Computing*, pages 229–239. ACM, August 1986.

[Lis83] Barbara H. Liskov. Guardians and actions: Linguistic support for robust distributed programs. *TOPLAS*, pages 381–404, July 1983.

[Mye82] Glenford J. Myers. *Advances in Computer Architecture.* John Wiley, 1982.

[Pit86] D. V. Pitts. *Storage Management for a Reliable Decentralized Operating System.* PhD thesis, School of Information and Computer Science, Georgia Tech, Atlanta, Ga, 1986. (Technical Report GIT-ICS-86/21).

manager inspects the lock segment. If there are any waiting threads which should be granted the lock, a note is made in the lock segment, and the isiba scheduler is called to schedule the isibas unblocked by the lock release operation.

The exact semantics of the locking schemes used are specified by the synchronization manager and not encoded in the kernel. Different synchronization primitives can be implemented and experimented with by modifying the synchronization manager. By separating the *mechanisms* of implementing synchronization variables from *policies*, by placing the actual synchronization queues and data in the object, and by entrusting system objects with the task of implementing the policies, we achieve a flexible synchronization mechanism which we believe will be easier to implement, maintain, and modify.

9 Conclusions

The Ra kernel is a minimal kernel for the Clouds operating system. The Ra design is modular and Ra supports extensibility. Ra uses the object-based paradigm supported by Clouds to implement Clouds, providing the first instance of an application of the paradigm.

The implementation of Ra is under progress. Ra is being implemented on a network of Sun-3/60 workstations. The workstations will run Ra, and the partitions will reside on a file server running Unix. The partition code will run as a Unix application program.

All user application code and user services will be implemented as a part of the Clouds system, and will run on top of Ra. User interfaces are not designed yet, but preliminary plans are to use X-windows and a Unix like shell for user interactions. Another scheme under consideration will use Unix as a front end user interface that allows access to Clouds. In this scheme the user workstation will run Unix, a set of rack mounted compute servers will run Clouds and a set of file servers will host the partitions, which can run under Unix, or in native mode under Ra.

References

[AAJK87] Mustaque Ahamad, Mostafa H. Ammar, José M. Bernabéu, and M. Yousef A. Khalidi. Locating resources in a distributed system using multicast communication. Technical Report GIT-ICS-87/44, Georgia Institute of Technology, December 1987.

[ABA88] Mostafa H. Ammar, José M. Bernabéu Aubán, and Mustaque Ahamad. Using hint tables to locate resources in distributed systems. In *IEEE INFOCOM'88*, 1988.

8.1.2 Threads

Threads are the only visible active entities in Clouds. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object by invoking an entry point in the object. Threads may span machine boundaries.

A thread is implemented by one or more processes. A process is a virtual space with one isiba associated with it, plus other process-specific information – data that is logically associated with the activity rather than with an object, such as controlling terminal, thread specific information, and action specific information. The process contains *stack segments*, one for each object invocation. These segments are created and destroyed in a LIFO manner. A process is created for each thread's visit to a machine.

Process management is handled by the *process manager* system object. A thread is created as follows: the process manager object creates a process virtual space that include a stack segment plus other thread related information. Next, it creates a segment and initializes it as an isiba context segment (note that only the machine independent part is initialized; the kernel manages the machine dependent part). The process manager sets the scheduling attributes of the isiba, sets the stack segment in the process virtual space as the isiba stack, and initializes the P space attribute to the process virtual space and the O space attribute to the object virtual space to be invoked.[3] The address to start execution is included in the initialization of the machine independent part of the control segment. Lastly, the process manager calls the isiba scheduler to make this isiba runnable.

8.1.3 Synchronization

The *synchronization manger* is responsible for intra-object synchronization. Each object contains one or more *lock* segments. A lock segment is a read-write segment (which is protected for system-mode access only). It contains lock information plus associated queues that indicate which threads (and isibas) are waiting on a lock, if any. A thread running in an object requests a lock operation as follows: the thread performs a kernel call giving the synchronization manager as a parameter, plus lock and mode requested. The kernel routes the call to the synchronization manager. The synchronization manager examines the appropriate lock segment in the object, and determines whether or not to grant the lock request. If it decides that the isiba has to block waiting on the lock to be released, it makes a note of this fact in the lock segment and calls the isiba scheduler to block the thread. When a threads releases a lock, the synchronization

[3] A thread starts its life by invoking an object. The O space attribute of the isiba is set to point to the virtual space of this object.

8.1.1 Objects

Clouds objects are structured Ra virtual spaces which which are mapped into the O-space when invoked. The segments constituting an object are distinguished by various attributes which are based roughly on whether the segment contains code or data. For example, code segments are read-only, while a segment holding recoverable data has a recovery attribute which enables the kernel to locate the partition responsible for handling page faults on the segment.

Objects are invocable which provides a structured means for an execution to enter and leave the virtual space associated with the object. Object invocations may be nested and thus consist of a call which transfers control to the specified operation in the desired object, and a return which transfers control back to the original calling object. A system object called the *invocation manager* is responsible for installing the appropriate object virtual space on object call and return. Consider the following typical scenario: an isiba executing in a user object invokes an operation on another object. A user-level object invocation appears to the Ra kernel as a request on a system object, the invocation manager. The kernel protected procedure call mechanism verifies that the specified system object, the invocation manager in this case, appears in the system object directory. The invocation system object is then called with four parameters: the invoking object, the object to be invoked, and a segment containing the object-call parameters, and the desired operation. When an object is found locally, the invocation manager installs the object's virtual space, and "returns" by transferring control to the location in the newly installed object virtual space corresponding to the desired operation. A similar scenario is followed on object return. (Section 8.1.2).

If the object to be invoked is not found on the local node, the invocation manager may solicit the help of other system objects to locate the desired object. This may be done by using the partition manager as described in Section 5.4 to activate the segment which contains the object's descriptor, or it may involve a more complex object location policy[2] performed by an *object locator* system object. Both the partition manager and the object locator would typically perform their function by communicating with their counterparts on remote nodes. Once the desired object has been identified on a remote node, the object is brought (whole or partial) from the remote node, installed in the O space and execution is resumed. The partitions in charge of mediating accesses to the invoked object segments maintain the coherency of these segments among the various nodes using them. The coherency scheme is encoded in the partitions, and different coherency schemes can be used to maintain the consistency of different segments.

[2]Such as one of those described in [Ber87].

segment. Virtual spaces abstract the notion of addressing domain and provide a means for manipulating virtual memory page and segment tables.

Proposed entities appear above the dotted line and are composed of the Ra primitive abstractions. These entities may be viewed abstractly as falling within one of two *spectrums*, either the computation or storage spectrum. Entities within the storage spectrum represent successive *specializations* of the abstract notion of store. Segments, as we have said, are the most primitive. Virtual spaces are composed of segments. Clouds objects are composed of a structured virtual space and an invocation mechanism. Objects can be further refined to produce recoverable objects[WL86] by placing additional restrictions on the behavior of object storage. One can imagine the spectrum continuing with further specializations of recoverable objects. Similarly, variations of entities within the spectrum are possible. Given the virtual space primitive abstraction, various forms of object can be composed by using alternate invocation mechanisms or structure.

Parallel to the storage spectrum is the computation spectrum. Isibas represent the most basic form of activity. Clouds processes are then composed of one or more isibas in conjunction with a virtual space. Next appears the Clouds thread. Threads are composed of a possibly distributed collection of processes. The computation spectrum can also be viewed as a specialization hierarchy. Actions[WL86] are threads which make certain guarantees about the recoverability and atomicity of computations which run on their behalf. As with the storage spectrum, variants of the basic entities can be proposed and may exist concurrently within the system.

Viewing entities in Clouds/Ra as existing on these spectrums provides a uniformity and structure to our system and appears as a natural result of Ra's extensibility. System objects and kernel classes are the mechanisms by which entities are composed. The storage and computation spectrums also provide a structured approach for exploring the design space of operating system components.

8 Using the Primitives

In this section we describe the implementation of some of the basic functionality of the Clouds operating system using Ra's primitive abstractions.

8.1 Implementing Clouds

An overview of the Clouds system was given in Section 2. Here we show how the different Clouds components can be built using the Ra toolkit of primitives.

the familiar abstract data types or modules of software engineering. We suggest that they be viewed rather as *mixin classes*. Mixins are special object classes used in various object-oriented programming systems which are designed, not to be instantiated, but rather, to be used to compose new object classes. In this sense, they support a powerful form of multiple inheritance. In Ra, system objects *import* or inherit specified kernel classes. System objects may invoke operations only on kernel classes which have been explicitly imported. Code and data from other kernel classes are protected at the linguistic level[1] and may not be accessed. Run-time protection will probably be too costly in general on traditional hardware although it should be possible to request hardware supported protection while a system object is being debugged.

There are five collections of kernel services. These services are defined as classes. These are the *Virtual Space class, Isiba class, Synchronization class, Device class,* and *System-Object class.*

The virtual space class provides system-objects with functions and data structures that manipulate segments, windows and virtual spaces. Using this class, we can build services that create and delete objects, provide invocation support and so on. The isiba class provide some isiba control primitives. Controlling the computation in the operating system, creating threads and controlling their execution can be acheived through this class, The synchronization class provides isiba synchronization primitives that are useful for locking and recovery control. The device class provides access to the device specific registers and interrupt vectors. These are used by system-objects that implement device services and need to install interrupt handlers. Finally, the system-object class, allow system objects themselves to control the behavior of other system objects, that is installation and repeal of services can be handled.

A more detailed description of the classes and the operations they support is in the appendix.

7 Computation and Storage Spectrums

The various components of the Ra kernel and proposed components of the Clouds operating system may be viewed as being *compositionally* related in the manner displayed in Figure 6. The components above the dotted line represent the three kernel primitive abstractions: segment, isiba, and virtual space. The way in which the kernel primitive abstractions are composed was described in detail in Section 5. Briefly, segments abstract the notion of store and are the most fundamental abstraction. Isibas are constructed from two segments and by abstracting the notion of computation. Virtual spaces are composed by associating segments with ranges of virtual addresses and then storing the information describing the association in yet another

[1] That is, at compile time.

desired goal of a minimal kernel.

System-objects serve as intermediaries for providing system-services. Many such services may be provided by user-level objects and system-objects should be introduced only when efficient access to protected kernel facilities is required. In the spirit of a minimal kernel we have argued that the kernel should contain only those facilities which cannot be located outside the kernel. We make a similar argument concerning system-objects. System-objects should contain only those facilities which cannot be reasonably provided by user-level objects. As an example, a print spooler should probably not be made a system-object. The spooler requires only services which can reasonably be expected to be provided by system-objects such as synchronization and bulk data transfer. A printer device driver, however, is a likely candidate to be made a system object. A device driver will need to access device registers and will need to accept and post interrupts.

System-objects may also be loaded dynamically to introduce and remove services. Thus Ra supports *plug-in* system-objects. There are many advantages to such a feature. First, such dynamic loading allows kernel extensions to be introduced in the form of system-objects without the need for recompiling the kernel and rebooting the system. Second, various configurations of system-services can be loaded based on the capacity and pattern of use of a given machine. For example, a small micro version of Ra/Clouds need not support the full range of system-services provided by larger systems. Third, dynamic loading of system-objects simplifies developing and debugging new system-services. Fourth, multiple versions of a particular service can coexist for comparison and analysis under the same load conditions. For example, two object location mechanisms may be implemented and executed in parallel to compare their efficiency.

Since system-objects may be introduced and removed dynamically, special kernel operations provide a means for facilitating the orderly transfer of responsibility for a given system-service from one system-object to another. Such a transfer of control may, in general, involve a complex protocol or it may, in special cases, be as simple as a change of registry.

System-objects are mapped into the kernel or K space when loaded and remain present, accessible to all processes and objects, until they are unloaded. System-objects are relocatable so that they may be loaded in differing places but this relocation is only done at load time, which is intended to be infrequent.

6.2 Kernel Classes

System objects "see" the kernel through kernel classes. Kernel classes are collections of kernel data and procedures to access and manipulate that kernel data. Thus kernel classes are like

6 Mechanisms for Achieving Extensibility

Ra *system-objects* provide structured access to kernel facilities which are grouped into collections or modules called *kernel classes* (Figure 5). The operations available on the Ra kernel classes collectively constitute the Ra *virtual machine*. Ra's system-objects and kernel classes together form the basis of Ra's claims of extensibility. This section describes system-objects and kernel classes, and how they will be implemented, and provides examples of their use.

6.1 System-Objects

Certain system-objects are designated *essential*. Essential system-objects provide services that are necessary for the system to function yet that admit to a variety of implementations. Virtual memory page placement and replacement policies are good examples of services which must be provided by one or more essential system objects. Such functions certainly do not belong in a minimal closed kernel. Many different policies are possible and the correct choice of policy may depend on the configuration and pattern of use of the system in question. Thus, on the one hand, we have a service which must not be placed in the kernel, and, on the other hand, we have a service which is absolutely essential for the correct operation of our system. Thus the introduction of essential system-objects. By moving such services outside the kernel the minimality of the kernel is retained, as well as the support for extensibility and flexibility is maintained.

System-objects may be viewed as a structured, controlled means of access to kernel functions and data structures. Viewed from a different perspective, system-objects represent a route to the *efficiency* of the kernel. They allow the system-object programmer access to trusted system functions with minimum overhead. System-objects are themselves trusted objects; that is, they are assumed to execute correctly. An error in a system-object may corrupt the entire processor which the system-object is executing on. System-objects provide a traditional object interface to the rest of the system and they are invoked, at the linguistic level, in the same manner as other user objects. Invocations on system-objects are actually implemented by means of protected procedure calls on the kernel for efficiency.

System-objects implement (encapsulate) policy and provide access to kernel efficiency when needed. Most systems provide a set of kernel primitives which can be combined in some fashion to provide new services. However, often the facilities for combining kernel primitives are restrictive and inefficient. For example, if a system-service is implemented by a traditional user-level process each access of that system-service will require an expensive process context switch. System-objects represent a means of providing both extensibility and efficiency while maintaining our

kernel). Deactivating the segment informs the partition that further access to the segment is unlikely in the near term. Reading and writing the contents of a segment are considered operations on the associated virtual space.

Alternative storage hardware such as write-once optical store can be easily integrated into Ra by introducing an appropriate segment attribute and by implementing a partition to support that attribute. Partitions are themselves implemented as system-objects. References to device drivers that access physical storage are encapsulated in partitions.

Segments may also be *remotely activated*. Remote activations are managed by a local partition called the *mediating* partition. The mediating partition receives the activation request and then cooperates with a remote partition to activate a copy of the segment locally. Segments may be shared remotely as well as locally and thus there may be more than one mediating partition for each segment.

Since segments are named by a system-wide unique sysid, remote activations require that the desired segment first be *located*, that is, the node actually containing a permanent copy of the segment be identified. (Typically, each partition is itself associated with either a node where a permanent copy of its segments is to be found or a method for consulting all or a subset of the nodes containing permanent copies of segments). This allows the system to use different location methods depending on the characteristics of the segments to be located (for a more thorough discussion of the performance issues related to resource location in distributed systems see [Ber87, ABA88, AAJK87]). A policy-making object called the *partition manager* may be consulted to determine the appropriate partition to be used for a particular activation request.

Distributed shared memory [RAK87, LH86] is implemented in partitions that support segments with attribute *dsm*. It an extension of the shared memory model to distributed systems. A set of operations are provided on shared segments and the partition maintains the state of the shared segment though a coherence protocol. The partitions take on the responsibility of acquiring the current copy of a segment on demand.

Distributed shared memory is the mechanism used to run Clouds in the workstation environment. Instead of sending the computation to a remote node, the required segments are brought from the remote node to the local machine. The remote node is the repository of the object (or the set of segments).

Note that the space attributes may change frequently while the isiba is executing.

An isiba may be removed by another isiba executing with appropriate privileges or an isiba may remove itself. In either case, the destination of the removed isiba must be specified. Typically, this will be a queue representing a synchronization variable or event. When an isiba is removed by another isiba it may only need to be removed from the ready queue. If the isiba removes itself (i.e. *blocks*) or if the isiba is removed by a concurrently executing isiba on a multiprocessor implementation of Ra, then another isiba must be dispatched in addition to the designated isiba being removed.

Isibas may be used as demons within the kernel or they may be associated with a virtual space to implement "heavier" forms of activity such as Clouds threads. Using isibas to implement threads is illustrated in Section 8.

5.4 Partitions

Segments are primitive in Ra but supporting certain segment attributes such as recoverability may require extensive policies. These policies should be kept out of the kernel. Also, it is not possible to predetermine all segment attributes. Thus we need a mechanism for handling attributes which might change as the operating system evolves. For this reason, a segment handler and repository called the *partition* is introduced. The partition is responsible for realizing, maintaining and manipulating segments. Although, partitions are a Ra primitive abstraction, and the existence of partitions is crucial to the operation of Ra, the partitions are not part of Ra. A partition handler is a Ra system-object. If the partition is stored at a remote site (data-server) the server runs a cooperating part of the partition code, to get the segment.

Each segment is maintained by exactly one partition and a segment is said to *reside* in that partition. Initially, a segment resides in the partition in which it was created. The partition in which a segment resides is referred to as its *controlling* partition. All operations on a segment with the exception of accessing its contents are performed by its partition and are initiated by requests to the partition.

Several operations are possible on segments via their controlling partitions. Segments may be created and destroyed and they may have their mutable attributes changed as described previously in Section 5.1. Segments may also change their partition of residence, that is, they may *migrate*. Migration may be desirable to improve the locality of reference to the segment and its contents. Finally, segments may be explicitly *activated* and *deactivated*. Activating a segment prepares the controlling partition for further activity relating to the segment and returns summary information about the segment and its attributes to the calling entity (typically the

machine registers, are called *transient*. When an isiba is inactive, all attributes represent the most recent state of the isiba. (This is analogous to the PCB of a process in a conventional kernel).

An isiba's stack segment contains the call stack of the computation. The stack is represented by a separate segment because it is a user accessible component of the isiba. Placing the isiba's stack in its own segment also allows a simple implementation of the model of computation in which a single activity may execute in multiple address spaces. A distinct stack segment is allocated to the isiba for each virtual space in which it executes. More than one stack segment may exist at any instant if the transfers from one virtual space to another are viewed as nested traversals. However, only one stack segment is considered the *current* stack of the isiba.

Like segments, isibas are uniquely identifiable by a sysid. Isibas have four required attributes, *scheduler*, *privilege*, *O space*, and *P space*. The scheduler attribute identifies a scheduler system object to invoke when a high-level scheduling decision concerning the isiba must be made. Privilege is a transient attribute that describes the processor mode (kernel or user) in which the isiba is executing. Privilege is used to define resources accessible to the isiba at any given moment. Finally, each isiba has two associated space attributes which identify the virtual spaces which must be mapped for the isiba to run. Either or both of these attributes may be null. If both attributes are null, then it is assumed that the isiba will execute entirely within the kernel address space (K space) which is always visible and need not be mapped for the isiba to run. Typically, the context segment of the isiba will be mapped into the kernel space and the stack segment into the P space; however, other schemes are possible.

When an isiba is installed, two parameters are provided to the kernel: a time quantum, and a maximum execution time. The isiba is placed on a queue along with other isibas waiting to execute (the *ready* list). A kernel entity known as the *short-term scheduler* (STS) periodically receives clock interrupts. On each interrupt, the STS determines if the currently executing isiba's time quantum has expired. If the isiba's time quantum has not expired, it continues execution. Otherwise if the cumulative execution time is less than the maximum, it is enqueued on the ready list once again and the STS removes the isiba at the head of the ready list and prepares the processor for it to run. If the cumulative execution time of an isiba has exceeded its maximum execution time, the STS calls the high level scheduler associated with the isiba, which will decide what to do with it. This procedure, called *dispatching* the isiba, may require the kernel to remap the P and O spaces. If the newly dispatched isiba has the same P and O space attributes as the previous isiba then no remapping is required. In this case, the two isibas are said to be executing *concurrently* within the same address spaces. If both space attributes of the new isiba are null then it runs, by default, in the kernel address space. The stack and context segments of the isiba must be attached to one of its associated spaces for it to execute.

When a virtual space is activated, the kernel allocates segment and page map tables in anticipation of the virtual space being mapped into the hardware P or O spaces. These structures are initialized according to the window descriptions in the virtual space descriptor. Attach and detach operations will cause these mapping tables to be updated as appropriate. A virtual space must be activated before operations such as attach and detach may be performed on it. Such operations will activate the virtual space implicitly when required. The companion operation, *deactivate*, updates the descriptor to reflect any changes necessary, detaches the descriptor segment, and deallocates segment and page maps used by the virtual space.

Install, maps the virtual space into the P or O space. If the specified virtual space has been activated, then the kernel simply needs to update the hardware segment and page tables to reflect the corresponding tables already constructed and maintained by the kernel. If the specified virtual space has not already been activated, the kernel will activate the space, and then install it. Segments may continue to be attached and detached while a virtual space is installed. In this case, the attach or detach operation will update the virtual space descriptor as well as modify the hardware mapping tables. A segment is attached *dynamically* when it is attached to an installed virtual space, and *passively* when it is attached to an uninstalled space.

There are three varieties of virtual spaces, depending upon where a virtual space is installed. The virtual space which maps into the hardware K space is called the *kernel space* or *kernel*. Similarly, a virtual space which is mapped into the P space (or which will be mapped into the P space) is called a *process virtual space*. A virtual space which maps into the O space is called an *object virtual space*.

5.3 Isiba

Ra *isibas* are an abstraction of the concept of computation or activity and are intended to be the lightest-weight unit of computation.

An isiba may be assigned a virtual processor by the operation *install*. The processor is relinquished when the *remove* operation is performed on the isiba. An isiba is said to be *active* after it is installed and before it is removed; otherwise, it is *inactive*. Isibas are composed of two segments: a *context* segment and a *stack* segment. Each segment contains data describing the activity of the isiba it composes.

The context segment contains the kernel accessible state information describing the computation such as the program counter and machine register contents. These values can be accessed as attributes of the isiba. When an isiba is active, some of its attributes may not reflect the current state of the isiba, but only a recent state. These attributes, such as the values of the

segments. Thus, a particular segment may be read-write accessible when it is associated with one virtual space, but read-only when it is associated with another virtual space.

A virtual space is composed (or built) by a sequence of *attach* operations on segments. Attaching a segment to a virtual space associates a range of addresses in the virtual space to a range of offsets in a segment. The attach operation requires as parameters a virtual space descriptor, a segment-id, a length (of the region to be defined by the association), a starting virtual address in the space, and a starting offset into the segment. The attach operation defines a one-to-one mapping between virtual space addresses and segment locations such that virtual addresses starting at the designated address are associated with segment locations starting at the offset and continuing for the length specified. That is, if virtual address x is associated with segment offset y, then virtual address $x + 1$ is associated with segment offset $y + 1$. Note that a virtual address range defined by an attach operation must fall entirely within the specified segment.

The mapped ranges in a virtual space may not overlap, but ranges in the virtual space may remain unmapped. Thus, a single virtual address may not resolve to two or more segment locations but virtual spaces may have "holes" in them. Virtual spaces may share segments (or ranges of locations in segments). No synchronization or access control is implied by an attach operation. Also, a segment may be mapped more than once to a single virtual space; that is, two or more windows in a single virtual space may refer to a single segment. Multiple mapping can happen in two ways: either two or more windows in the space refer to disjoint regions of a single segment, or two or more windows in the space refer to overlapping regions of a single segment. In the second case, distinct virtual addresses in the same space resolve to the same segment location.

The *detach* operation undoes the work of attach and removes the association between a range of virtual addresses and range of offsets in a segment.

The *activate* operation on virtual spaces conceptually provides the kernel with the information in the virtual space's descriptor. Since the window descriptor is stored in a segment, the virtual space activate operation can be described in terms of operations on segments. A virtual space is *activated* by supplying the kernel with the sys-id of the segment that contains the virtual space descriptor. The kernel then communicates with the segment's controlling partition (see section 5.4) to activate the descriptor segment if necessary and then attach it to the kernel virtual space. In this way, the descriptor's contents become visible to the kernel. Once the descriptor is attached, some of its information is stored redundantly in kernel data structures for efficiency.

Activate may be seen as a notification to prepare for further activity on the virtual space.

Segments are explicitly created and persist until explicitly destroyed. They may be of arbitrary length and their size may vary during their lifetime. A segment has a set of attributes, which include length, types, and storage methods. Certain attributes are considered *immutable* and remain as set throughout the life of the segment. Other attributes such as length, and whether a segment is sharable , are considered *mutable* attributes and may be altered during the life of the segment.

One major characteristic of a segment is its *storage* attributes. These attributes will make guarantees about the behavior of accesses to the contents of a segment in the presence of failures. For example, a *volatile* segment is guaranteed to have zero-filled contents on first access, following a failure of the system that hosts the segment. Changes to the segment are guaranteed to persist up to the first system failure following the modification or the next modification. A *recoverable* segment guarantees that changes persist across system failures. Thus segments with attributes refer to particular methods of using hardware to support storage.

In addition to providing naming and addressing for bytes in a single-level store, segments may be shared between virtual spaces. That is, a given location in a segment may be simultaneously mapped by two or more virtual addresses so that concurrently executing isibas may share a common store. Activities may then communicate using any agreed upon protocol through this shared store. Some variations of sharing are possible and these are described in Section 5.2.

A special form of segment called a *fast segment* requires minimal partition support and is highly optimized. Fast segments are volatile and may not be shared. Because of these restrictions, they may be accessed very efficiently and are used in situations where high performance is required. One common use of fast segments is for passing parameters during object invocation.

5.2 Virtual Space

Ra *virtual spaces* are an abstraction of the notion of an addressing domain and provide a metaphor for manipulating virtual memory mapping mechanisms. Conceptually, a virtual space is a collection of sub-ranges of a set of ordered segments.

A Ra virtual space is implemented by a segment called the *virtual space descriptor* which contains a collection of *windows* (Figure 4). Each window identifies a segment, a range of virtual addresses in the space, and locations in the segment that back the designated virtual addresses. The term window emphasizes that ranges of virtual addresses need not map to an entire segment but may map to portions of segments. Windows also describe the protection characteristics of ranges of the virtual space such as *read-only* or *read-write*.

Note that protection is a characteristic of a virtual space and not of its associated backing

4 Related Work

The design of Ra covers two areas of active research, that is, the development of object-based systems and the development of minimal kernel based systems. Some of the notable projects that have implemented object based operating systems are Eden [ABLN85], Cronus [STB86], and Argus [Lis83]. All these systems structure the kernel as a process running on top of Unix. Thus the kernel in these systems are neither native nor minimal.

The Alpha kernel [J+85] used by the ArchOS project and the Clouds kernel [Spa86] are some of the native kernels used in object-based operating systems. Neither are minimal kernels.

Minimal kernels have been used quite extensively to build message-systems. Some of the notable examples are the V-system [CZ83], Ameoba [TM81], Accent [RR81], and Quicksilver [HMSC87].

5 The Ra Primitive Abstractions

The four primitive abstractions recognized by Ra are *segments, virtual spaces, isibas* and *partitions.* The Ra primitive abstractions reflect the mechanisms needed by an object-based system; that is segments reflect logical data units, virtual spaces reflect larger modules of data and code, isibas reflect computation and partitions reflect long-term storage.

In the object model of computation a single thread of activity spans many addressing domains. Ra dissociates the concept of an address space from the concept of a computation by the distinction between the virtual space and the isiba. Similarly there are many forms of data that are used in the domain of one virtual space. Examples are executable code, stacks and heaps, logical clusters of data (or lockable units), parameters. This leads to structuring of the virtual space as a set of segments. Closely associated with the assembling of segments into virtual spaces is the concept of windows. A window allows us to map a range in a segment to a range in the virtual space, giving a completely general composition mechanism. The non-volatile nature of most segments needs a repository for them. This is provided by the partition abstraction.

5.1 Segment

Segments are the basic building blocks provided by Ra. A segment is an uninterpreted, ordered, set of bytes. Bytes in a segment are addressed by their displacement from the beginning of the segment, and the segment is identified by a system-wide unique *sys-id.*

3.4 Ra Hardware Architecture Assumptions

Ra is targeted to run on a variety of commonly available hardware, especially workstations. For efficient implementations, we assume that the hardware should have virtual memory and networking support. The characteristics of Ra's minimal hardware support are:

- *A large, linear, paged or segmented virtual memory.* In object-based systems computation proceeds through object invocations. An implementation typically requires the unmapping of an object space and the mapping of another for each object call and return. Thus, some form of paged or segmented virtual memory scheme is vital for the efficient implementation of Ra.

- *Three distinct machine-level virtual spaces which may be efficiently manipulated separately.*

  The hardware supported virtual address space of the CPU should consist of (at least) three distinct spaces, that we call the K , O and P spaces, for kernel, object and process, respectively (Figure 2). The kernel and system objects are intended to be mapped into the K space, the current process is intended to be mapped into the P space, while the current object is intended to be mapped into the O space. With the exception of the K space, different virtual spaces will be constantly mapped and unmapped into the P and O spaces (Figure 3). In general, the hardware virtual space can conceptually be divided into three segments by appropriate use of the the high-order bits in a virtual address. However this can lead to very large page table. Thus the hardware needs to handle segmented virtual spaces.

- *User/system privileges and page-level read and write protection.*

  Most architectures provide at least user and supervisor execution modes and the ability to protect memory, based on the current mode and the operation attempted. The ability to apply a protection specification to a range of memory will also be useful. Page level read write protection, though not mandatory, will allow us to automate the locking support needed by action managers. The detection of a fresh access to a page of data by a thread can be detected and analyzed (read access or write access) and thus set read or write locks as appropriate. If this facility is missing, language processors have to generate code for locking data pages.

code in any space to which the isiba is assigned. The assignment of isibas to virtual spaces form the basis of object invocation mechanisms.

The attributes and support mechanisms for segments, windows, virtual spaces and isibas are discussed in Sections 5 and 6. Section 8 shows how these abstractions are used to form the final support mechanisms for Clouds objects and threads.

3.3 Ra System-Objects

In addition to the basic services defined above, Ra needs to provide interfaces for other operating system services. Ra does not provide most of the services that a general purpose operating system provides, especially when these services require policy decisions. What Ra does not provide include naming support, communication support, partitions, action management, replication support, and accounting support.

Ra provides a mechanism, by which operating systems services can be implemented via *system-objects*. A system-object is like a Clouds objects in most respects except that it lives in a protected address-space, often is resident (as opposed to paged) and has access to some of the data structures defined by Ra. Unlike the other Clouds objects, the state of a system-object need not survive system crashes, and are re-initialized at reboot. The system-services can be implemented as system-objects.

Like Clouds objects, system-objects can be invoked by user processes, as well as by Ra itself. Thus Ra can use hooks provided by system-objects to get access to a particular service if that is available. For example, management of segments may differ depending on whether the segment is a "recoverable" segment or not. For recoverable segments, the action manager provides hooks that Ra uses to page the segment in and out.

Some system-objects are mandatory, since Clouds cannot function without these. Examples include the partition handlers, the communication drivers and the tty drivers. Most system-objects are optional, such as naming services, action management, accounting, date and time handlers.

Ra is designed to allow plug-in system-objects. That is, system-objects can be loaded or removed at runtime. The system-objects are not linked to the kernel, the binding is dynamic. More details on system-objects are in Section 6.

implemented system service routines.

Given the following set of facilities required of Ra, we can derive the primitive abstractions that Ra should support.

- Ra should support the invocation mechanism since the basic activity in Clouds is object invocation.
- Ra should provide methods for supporting objects as objects are the most important entity in Clouds.
- Ra should provide support for threads, and support for scheduling since activity in Clouds is handled by threads.
- Ra should provide simple yet powerful virtual memory support mechanisms, since objects storage rely heavily on virtual memory.

The need for computation leads to the need for processes and process scheduling. The object invocation support needs the ability to map memory spaces into process spaces. The object support needs the ability to map a collection of memory spaces into a single address space.

In the design of Ra primitives, the basic unit of memory is a *segment*. A segment is used to hold data (or code) as an uninterpreted sequence of bytes. Some segments are permanent (data and code) and some are not (stacks, parameters). The permanent segments reside in *partitions*. Partitions reside on data-servers and the segments are provided by the partitions on demand. Thus, although Ra knows about the existence of partitions, partitions themselves are not part of Ra.

Segments can be coalesced together (using a mechanism called *windows*) into an address space, called a *virtual space*. Each object is supported by one virtual space. The virtual space can live in any one of at least 3 hardware defined segments (or *hardware-segments*, see section 3.4). We call these hardware-segments the *K-space* (for resident objects) the *P-space* (for process specific data) and the *O-space* (for object virtual spaces).

The basic services provided by Ra include the mapping mechanisms needed to provide support for segments, windows, and virtual spaces; and the mechanisms needed to provide support for computations to access the required segments; and the demand paging of segments from partitions.

In addition to the segment and space services, Ra provides support for lightweight processes called *isisbas*. An isiba is scheduled by the low-level scheduler (using time-slices) and can execute

In addition to the design goals, Ra has to meet the following longer term goals to satisfy current plans for Clouds as well as future research interest. Note that the following functions are not supported in Ra, but the design of Ra should allow the following to be supported.

- Ra should provide mechanisms needed to support distributed shared memory.

  Ra is targeted to work in a compute-server – data-server configuration. To handle object invocation in such a network, we have designed a technique called *distributed shared memory*[RAK87]. Distributed shared memory provides a means of viewing objects as existing in a single global distributed shared memory, where although the objects reside on a set of data-servers, they can be used anywhere on demand.

- Ra should provide mechanisms needed to support inheritance.

  Inheritance has emerged as a useful means of controlling complexity in object-oriented programming languages. We have identified a means of efficiently implementing multiple inheritance by taking advantage of "multiply mapped" virtual spaces.

- Ra should provide mechanisms needed to support resource location.

  Resource location has been identified as a critical issue to the performance of a distributed system such as Clouds [Ber87]. Ra must support a clean separation of mechanism and policy to allow implementation of a variety of resource location policies.

- Ra should provide mechanisms needed to support action management.

  Controlling consistency and concurrency of access to shared data is needed in the Clouds system. Action management needs support from the kernel, and the design of Ra should provide these functions. These include cooperation with object invocation, detection of read and write accesses to virtual memory, synchronization, and a means for constructing recoverable storage.

- Ra should provide mechanisms needed to support replication.

  While considering the incorporation of replication under Clouds, we realized that efficient replication schemes required modifications to the basic object invocation mechanism. Similarly, implementing quorum-consensus schemes [Gif79] was best done by overriding Clouds' location transparency.

3.2 Ra Primitives

For an effective minimal kernel kernel design, the set of kernel functions should not need constant revision, but should be able to support all the services required by the user level through properly

A programmer's view of the computing environment created by Clouds is the following. It is a simple world of named address spaces (or objects). These objects live in an integrated environment, which may have several machines. Activity is provided by threads moving around amongst the population of objects through invocation; and data flow is implemented by parameter passing. The system thus looks like a set of permanent address spaces which support control flow through them, constituting what we term *object memory*.

To support the Clouds operating system, each compute engine in the network has to run a specialized kernel that supports mechanisms needed to manage objects and threads. The Ra kernel is a minimal kernel designed for Clouds.

3 The Ra Kernel

As mentioned earlier, the design of Ra was motivated by the need for a modular, extensible, maintainable and reliable kernel. To this end we have identified a set of design goals for the Ra kernel.

3.1 Ra Design Goals

The following are basic criteria that have been used in the design of Ra:

- Ra must be a minimal kernel.

  This decision was based on experience with testing and maintaining the first Clouds kernel.

- Ra should provide mechanisms to support the Clouds model of computation.

  Needless to say, that is the goal for Ra. Ra itself does not necessarily support objects and threads, but provides the mechanisms needed to do so.

- Ra should be extensible.

  Ra is intended to be used in a academic/research environment, which implies that it will go through many upgrades, additions of functionality, changes of techniques and so on. The design of Ra should incorporate ease of extension and ease of modification. To this end, our design uses kernel classes, plug-in system objects, and well defined interfaces inside the kernel, that enhance extensibility.

- Ra should be implementable on a variety of computers.

set of compute-servers (or workstations) and a set of data-servers (or file-servers). Clouds integrates all the sites on the network to form a single computing resource.

- Clouds uses the *object-thread model.*

  The unit of storage and addressing in the Clouds model is an *object.* The unit of activity in the Clouds model is a thread. The Clouds system provides a storage system that is a true *single-level store* [Mye82]. This paradigm is detailed in section 2.1 and in [DLA88].

- Clouds supports consistency of data.

  Since all the data is contained in permanent, shared address spaces, failures can destroy the consistency of the data. The integrity of these address spaces are ensured by mechanisms (such as atomic actions) that guarantee consistency of the data [CD87].

2.1 The Object-Thread Model

All data, programs, devices and resources on Clouds are encapsulated in entities called objects. An object consists of a named (virtual) address space, and its contents. A Clouds object exists forever (like a file) unless explicitly deleted.

Threads are the active entities in the system, and are used to execute the code in an object. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. Threads can span objects, as well as machine boundaries. A thread executes in an object by entering it through one of several entry points, and after the execution is complete the thread leaves the object. An object invocations may pass parameters to the callee, and return parameters to the caller. Several threads can simultaneously enter an object and execute concurrently.

The structure created by a system composed of objects and threads has several interesting properties. First, all inter-object 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. Second, the storage mechanism used in object-based systems is quite different from that used in conventional operating systems. All data and programs reside in permanent virtual memory, creating a single-level store.

The single-level store dispenses with the need for files. Files are special cases of objects, and can be implemented (by the user) if necessary. Similarly the shared nature of the objects provides system-wide shared memory, dispensing with the need for messages. Messages can be implemented by the user if desired.

- The same operating system can support a variety of services or policies (or even several instances of the same service) which is not generally possible in monolithic kernel situations.

A problem with minimal kernels in message-based operating systems has been performance. If a single server process provides a particular service, there is a lack of concurrency and hence the service waiting times become large. So multi-threaded servers are often used, at the expense of more complex server code. This problem does not exist in monolithic kernels (such as Unix), since the kernel services are protected procedure calls, and can be executed concurrently by multiple processes.

In an object system like Ra/Clouds, the services above the kernel reside in objects and are called upon by user processes like the protected procedure calls to systems services (not via interprocess message exchange). Hence services can be accessed concurrently by multiple clients. However object-invocation overhead is higher than the system-call overhead, therefore there is some degradation in performance compared to a monolithic kernel, but this is less than the cost incurred in a message kernel, as well the benefit of simpler code in the service routines. Also, the design of Ra maps the system object (system services) address space into the kernel address space to substantially reduce the service object invocation cost. Thus Ra achieves most of the benefits of a minimal kernel and does not suffer from some of the performance (or complexity of server) penalties.

The rest of the paper is organized as follows. First we present a brief overview of Clouds. Then in Section 3 we present the design goals and objectives of Ra, the rationale for the minimal set of primitives supported by Ra and the hardware needs for implementing Ra.

Section 5 provides details about the primitives supported by Ra, and section 6 shows the mechanisms used for providing extensibility in the Ra design. Finally section 8 shows how the Ra primitive abstractions will be used to build the Clouds operating system.

2 The Clouds Operating System

The design of Clouds has been influenced by the following concepts, that are basic to it.

- Clouds is a distributed operating system.

  The hardware environment for Clouds consists of a set of one or more computers connected by a network. The set of computers can be a collection of mainframes, or more desirably, a

1 Introduction

This paper is a description of the architecture of *Ra*. The motivation for this work comes from the need for a robust kernel that can support the development of a distributed object-based operating system called *Clouds*. Clouds is a research project at the Georgia Institute of Technology. The goal of the Clouds project includes the development of a a uniform, integrated computing environment (OS and applications) based upon the object paradigm (see below). Clouds is the operating system for this environment.

The first kernel for Clouds was a monolithic kernel. This prototype kernel executed on a set of minicomputers (Vax-11/750) and has been operational since 1987. As we enter a new phase of the project we have felt the need for a more robust, simple, portable and extensible kernel. Ra is the result of a fresh design attempt, using the minimal kernel approach. The design of Ra reflects the experience gained and lessons learned during the design, implementation and use of the original Clouds kernel [Spa86, Pit86].

1.1 The Minimal Kernel Approach

A *minimal* kernel converts the underlying hardware into a small set of functions (or abstractions) that can be effectively used to build operating systems. Any service that can be provided outside the kernel (without adversely effecting the performance), is not included in the kernel. Minimal kernels normally provide memory management, low-level scheduling and mechanisms for inter-entity interactions such as message-passing or object-invocations. The minimal kernel idea has been effectively used to build message-based operating systems such as the V-system [CZ83], Accent [RR81], Ameoba [TM81].

The minimal kernel method of building operating systems is akin to the RISC approach used by computer acrchitects. Some of the advantages of minimal kernels are:

- The kernel is small, hence easy to build, debug and maintain.
- The separation of mechanisms from policy. The kernel provides the mechanisms, and the services above the kernel implement policy [WCC+74].
- The functions provided by the kernel are expected to be efficient (fast).
- Most of the services of the operating system are implemented *above* the kernel, hence they are simpler to test, debug and maintain. The services can also be added, removed or placed without the need for recompiling the kernel, or in most cases without the need for rebooting the system.

The Architecture of *Ra*: A Kernel for *Clouds**

José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi,
Mustaque Ahamad, William F. Appelbe, Partha Dasgupta,
Richard J. LeBlanc and Umakishore Ramachandran

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

Technical Report GIT-ICS-88/25

Abstract

Ra is a native, minimal kernel for the *Clouds* distributed operating system. *Clouds* supports persistent virtual spaces, called *objects* and *threads* that direct computations through these spaces. *Clouds* is targeted to run in a compute-server — data-server environment, in which processing is done by threads executing on compute-servers (or workstations) and the objects will permanently residing on data-servers (also called file-servers). Service between data servers and workstations is handled by a *distributed shared memory* paradigm.

The major function of *Ra* is to provide segment-based virtual memory management, and short term scheduling. *Ra* provides support for objects through a primitive abstraction called *virtual spaces*. Virtual spaces are composed of *windows* on *segments*. Ra provides support for threads throgh *isiba*s. An isiba is the unit of activity supported by *Ra*.

The *Clouds* system will be supported on top of *Ra*, by adding *system-objects* to *Ra*. System-objects provide services, such as communication, thread management, atomicity support, object naming and location, and device handling. *Ra* provides mechanisms that allow system-object to be plugged in and out of *Ra* dynamically. *Ra* also provides a set of kernel classes that allow machine dependent variables and interrupt services to be accessed from system objects in a uniform machine independent fashion. Thus *Ra* is a highly modular, expandable and flexible kernel.

Ra is currently being implemented on a collection of Sun-3 workstations and will eventually run on a heterogeneous network of computers of varying sizes including multiprocessors.

This paper describes the architecture and functionality of *Ra*, preliminary details on its implementation, and the envisioned strategy for building a distributed operating system (*Clouds*) on top of *Ra* (by adding appropriate system-objects).

*This work has been funded by NSF grant CCR-8619886.