Abstract

The Clouds kernel is a native layer distributed kernel supporting the Clouds operating system. Clouds is a distributed object-based system, designed to support fault tolerance, location independence, and an action/object programming environment. Some of the key issues in supporting Clouds are the availability of Object Memory, Object Location and Object Recovery. Object Memory provides a set of global, persistent, named address spaces for storing objects. The address spaces resemble conventional segmentation schemes, but are persistent and thus replace both the computational and storage systems used in conventional schemes by a more powerful paradigm. The Object Location system provides transparent object operation invocation mechanisms throughout the distributed environment. The Object Recovery system supports recoverable objects through shadowing and two-phase commit techniques to allow atomicity of actions. This paper describes, in brief, the key issues in the design and implementation of the Object Memory and Storage Management system. The implementation is operational and in use by the Clouds Project at Georgia Tech.

1. Introduction

The Clouds kernel is a distributed kernel intended to support the construction of a fault-tolerant distributed system using the object/action paradigm. The kernel provides three primitives for use by higher level software (operating system and application level software): passive objects, atomic actions, and processes. Objects provide storage (actually persistent virtual memory) for data and code; processes provide the thread of control that executes within objects; and actions provide atomicity necessary for failure or error containment.

The philosophy and implementation of these primitives and the kernel are described in detail elsewhere[1,2,3,4, and are mentioned only in passing in this paper. The paper focuses on the use of the virtual memory systems of general purpose computers to support the object/action paradigm.

1.1 Clouds and Object Support

This section provides a brief overview of the Clouds system and of the support provided for objects.

1.1.1 Objects in Clouds The central theme in the structure of Clouds is the concept of an object. Objects encapsulate code and data into a single unit. A set of defined entry points into the object's code (a set of operations) provides the only means of modifying and accessing the object's data. An object in Clouds has the following characteristics:

Segmented Address Space

Each object resides in its own segment and has a private virtual address space. Irrespective of whether the object is stored on secondary storage or in physical memory, the object always appears to be in this virtual address space.

Passive Objects are passive; i.e., they do not have any active entities (processes) permanently residing in them. As described, an object is simply a named, virtual address space.

Persistent Objects are persistent. An object can be created by users (or by the system) and it exists until explicitly deleted. All modifications to the data in an object during its lifetime are retained. Normal (non-recoverable) objects are as persistent as conventional files, and are subject to similar problems of consistency with respect to failure.

Object Invocation by processes A process invokes an object by entering the address space of an object at an entry point, and executing inside the object. Any number of processes can invoke an object concurrently; they all execute inside the virtual space of the object. Object operation invocations can be nested.

Recoverability (if necessary) The data in the object is recoverable if the object is defined to be recoverable. Any action touching a recoverable object can be rolled back if, for any reason, the action fails.

1.1.2 The Clouds Paradigm

Figure 1 illustrates a typical Clouds object. In the Clouds system, every entity is either an object, a process, or an action. All long-lived components are objects. Since each object lives in a persistent virtual address space, the object can serve as a vehicle for not only computation, but also for storage. Any information placed in an object remains there. Thus, given an object...
system, there is no need for a file system, no need for program and data files, and no need for conversion schemes to convert memory structures to file structures. In fact, there is no need for I/O. Even terminal I/O can be viewed as a side effect of invocations on a terminal driver.

The key mechanism that supports this view of a computational environment is the object memory manager. The object memory manager creates the view of the Clouds world as populated by a set of passive, persistent objects, which reside in virtual memory. The object memory manager is supplemented by the object location system, which integrates the objects into a single, location independent domain, and by the recovery manager, which provides support for keeping object data consistent. The focus of this paper is to provide details about these subsystems, which are the building blocks for supporting the Clouds paradigm.

There are several other projects and systems which have goals similar to those of the Clouds operating system: providing objects and actions on a distributed system. Some representative ones are Argus,[3,4] ISSIS,[7,4,9] Eden,[10,11,12] TABS,[13,14] and COSMOS.[15,16]

All of these systems are built on top of conventionally structured operating systems such as UNIX™ (TABS is built on Accent.) Thus these operating systems use the memory management of the host system and do not have custom management for object support. Having custom management has several payoffs, the most notable one being the conceptual simplicity of the underlying support, leading to better integration of the implementation. This has several pre-mediated side effects, such as efficient support for objects, true concurrency in objects, true segmentation allowing processes to traverse object spaces, and support for atomic actions at the native layer.

1.2 Object Memory and Storage Management

This section provides an overview of object memory and storage management. We divide the task of managing storage and memory for objects into three functions.

Object Location:
Since the storage system is a repository for objects, it must provide the services for locating named objects.

Object Memory Management:
Though objects are really stored on secondary storage, the user is made to believe the objects are always resident in physical memory as a set of virtual address spaces. The object memory manager creates this view through demand paging and handling of object invocation.

Recovery Management:
As processes execute in an object, the data of the object is updated. If the process is executing on behalf of an atomic action, these updates should not become permanent until the action commits. The recovery manager provides the commit support and is able to roll back updates made on behalf of failed actions.

The Object Location system uses sysnames to name objects. Sysnames are guaranteed to be unique throughout the distributed Clouds system. Sysnames are combined with access rights to form capabilities. The Clouds object invocation uses the information to validate the invocation and then calls the storage manager to locate the object.

The primary goal of the virtual memory system is to support the abstraction of object memory. Object memory is the persistent, constantly available state of an object.

Each virtual address space representing an object is kept on secondary storage as a core image which is paged into physical memory on demand. Each process on the system has a stack space, which is also a virtual address space. When a process invokes an object, the object virtual address space is mapped onto the process virtual space, thus merging the object and the process stack into the same address space. (Almost all paging systems allow processes to have two segments, namely the code and the data segments. We use the data segment to store the process stack and map the object into the code segment.)

In conventional operating systems, the code segment of a process is static and contains the code the process executes. In Clouds the code segment of the process contains the object (code and data). As the process traverses from one object space to another, this segment is switched allowing the process to enter the target object. Figure 2 illustrates the use of object memory within this computational model.

The calling process is not required to locate or otherwise ready the object for processing. The object manager maps the object into the process's space on demand. Inside the object, the programmer manipulates object data directly; there is no explicit input/output to perform, because object data is always present in object memory. The kernel finds and prepares objects transparently, using a local invocation as described above, or issuing a remote procedure call (RPC) if the object is not available locally.[4]

Remote invocations are handled at the remote site, where a slave process is used to invoke the object. Thus, RPCs are handled like local invocations at the remote site.

The recovery management system supports the consistency of object data. Operations performed in object memory must be made recoverable if performed by an action. Objects are considered by the kernel to be either non-recoverable, meaning that the consistency of persistent object data is not guaranteed, or recoverable, meaning that part or all of object memory is subject to recovery processing and system crashes will leave this portion of object memory in a consistent state. The programmer of an object does not specify how data is made recoverable, although he may specify what data is recoverable, using the features of the Aeolus programming...
language. The virtual memory system provides much of the information necessary to ensure that object data is recoverable. This information is used by a set of recovery protocols provided by the storage manager, which are described in section 4.

In the next few sections we discuss the object location mechanisms (briefly), the object memory management design (in detail) and the recovery scheme.

2. Object Location

Every object in the Clouds system has a unique capability, composed of a unique sysname and some access information. The capability contains no information as to the location of the object; it simply identifies one object in the entire distributed system. Processes access objects (invoke one of the object’s operations) by using the capability. When a process requests access to an object having a capability, the kernel uses the local storage management system to search for the object. If the object is found locally, the request proceeds. If the object is not found, then a global (broadcast) message is sent, and all of the participating sites try to find the object. The site which finds the object uses a slave process to access the object on behalf of the calling process. The kernel search mechanism is a very simple one, and was chosen primarily for the simplicity of its implementation in light of the fact that our nodes are connected via an Ethernet. We do not expect that this will be the ultimate search strategy and, in fact, we do not believe it will scale to larger configurations than our current one. A good search strategy is the topic of ongoing research.

Storage management is ultimately responsible for finding an object at any site. The kernel storage manager attempts to reduce the search time at each node, particularly in those instances where the object is not present on the node. This does not remove the need for a good object search strategy, but is certainly a good optimization for any search strategy, provided the overhead is not too high. To determine whether an object is present locally, the segment system examines in sequence three structures: the Active Object Table (AOT), the maybe table and finally the partition directories. If an object is currently in use, it is found in the AOT, and the search succeeds quickly. Otherwise, the maybe table is consulted. Given a capability, the maybe table indicates either that the object is not present locally or that it might be. Thus, the maybe table is an approximate membership test. The maybe table trades accuracy of its membership test for speed of the test. The purpose of the maybe table is to provide a very efficient means of eliminating sites from a search for an object. Hence, the maybe table is particularly useful in remote operation invocations. The queries to the AOT and the maybe table are quick enough that they may be done before a slave process is started for an incoming search request. If the search is short-circuited at this site (if the maybe table returns a negative response), then no process cleanup is needed. Short-circuiting of local site searches can affect the performance of remote invocations dramatically as otherwise a search of the local storage may be necessary.

If it is determined through the maybe table that the object may be local, the search continues. The disk partition directory provides the final say as to the existence of the object. If the search is on behalf of a remote request, a slave process is started by the process manager for searching the partitions and invoking the object, if it is found. Figure 3 illustrates the use of the maybe table during an RPC.

3. Object Memory Support

Object memory relies primarily on the storage management and the object management subcomponents of the Clouds kernel. Object management is responsible for handling object operation invocations. The operation invocation kernel has three functions: validating the capabilities, argument parameters, and operation code used in the invocation; initiating the invocation and handling the return from an operation; and management of an object’s memory. The storage manager is responsible for the maintenance of structures representing object data, recovery of object data (as directed by action management), and performing the actual mappings necessary for object memory. An important part of the storage manager is the segment system.

3.1 Segment System

The segment system forms the interface of the kernel storage manager and is the mechanism underlying the abstraction of object memory. The segment system enables the kernel to map a collection of segments into object memory, forming the object memory space for a Clouds object.

3.1.1 Clouds Segments. Segments represent an alternative view of object data and code. This view is reserved for the kernel and facilitates the kernel’s treatment of objects. Whereas regular users of an object can manipulate data only through operations defined for the object (via a system call to operation invocation), the segment system allows the kernel to operate directly on object data and code through a set of primitive operations. The segment system can be viewed as a collection of objects of a single, primitive type. Segment data is an uninterpreted sequence of bytes which can be manipulated by segment operations such as: mapping or unmapping a segment into virtual memory; reading or writing a page of a segment; and performing recovery procedures on segment data.

Clouds segments serve a similar purpose to the segments found in Multics. Many of the internal structures used to implement segments are similar. However, our view of segments is at a lower level than that presented by Multics and segments find use both in the support of object mapping and in object recovery.

3.1.2 Segment Representations. The abstract abstraction of a segment is supported by two distinct representations of segment data. One is the segment as it is mapped into virtual memory. The other is the representation residing on secondary storage (referred to simply as storage henceforth). This representation is a tree of storage pages. The root is a segment header, which contains the identity of the segment and other descriptive information. The tree leaves are
The segment data. Internal nodes of the tree are mapping blocks, providing paths to data blocks.

The segment header must reside entirely within a storage page, in order to satisfy the correctness assumptions made for recovery management. Currently, the design provides for page-sized segment headers, simplifying the management of the headers. The information contained in the segment header, with a few exceptions, is not dependent on the object system residing on top of the storage manager. The segment header has two components: the segment descriptor and the segment map. The segment map contains pointers to the segment data, either through mapping blocks or directly to data blocks. The segment descriptor contains information describing the segment, such as the segment’s sysname, its size, whether it represents data from a recoverable object, and which portions of the segment data are recoverable. It also contains information describing the segment’s recovery state. Important information includes a pointer to a shadow copy of the segment and a field holding a sysname for the action that is committing in the segment. These fields are only valid when the segment’s state is “precommitted”.

Storage management preserves the consistency of an object’s persistent state through the use of the recovery protocols described in a later section. Thus, the storage image for a segment always reflects the latest committed image of the object’s memory space.

3.2 Object Operation Invocation

Clouds objects are typed. The type of an object is described by its object template, which is used to create instances of the object. Information found in the template includes the code, the data format, and information describing how object memory is to be constructed. Objects are actually a collection of segments; typically, an object will have a code segment and at least one data segment. Segments may have different attributes depending upon their use: volatile (non-persistent) heap segments; read-only code segments; and persistent object data segments. Segments, particularly code segments, may be shared among several objects. By the use of the mapping mechanism on segments, the abstraction of a constantly available object memory space is presented. The object template contains the description of how the segments must be mapped by the object manager to build object memory. The structures used are called windows. Each window specifies where in object memory the segment is to be mapped and the attributes of that segment. In addition, windows allow the mapping of only portions of a segment. A segment may appear as a window in any number of object memories, facilitating sharing.

3.2.1 Operation Invocation. Underneath the abstraction of object memory is the reality that an object’s state may reside on storage and is not immediately accessible. Objects are activated on demand by an invocation request. A description of the general framework of operation invocation is presented, discussing how the mapping of object memory is initiated. For further details on the invocation mechanism itself, see [4]. The active object table (AOT) and its associated active object descriptors (AODs) represent objects whose operations have been recently invoked. Activated objects are those objects which have descriptors in the AOT. AODs are created (and the object activated) when an operation invocation is first made on the object and remains until aged from the table or replaced by other AODs that are used frequently. In addition to information identifying the object and important to the invocation mechanism, the AODs also contain descriptors reflecting the mapping of segments into object memory, which are obtained from the object template.

The invoker of an object operation makes a call on the kernel operation invocation mechanism in the object manager. After formatting the parameters to the invocation and validating the object capability and operation code, the object manager determines whether the object resides on the local site or whether an RPC is necessary. If the object is not activated already, object management makes a call on the storage manager to perform a search for the object as described earlier. The result of this call indicates that the object is either local and invocation can proceed, or that the object is on a remote site, and an RPC is necessary. In either case the object is activated and an object descriptor for the object is added to the AOT. For a local object, the information describing the segments necessary for object memory mapping is present; in the case of a remote object, the object descriptor indicates simply that the object is not local and no mapping information is available.

When a site receives an RPC, storage management performs a local search for the object’s data segment. If the search is successful, then the object manager at this site is called to invoke the appropriate operation. An object descriptor representing the object memory is added to the AOT and the invocation is performed.

3.2.2 Segment Mapping. This section discusses the support that the segment system provides for the operation invocation mechanism and the abstraction of object memory. Similar to the AOT, the active segment table (AST) and the associated active segment descriptors (ASDs) are used to manage activated segments. Each ASD contains information used to map the segment in virtual memory. Important to virtual memory management are: the storage segment header, information describing the windows into which the segment is mapped; the virtual page table; the storage page table; and caches of storage data for efficiency. The storage segment header provides information as to where the segment resides. The virtual and storage page tables describe the mapping of the virtual memory image of the segment to the physical memory image and to the storage image, respectively.

The AST and ASDs serve two purposes. First, they indicate segments which are currently being accessed by a process or action or which were recently accessed. This makes the location of active segments faster, since searches through partition directories are not necessary. Second, the descriptors act as caches for the management of the segment. This makes services such as page-fault handling more efficient.

The page tables provide the actual mapping for the segment. The implementation of the tables depends heavily on the hardware and is not discussed in this paper. The interested reader is referred to [21], [4], and [23]. An important use of the mapping information is in the support of action processing. Special mechanisms for providing action versions of object data using the page tables are described in [23] and [21]. These mechanisms ensure that each action perceives its own view of object memory. The recovery protocols used in the Clouds systems rely in part on the information held in the page table, particularly the information as to what pages have been modified.

A segment is activated if it has a descriptor in the AST, and deactivated otherwise. The activation of a segment is caused by the operation invocation’s attempt to satisfy an invocation request. At the end of a successful search, a segment is activated, but might not be mapped into virtual memory. Before the invocation can continue, portions of the segments must be mapped into object memory windows and the virtual and storage page tables must be created and initialized to reflect the mapping. The page tables are created with only information describing the attributes of the
segment pages; the mapping of segment page to physical memory or storage is done via demand paging. In fact, all activation—object, segment, and page—"activation" (mapping a faulting segment page into virtual memory)—is by demand. Volatile segments for objects, which have no persistent image, may be created at this time.

Segments may be deactivated when no longer needed by any object; i.e., when the segment is not mapped into any object memory window. In the case of volatile segments, there is no persistent state associated with the segment and deactivation is really the destruction of the segment. The deactivated segment's descriptor is removed from the AST, and the pages used in its mapping are freed. For segments which represent persistent portions of objects, the segment might be left active for a time, depending on the fraction of AST entries that are full. When it is actually decided to deactivate the segment, due to lack of references or being bumped by another segment, its persistent state must be flushed to storage.

The segment system also provides maintenance of segment mappings. There are two parts to this mapping: the mapping between object memory and storage; and the mapping between object memory and physical memory.

The segment system is responsible for locating and transferring segment data into object memory. Segment pages will be mapped into object memory (due to page faults) and out of object memory (due to page replacement) as in conventional systems. However, there are some differences, since this maintenance by the segment system is especially critical to recovery management under Clouds.

The recovery algorithms rely on the mapping information to determine what data has been modified. Also incorporated into object memory management is the management of action versions. Since an arbitrary number of actions may be executing inside a given object, an important task is finding the version of the data that is visible to a given action. Thus, mapping pages into a segment and unmapping pages from a segment (part of page replacement) must be handled differently in Clouds than in a conventional system.

To understand how the segment system selects local storage to map to an object memory page, it is necessary to understand the underlying semantics of different classes of segments. There are three important classes of segment to consider: volatile segments; non-recoverable segments; and recoverable segments. The basic use of volatile segments is for the data (stack and heap) portion of processes. These segments have no persistent image; when a local site crashes, the processes and their segments are gone. Non-recoverable segments support non-recoverable objects. The information contained in these segments is persistent, but consistency is not maintained. Lastly, recoverable segments support recoverable objects, and therefore the data in these segments must be both persistent and must be maintained in a consistent state. The differing semantics of these three segment classes requires differing treatments of requests to map a page into the segment or remove a page from the segment. The state of the particular page within a segment also affects the treatment of the page.

The page states reflect the status of the storage page table entry for the page. The state can be determined by examining the storage page table entry. The states are formed from a cross product of vectors (shown in Table 1), one of which reflects the mapping between segment page and storage image and the other of which reflects the existence of a storage image for the page and whether there is an action version of the page.

A page is unmapped when it has not been referenced before the time of the page-fault and has not been mapped into object memory; i.e., it does not appear in the underlying virtual memory mapping tables. Otherwise, the page is mapped. The descriptions allocated and unallocated refer to whether a storage image exists for the page. A volatile page initially has no storage image. In the case of non-recoverable and recoverable segments, not all of the pages may be initially allocated on storage. This speeds object creation, but requires that storage be allocated when the pages are referenced subsequently.

The altered description indicates that a page has more than one storage image associated with it, resulting from a modification by an action being written to local storage. Note that this description is appropriate for recoverable segment pages only. In this case, the action versions of these pages are not part of the persistent segment image. Unaltered pages have only one storage image contained in the persistent state of the segment.

To describe the actions taken in the cases represented by these page states, the following descriptions of local storage are used:

- **Paging storage:**
  - This is local storage used as a temporary image for data mapped into object memory. This type of storage is most commonly used for volatile pages. Paging storage does not survive between activations of a segment and is not persistent.

- **Segment storage:**
  - This is storage that is part of a non-volatile segment. As such, this storage survives segment activations, and represents part of the persistent state of a segment. Such storage is used for both non-recoverable and recoverable pages.

- **Shadow storage:**
  - Shadow storage is used to represent the uncommitted modifications of an action which were written to local storage (as may happen during virtual memory page reclamation). This storage may become part of a segment after a successful commit and is only used for recoverable pages. Hence, on action commit it becomes segment storage. Shadow storage is not persistent.

Table 1 illustrates the use of each storage class as a function of page state and segment class. In the case of unmapped pages, storage is being allocated. In the case of mapped pages, storage already exists. In the unmapped/unallocated state, no page has a storage image and a reference to such a page results in the mapping of a zero-filled page into the segment. For non-recoverable segments, pages are mapped from segment storage. Either the segment storage is cached in memory or the segment system must query on-storage tables. For an unallocated page, segment storage must be allocated as part of the mapping. Recoverable segment pages initially are mapped from segment storage, but after they are altered the mapping is from shadow storage. The exception to this is when the recoverable page is initially unallocated. In this case, storage is obtained from shadow storage, so that an abort or commit may be done properly later.

| Table 1. Non-modifying references to segment pages |
|-------------------|-----------------|-----------------|-----------------|
| Page state        | Segment Class   | Non-recoverable | Recoverable     |
| Unmapped/unallocated | Zerod       | Zerod           | Zerod           |
| Unmapped/allocated    | N/A             | Segment         | Shadow          |
| Mapped/unaltered    | Paging          | Segment         | Segment         |
| Mapped/ altered     | N/A             | N/A             | Shadow          |
The discussion so far has concentrated on the mapping between conventional systems. However, recoverable segments require versions for modified recoverable pages to make up the action's version of the segment. The creation of pages must be flushed to storage. In any state, unmapping a page in segment storage is performed by the segment system. All of the modification history for segment pages is stored in the shadow. Only at the coordinating site is an additional log entry needed.\[25\] The result of precommit is a shadow version of the segment header points to the shadow version.

<table>
<thead>
<tr>
<th>Table 2. Modifying references to segment pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page state</td>
</tr>
<tr>
<td>Segment Class</td>
</tr>
<tr>
<td>Non</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Unmapped/unallocated</td>
</tr>
<tr>
<td>Unmapped/allocated</td>
</tr>
<tr>
<td>Mapped/unaltered</td>
</tr>
<tr>
<td>Mapped/altered</td>
</tr>
</tbody>
</table>

The virtual memory system has a great deal of influence on recovery processing. Clouds recovery processing is entirely separated from the concurrency control done by action management. Clouds recovery is pessimistic; no part of the recoverable state of a segment is modified until the commit is complete. Until that time, all modifications to the segment's recoverable state exist as shadow copies. Shadowing is done at the page level. Because Clouds actions may span several sites, action management uses a two-phase commit protocol to commit an action. The recovery protocols used by the storage manager fit into this model.

During the first phase of commit, the preparation phase, action management calls upon the storage manager to \textit{precommit} action modifications. The storage management protocol for the prepare phase takes the action and object sysnames and builds a shadow version of each object data segment on local storage. The identity of modified segment pages is found in the page table for the committing action. Modified pages must be flushed to local storage as shadow pages. Note that shadow images may have already been allocated for some pages that have been written to storage prior to precommit. Indeed, if there was no modification after this initial write, no write is necessary during precommit for the modified page; it is already shadowed correctly. As noted in \[25\], pre-flushing of segment pages modified by actions as a background activity could produce considerable savings in terms of writes to storage during the preparation phase. Besides modified data pages, mapping pages affected by the action and the segment header must be shadowed as well. In fact, the shadow version of the segment header acts as a sort of commit log for action management, as the action syscall is stored in the shadow. Only at the coordinating site is an additional log entry needed.\[25\] The result of precommit is a shadow version of the segment header points to the shadow version.

During the second phase of the commit, the action manager asks the participating sites if they can commit; that is, if the necessary shadows have been built. If all agree, commit occurs. Otherwise, the action aborts. Storage management has a protocol for each case. Action management invokes each as necessary. If a commit is indicated, the storage manager makes directory adjustments so that the shadow versions of the touched segments become the working versions. Allocations of the shadow pages become permanent and previous page versions are deallocated. If the case of an abort, the current segment header is rewritten so that the shadow version is not referenced and the shadow pages are deallocated.

The recovery protocols assume that single page writes to storage are atomic and that only one action is committing (although more than one action may be active) inside a segment at a given point. Given these assumptions, the recovery protocols leave a segment's recoverable data in a consistent state, even in the event of a site crash. After such a crash, information as to what segments are in a precommitted state is reconstructed by examination of the segments. This entails reconstructing the storage allocation state and checking the structural consistency of all segments as well. Segments in a precommitted state are then subject to either the phase two commit or abort protocol, depending on the final outcome of the action (the action syscall is found in the shadow segment header).

4. Recovery and Segment Mappings

The virtual memory system has a great deal of influence on recovery processing. Clouds recovery processing is entirely separated from the concurrency control done by action management. Clouds recovery is pessimistic; no part of the recoverable state of a segment is modified until the commit is complete. Until that time, all modifications to the segment's recoverable state exist as shadow copies. Shadowing is done at the page level. Because Clouds actions may span several sites, action management uses a two-phase commit protocol to commit an action. The recovery protocols used by the storage manager fit into this model.

### Table 2. Modifying references to segment pages

<table>
<thead>
<tr>
<th>Page state</th>
<th>Segment Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non</td>
<td>Recoverable</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Unmapped/unallocated</td>
<td>Paging</td>
</tr>
<tr>
<td>Unmapped/allocated</td>
<td>N/A</td>
</tr>
<tr>
<td>Mapped/unaltered</td>
<td>Paging</td>
</tr>
<tr>
<td>Mapped/altered</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The discussion so far has concentrated on the mapping between object memory and storage. The segment system is also responsible for the mapping between object memory pages and physical page frames. For the most part this mapping is similar to that found in conventional systems. However, recoverable segments require special treatment. Whenever a recoverable page is modified by an action, conceptually a new version of the page is created for that action. The collection of pages modified by the action collectively make up the action's version of the segment. The creation of versions for modified recoverable pages is handled transparently by the segment system. For more details, see \[21\] and \[23\].

Table 2 presents a summary of the storage classes used when an object memory page must be unmapped. Additionally, modified pages must be flushed to storage. In any state, unmapping a recoverable page requires shadow storage. When the page is unallocated, the segment system must allocate shadow storage for the page. This is so that its contents do not overwrite the current state of the page in segment storage, as the current state may be in use by other actions. Similarly, when the page is unallocated, shadow storage must be allocated before the page is written to storage. When the page is altered, however, there may already be shadow storage reserved for the action's version of this page. If so, it is used; otherwise new shadow storage is allocated.

In the case of volatile and non-recoverable segments, the appropriate storage is allocated when the state of the page is unallocated. Otherwise, storage has already been allocated.

With these mechanisms, object memory is made available to invoking processes and the multiple images associated with the actions concurrently updating the object are maintained correctly. The next section examines how the segment system uses this information in recovery processing for action events.

5. Current Status and Performance

This section briefly describes the current status of the kernel prototype and gives some preliminary performance figures for recovery processing.
5.1 Status

The structures and mechanisms described in this paper have been implemented. Clouds is in operation and uses the object memory and storage management system to support objects. Operation invocations (and the attendant mapping of segments into object memory) is one of the most frequent operations in the Clouds system. The invocations are location independent and remote invocation of object operations is currently supported. Storage management provides the necessary persistent storage for object data.

The implementation environment for the first prototype consists of three VAX 11/750s connected via a 10Mbit/sec Ethernet. Currently, the only secondary storage devices are three ten megabyte removable media RL02s. Software for larger storage devices (RA8ls) is under development, as is a new kernel prototype on a network of Sun 3/60 workstations.

The recovery protocols necessary to support action management have been implemented. They have been tested and work correctly. An action management system that provides user level transaction management uses the commit and recovery mechanisms of the storage manager to support the user level services.

5.2 Performance

This section presents some timing results for two of the recovery protocols provided by the storage manager: the preparation phase protocol; and the final commit protocol for phase two. The recovery protocols are the slowest of the mechanisms discussed in this paper, and thus the following performance numbers represent an upper bound on timing figures for the storage management system.

Also presented are some timing figures on the underlying I/O support required by recovery management. This information, along with an analysis of the number of I/O requests required to perform the tests described, is used to factor out the hardware impact on the performance of recovery management. As mentioned previously, the prototype uses a RL02 removable pack disk drive. This device is both slow and has limited storage capacity (10 megabytes), but has such a simple interface that a device object was easily implemented. This provided the ability to test the storage management and virtual memory management aspects of the prototype. The RL02 has a one cylinder seek time of 15 milliseconds and a data transfer rate of 512 kilobytes per second.\[200\]

In timing tests presented in[21], the average time required for a random read or write over a two megabyte partition was 51 milliseconds. Average seek times recorded for short, five cylinder seeks and long, 512 cylinder (across disk) seeks, were 21 milliseconds and 66 milliseconds, respectively.

Table 3 shows results for the recovery protocols performed on segments of various sizes. All times are in milliseconds. Precommit is the preparation phase protocol that builds the shadow version of the segment. Commit is the second phase protocol used in the case of a successful commit. For each segment, varying numbers of pages were modified and the recovery protocols used on the segment.

<table>
<thead>
<tr>
<th>Test</th>
<th>Time (ms) Precommit</th>
<th>Time (ms) Commit</th>
<th>#Reads</th>
<th>#Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 page</td>
<td>78</td>
<td>113</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>14 pages</td>
<td>403</td>
<td>132</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>40 pages</td>
<td>1003</td>
<td>163</td>
<td>0</td>
<td>42</td>
</tr>
</tbody>
</table>

For each test the average time in milliseconds to perform the protocol is presented along with the number of read and writes necessary to perform the necessary protocol. The reads and writes include those necessary to shadow segment structures such as mapping pages and the segment header.

Note that these results represent only timings for the shadow paging scheme and do not represent the efficiency of the segment input/output used to provide paging of object data. The timings for these operations would on average be closer to those for the disk reads and writes. Also, these results represent a worst case for recovery; none of the modified pages was flushed to disk prior to commit. For larger objects a mechanism to flush modifications into shadow storage prior to precommit should improve commit times greatly.

6. Summary

The Clouds kernel supports the object/action paradigm, providing long-lived objects both for computation and for storage. In addition, provision for the recovery of object data is made through a set of protocols provided by the storage management system. The three major functions which support this paradigm, Object Memory, Object Location, and Object Recovery, are provided transparently by the storage management and object management subcomponents of the kernel. These abstractions and because of the transparent nature in which they are provided, Clouds users and programmers are provided with a unified distributed environment in which to work. There is no need for programmers to be concerned with the underlying mechanisms.

As the kernel is built directly on the bare machine, the design and implementation decisions have been made on the basis of what best supports the object/action paradigm. The result is a set of subcomponents, object management and storage management, customized for the support of objects. These components are well integrated and provide efficient support for the objects.

This is particularly evident in the support provided for object recovery. A large part of the information required to support the recovery protocols is obtained from the support necessary for object memory. Improvements in the performance of recovery processing can be obtained through simple changes to the current mapping mechanism for object memory (namely the pre-flushing of modified recoverable pages).

7. Acknowledgements

The authors would like to recognize the many people who have put a great deal of effort into getting Clouds off the ground. The initial Clouds design was due in part to Martin McKendry and Jim Allchin. Since then Clouds has been the result of a team of dedicated researchers that includes Mustaqa Ahamad, William Appelbe, Glenn Benson, José Bernabéu, Phil Hutto, Greg Kenley, Yousef Khalidi, Richard LeBlanc, Gene Spafford, Henry Strickland, Win Strickland, Scott Vorrheinmann, Peter Wan, and Tom Wilkes.

References


