The iMAX-432 Object Filing System

Fred J. Pollack, Kevin C. Kahn, and Roy M. Wilkinson
Intel Corporation, Aloha, Oregon

ABSTRACT

iMAX is the operating system for Intel's iAPX-432 computer system. The iAPX-432[1] is an object-oriented multiprocessor architecture that supports capability-based addressing. The object filing system is that part of iMAX that implements a permanent reliable object store.

In this paper we describe the key elements of the iMAX object filing system design. We first contrast the concept of an object filing system with that of a conventional file system. We then describe the iMAX design paying particular attention to five problems that other object filing designs have either solved inadequately or failed to address. Finally, we discuss an effect of object filing on the programming semantics of Ada.

1. Introduction

The object filing system of iMAX is responsible for supporting the permanent storage of 432 objects in a manner that is consistent with the access control mechanisms of the hardware. It is a store permitting the storage and retrieval of any 432 object, simple or complex, while safeguarding all essential characteristics of the object including, most significantly, its identity, type, and structure.

The iMAX object filing system differs from conventional file systems. In such systems, the filing store is organized as a tree-structured hierarchy where the interior nodes are directories and the leaf nodes are files. Most files are used to store unstructured data streams. Since programs in such systems map into relatively flat and static addressing structures, they may also be stored in files as an unconnected collection of pages or segments.

In contrast, the iMAX object filing system is responsible for the 432 **object space**, the collection of all objects in the system. Although simple unstructured data is contained in this space, it in general has much more structure. It can be viewed as a network in which a node is an object and a directed arc from node A to node B is an access descriptor contained in object A which references object B. Any object may contain access descriptors for other objects. Such references may be circular and may cross device boundaries. Because the 432 supports a more dynamic and structured addressing environment, a 432 program consists of a network of objects connected by access descriptors. Object filing stores such a network on a permanent memory device (e.g. disk) in a manner consistent with the 432 architecture. In addition to programs, the object filing system can also be used to store structured data. This is accomplished by extending the protection and addressing mechanisms of the architecture from volatile memory to permanent memory. The result is a computer system in which all facets of protection are based on a single mechanism. In contrast, conventional computer systems implement several protection mechanisms, one for memory, a different one for files, a third for directories, a fourth for privileged programs or users, and so forth. Without doubt, the use of multiple mechanisms creates difficulties where the

different mechanisms must interface, e.g. a particular file protection indicates a particular memory protection if the file contains a runnable program. The difficulty in defining the interfaces between protection mechanisms also leads to a lack of security. For example, on most systems it is impossible for a user to run a program and limit that program's access to the user's files.

2. Overview

Object filing systems have been implemented in two commercial systems, IBM System/38[2] and the Plessey/250. [3] There have also been two research implementations, for the Hydra[4,5] and CAP[6,7,8] operating systems. In addition, MIT has done significant research in this area. [9,10,11] In this paper, we give special emphasis to five problems. These are problems that previous designs have either solved inadequately or failed to address. While we will describe these problems in more detail as we present the iMAX design, we begin with a brief description of them.

The first problem is assuring the uniqueness of object names across system boundaries. An access descriptor in the object store contains a name for the object which it references. If uniqueness of object names were not assured, the movement of an object from one system to another (e.g. moving a disk pack containing objects) could result in a loss of protection; i.e., a moved object's access descriptor could reference an object with the same name as the one referenced on the originating system.

The second problem is providing an object-oriented, logical naming mechanism. Unlike conventional systems in which absolute object identification is difficult but indirect naming is easy, capability-based systems have the problem that names may be too tightly bound to their referants. For example, a user program normally should contain a logical name for a utility program in order to get the most recent version, as opposed to the version of the utility that existed when the user program was compiled and linked. If the user program holds an absolute access descriptor to the utility, this will not occur.

The third problem is maintaining the consistency of a collection of objects across updates, i.e. supporting atomic actions. This involves the ability to make a group of changes, called a transaction, to a collection of objects such that if the system crashes, either all the changes or none of the changes that make up the transaction will be reflected in the collection. While this problem is addressed by most data base systems, it is considered by very few operating systems.

The fourth problem is the efficient management of the object space. The major aspect of this problem is the reclamation of space that is no longer needed. Because of the network organization of the object space, it is difficult to know when the space for an object can be reclaimed.

The fifth and final problem to receive special consideration in this paper is the inefficiecy, both in space and time, of dealing with small objects. In Hydra measurements, [12] the average size of an object in the filing store (i.e. the Hydra passive GST) was 222 bytes. In the active system, the average was 326 bytes. In Batson's study of the B5500 MCP, [13] the average size of Overlayable Data Segments was 59 48-bit words. Both studies emphasize a preponderance of small objects. Our initial data indicate that the average size of an object on the 432 will be slightly less than these figures. The space overhead for storing such small objects could be large. Also, if small objects had to be accessed individually, the performance of the system would suffer. For example, since a large 432 program will typically consist of hundreds of objects, the loading of such a program would be grossly inefficient if each object had to be loaded individually. Consequently, grouping objects is essential to providing acceptable performance and space utilization.

In the next section, we discuss the role of type managers in the overall design of iMAX. Following that, we describe the structure of the object space. We then present the object filing extensions to the protection and addressing mechanisms of the 432 architecture. Then we describe an example of object filing on programming semantics. Finally, we review how the iMAX object filing design has addressed the problems we stated above.

3. Types And Their Managers

The major goal of iMAX[14,15] is to provide Intel customers, principally original equipment manufacturers, with an extensible operating system. This is made possible by the 432's uniform approach in the design of its hardware, operating system, and language. The key element in this approach is support for data abstraction. The architecture provides efficient hardware operations that enable a user to define a new object type and a type manager that operates on objects of

that type. The iMAX object filing system provides the software operations that give type managers the ability to store their objects in a permanent file store in a manner consistent with the architecture's protection and representation mechanisms.

Through the operations it provides, a **type manager** is the exclusive agent for changing the state of an object. An object, however, can enter some states that only the operating system (in our case, iMAX), and not the type manager, is aware of. For example, the iMAX garbage collector[15,16] finds objects that are no longer accessible. When such an object is found, iMAX manufactures an access descriptor for it and sends it to its type manager.

A **type definition object** forms the basis of the mechanism that enables iMAX to communicate with a type manager. Every object has an access descriptor for the type definition object that represents its type. A type manager can store access descriptors in its type definition object for ports[15] and subprograms. In the above example, iMAX selects the appropriate port access descriptor from the object's type definition object and sends an access descriptor for the object to this port.

4. Structure of the Object Space

The **object space** (i.e. the collection of all objects in the system) is decomposed into two distinct spaces, **active** and **passive**. The passive space supports the permanent storage of objects. While stored in the passive space, the representation of an object can only be manipulated by invoking the operations of the object filing system. For a programmer to address the object's representation directly (e.g. as the operand of a hardware instruction), the object must first be mapped into the active space. This action will create a 432 object descriptor for the object. Hence, the active space consists of those objects for which hardware-recognized object descriptors exist.

A consequence of the two-space approach is that it is possible for the active version of an object to be different than its passive version. The updating of a passive object, based on its active version, is under the control of the object's type manager. For this reason, the passive space should not be viewed as a virtual store.

It is, in fact, the active space that is implemented in virtual memory. An object in the active space is not necessarily in primary memory; i.e., the object could be swapped. Such an object is still considered to be in the active space since it is still in hardware-recognized form. For example, a swapped object can contain hardware-recognized access descriptors. The object descriptor for the swapped object would indicate that the object was swapped; so that a reference to its representation would result in a swap-in request.

One important property of the two-space approach is that a typical system crash (e.g., a power failure), while destroying the active space, will not destroy the passive space. Such a crash should not have catastrophic effects since users can periodically update their objects in the passive space.

In addition to accessing a passive object by activating it, it is convenient for a type manager to be able to manipulate its objects' passive definitions. Directory objects are a case in point. A typical directory operation is to retrieve an access descriptor from a directory. The activation of the entire directory object would result in needless overhead. All access descriptors in the directory would have to be transformed from passive to active form even if only one was actually being accessed. Subsequent passivation would perform the reverse operation. The directory manager can avoid this overhead by invoking software operations provided by the object filing system. These operations will access the passive definition of the directory without causing an activation.

4.1. The One-Space Alternative

Other object filing systems, e.g., Hydra and System/38, have chosen to implement a one-space model as opposed to our two-space model. We rejected the one-space model because of the difficulty in maintaining object consistency in the presence of system crashes. In the one-space model, the operating system chooses when to update the passive version of an object with its active version. Since an object may go through several intermediate transitions between consistent states, the operating system may choose to update the object in a transitional state. If an active space crash occurs, the passive object would not be in a consistent state. Since it is not possible for the operating system to know when an object or a collection of objects is in a consistent state, there is no convenient solution to this problem for one-space systems.

In our two-space model, a type manager has control of storing its objects in the passive store. This control can be used to guarantee the consistency of a passive object. A request for the passivation of an object can be triggered explicitly by a user with an access descriptor for it or implicitly by the operating system. An explicit request occurs when a user program calls the object filing subprogram, Update, passing an access descriptor for an object. Update will call the type-specific update subprogram for the passed object. This subprogram is found in the type definition object that is referenced by the passed object. The type-specific update program, part of the type manager, can choose to update the object's passive definition, or simply refuse the update request. iMAX will generate an implicit passivation request when an active object, which has a passive version, becomes inaccessible in the active space. An access descriptor for the inaccessible object will be sent to the port object referenced by the object's type definition object. The type management process that is receiving messages from this port has the same options as the update procedure.

If a type manager chooses not to supply a type specific update subprogram or port object, iMAX will supply reasonable defaults. For update, it will change the passive version so that its value is the same as the active version. For an inaccessible object it will do the same, and also delete it from the active space.

Although iMAX implements the two-space model, most users, namely, those who do not write their own type managers, actually see a one-space model. This is best illustrated by example. When a user accesses the representation of an object, the object will be activated. It will remain in the active space as long as active access descriptors for it exist. When the object is no longer accessible via active access descriptors, the object will usually be stored in the passive space. The actual storing is under the control of the object's type manager. Hence, the two-level nature of the store is more visible to type managers than it is to casual users.

4.2. Active and Passive Forms

An access descriptor consists of an object name and rights which determine the operations permitted on the object using the access descriptor. The active and passive forms of an access descriptor differ in the format of the object names. The active form of a name is a 24-bit address, mapped by hardware, for an object descriptor. In the passive form of an access descriptor, the name is a unique identifier, **UID**. The first time an access descriptor for an object is stored in the passive space, the object is assigned a UID.

When an access descriptor is made active, the active space must first be checked to see if the referenced object has already been activated. The **active object directory** (AOD) contains this information. An AOD entry contains a UID object name and an active access descriptor for the object. (Note: This reference is not considered by the garbage collector; if it were, the active object would never be considered inaccessible.) If a UID is not found in the AOD, a new entry is made and an object descriptor is allocated. The object itself is not activated until an attempt is made to access the representation of the object.

In the passive space, the mapping of a UID to an object is accomplished using the **passive object directory** (POD). An object's POD entry consists of its UID and its physical location, e.g. a disk address. Part of a UID contains an index into the POD. Hence, given a UID it is simple to find the corresponding POD entry. Since the POD is typically too large to keep in primary memory, a directory of the POD is kept in memory, and a software implemented cache scheme is used to keep the most recently used POD blocks in primary memory.

4.3. The Uniqueness of UIDs

Since a passive object on one device may contain a passive access descriptor (UID) for an object on another device, and devices may be moved from one system to another, the uniqueness of UIDs must be assured in order to prevent protection violations. To see how this is done in iMAX, we will examine the format of a UID in detail.

A UID is 80-bits in length and consists of four parts: (1) a logical-device name, (2) an index into the POD of the device, (3) a generation number, and, (4) a 16-bit checksum of the first three fields. The purpose of the checksum is to prevent the transformation of a data error into a protection violation and to differentiate between a corrupted access descriptor and the case of an object that has been destroyed before all access descriptors for it have been deleted.

Two objects that have non-intersecting lifetimes may be assigned the same POD index. However, the two objects will have different UIDs since their generation number will be different. When an object is assigned a UID, the object filing system guarantees that the UID's generation number will be different from all UIDs, past and present, that share the same POD entry. The size of the POD index field is 24 bits. The size of the generation field is also 24 bits. The size of these fields is sufficient to assure the uniqueness of a UID within a device over a reasonable lifetime (100 years).

The name of a logical-device consists of two parts, an ASCII name chosen by a system administrator and an 8-byte random number chosen by the system when the device is created. The object filing system presumes that this is sufficient to guarantee the uniqueness of device names within an acceptable probability. The uniqueness of device names and the large number of identifiers within a device assures the uniqueness of a UID.

Since the logical-device name part of a UID is long, a UID actually contains a short 16-bit device ID. This ID is an index into a device name table, one per logical-device. Since different logical-devices will have different device name tables, the object filing system maintains a set of tables to efficiently translate between device-relative UIDs and system-relative UIDs.

5. Extensions to the Hardware Mechanism

The object filing system extends the object-oriented addressing and protection mechanisms of the 432 hardware. The extensions are made to deal with the problems of supporting a passive object space and do not change the active space semantics.

5.1. Composite Objects

To improve the performance and reduce the overhead associated with an object filing system, we introduce the notion of a composite object. A **composite** object consists of one or more objects. A composite object has a single root object. The other objects of a composite are called **components**. Each component of a composite is reachable from the root via an access path contained within the composite. Only the root object has a UID and can therefore be referenced by passive access descriptors contained in other composite objects. Hence, an object should not be made a component of a composite unless it is semantically part of the composite.

Since a composite object is stored in a manner comparable to storing a single large object, The activation (or passivation) of a composite object is more efficient than would be the separate activation of each object contained in the composite. Also, a composite object can be stored more efficiently than storing each element of the composite individually. Since a composite object is assigned only one UID, the overhead of the POD and, more importantly, the AOD is less.

Composite objects solve the small object problem in the passive space. The bulk of a 432 program, i.e. the non-sharable part, can exist as a single composite object, sometimes consisting of hundreds of objects. Hence, the loading of a such a program is one or two orders of magnitude faster than if each object of the program had to be loaded individually.

5.2. Object Space Management

The major space management issue in both the active and passive spaces is knowing when to destroy an object so that its space can be reclaimed. Other designs have used either a **reference-based** approach or an **ownership** approach. In the former, an object is destroyed only if it can no longer be referenced. Reference counts, a parallel garbage collector, or both are used to locate objects to be destroyed. Since reference circularities can exist, reference counts can not form the sole basis of object reclamation. In the ownership approach, the owner of the object can explicitly destroy the object. The major disadvantage of this scheme is that explicit deletion of objects is prone to dangling reference problems. Also, ownership schemes are difficult to implement in a dynamic environment.

Based on the disadvantages of the ownership scheme, we chose the reference-based scheme to manage the active object space, i.e. iMAX implements a parallel garbage collector. However, we chose an ownership scheme for the passive space, because we found garbage collection to be inappropriate for this space for three reasons.

First, object space is reclaimed only at the end of a garbage collection pass. On large devices, a garbage collector may not be able to keep up with the rate at which garbage is being created.

Second, since we allow inter-device references and transportable devices, garbage collection becomes more complex, if not undo-

able. For example, the only references for an object on one device may exist on a device that currently resides on a different system. Although it may be possible to deal with such situations, [10] some objects may exist, if not forever, then much longer than they should.

Last, with no concept of ownership, it is difficult to determine the cause of a full disk condition. This is due to the spreading of access descriptors, an inherent problem in capability systems. Since it is difficult to track down all the access descriptors for unneeded objects and delete them, unneeded objects will not always be reclaimed.

For these reasons, we chose a scheme based on ownership. In this scheme, an **owner** right is defined in passive access descriptors. An object will be deleted from the passive space when there are no outstanding access descriptors for the object with owner rights. Since we expect that the number of passive access descriptors with owner rights for a given object will be small (almost always 1), they will be handled in a special way to facilitate object reclamation. Specifically, the representation of a passive object, say P, will have a list of other objects which contain passive access descriptors with owner rights for P. When this list becomes empty, P will be destroyed.

Since circularities can occur, waiting for the list to become empty is not sufficient, i.e. a garbage collector is still needed. The garbage collector is responsible for detecting passive objects that are unreachable through an access path consisting of "owner" access descriptors. Since only owner access descriptors are considered, the garbage collector involves less overhead than one that would look at all access descriptors. Also, since the generation of such unreachable objects is likely to be rare, garbage collection can be run infrequently.

An implicit assumption in the above discussion is that a passive access descriptor with owner rights cannot reference an object contained on a different logical-device. If this were allowed, this garbage collection scheme would not be any less difficult or costly than the one without owner access descriptors.

5.3. Linkage

As so far described, an access descriptor contains an absolute name for an object. That is, no provision is made for an access descriptor that logically names an object such that at different times it might reference different objects. Lacking this provision has two consequences which we will describe by two examples.

First suppose that a user program has an access descriptor for a system program (e.g. a data base manager). When a new version of this system program supplants the old version, we would want the user's program to reference the new one without causing the user to recompile or relink.

Second, consider the effects of transporting the above user program to a different system, e.g. moving a disk pack between two systems. The program would not run correctly, because it would try to invoke the system program (i.e. the data base manager) on the originating system.

Provision for logical naming could be made by interposing a **link** object between a passive access descriptor and the object it references. On activation the link object would be evaluated, so that the program would have an active access descriptor for the object.

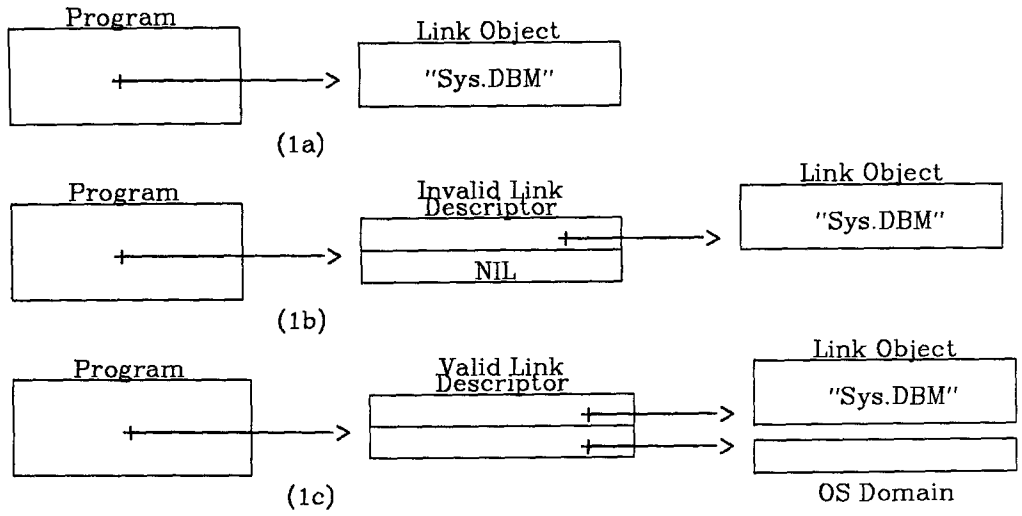
Provision for dynamic linking at activation time is not a sufficient solution. When the above described user program is passivated, the passivated program will have an absolute access descriptor for the system program unless a link object is inserted at passivation. We refer to this as **dynamic unlinking**.

We will first describe our preferred solution, which was not implementable due to hardware constraints. We will then describe our actual solution.

5.3.1. The Preferred Linkage Solution

A solution to both the dynamic linking and unlinking problem is to insert a more permanent link object between an access descriptor and the underlying object. This could be provided by a "link" object descriptor. Such a descriptor would contain two access descriptors, one for the link object and one for the underlying object when the link object was evaluated. When a link object descriptor is valid, it would be invisible; i.e., access to the representation of an object would result in the automatic traversal of any intervening link object descriptors.

Figure 1 depicts the usage of a link object descriptor to implement link objects. In figure 1a, a program has an access descriptor for a link object (which contains the name of



Linkage Using Link Object Descriptors
Figure 1

a system program). Figure 1b is a picture of the program when it is activated, i.e. a link object descriptor is created -- the link object is not yet evaluated. When the program tries to call the system program, a process-level fault will occur. This will cause the evaluation of the link object and make the link object descriptor valid. This results in figure 1c. Hardware will automatically traverse the link object descriptor. When the program is passivated, the link is undone, resulting again in figure 1a. The functionality described allows multiple links for the same object to co-exist even through independent activations and passivations.

5.3.2. The Actual Linkage Solution

Unfortunately, link object descriptors are not implementable with the present 432 hardware implementation. Hence, a less functional but compatible link design is implemented.

When an access descriptor for a link object is activated, the link will be evaluated. This will result in an absolute access descriptor to the object, i.e. no indirection through a link object descriptor. This implies that when an access descriptor for an object is passivated, the link object used in the activation cannot be recovered. To deal with this problem, we permit the owner of an object to assign a link object to his object. When an access descriptor without owner rights is passivated, the link object is copied into the composite object containing the passivated access descriptor. It is quite possible that this link object is not the same one that was used for activation.

5.3.3. Link Objects

We have been using the term "link object" quite loosely, we will now be more precise. A **link** object is an object whose type definition object has the "link" attribute. This attribute can be assigned programatically to any type definition object. This attribute has significance at activation; i.e. when an access descriptor for a link object is activated, the link is evaluated. Hence, link objects themselves are never activated.

This linkage mechanism solves the naming problems described above except for the problem of references to type definition objects. Every typed object, including link objects, contains an access descriptor to its type definition object. When a composite object is activated, the component link objects are evaluated. That is, an access descriptor for a link object is sent to a port for evaluation. An access descriptor for this port is found by examining the type definition object referenced by the link object. However, if the composite object being activated originated from another system, the referenced type definition object would be in the originating system and not accessible. Consequently, the port could not be found and the link object could not be evaluated.

This problem is solved in iMAX by recognizing a distinguished set of type definition objects. These are assigned UIDs that are recognized by iMAX and these are the same on all iMAX-based systems. The linkage types that are supported by iMAX are part of this set.

5.4. Synchronization

Several applications that will use object filing require primitives to synchronize access to shared objects. These primitives could be built on top of object filing or as part of object filing. We have chosen to do the latter because the concepts of consistency and synchronization are related and treating them independently would likely result in a non-uniform user interface.

Our synchronization strategy is based on the notion of Reed's atomic actions. [18,19] Atomic actions allow a user to update a collection of composite objects in a consistent manner. Specifically, the changes made to such a collection is not visible until the atomic action is committed. When this happens all changes appear to happen simultaneously. If an atomic action is aborted, no changes are made to the objects that are part of the atomic action.

Reed describes the basic philosophy of his scheme as follows: "Updating an object will be thought of as creating a new version, while reading an object will be thought of as selecting the proper version and obtaining a value." Thus, the implementation of atomic actions involves creating, maintaining, and providing access to the collection of object versions.

A literal implementation of Reed's scheme would be rather inefficient. [20] Consequently, we have made a number of modifications that improve performance and storage space utilization, and reduce the complexity of implementation. On the other hand, in making these modifications we also give up some flexibility and functionality. A description of our scheme follows.

5.4.1. Implementation Strategy

As in Reed's scheme, timestamping is used for synchronization. Each version of an object has two associated times, when the version was written, and, when the version was most recently read. At the beginning of an atomic action, a **pseudo-temporal environment** (PTE) object is created. A PTE has an associated timestamp. A PTE is specified in all operations on an object and it is the PTE's timestamp that is reflected in the two timestamps of an object version. When a PTE is created, a timeout for the transaction is specified. If the transaction fails to be committed in the allotted time, it will be aborted. An atomic action is committed by calling the Commit operation and passing an access descriptor for a PTE. An atomic action can be aborted by calling the Abort operation.

Our first modification is to allow only one write operation to be in progress for an object at a time. With this simplification, only the most recently committed version of an object needs an associated read time. The read time of an older version is implicitly assigned to be one time unit less than the write time of the subsequent version. The read time of the most recent committed version is kept in the object's POD entry. Hence, updating the read time of an object version does not affect the version's immutability.

Our second modification is to make Open operations the primitive operations instead of individual reads and writes. An Open operation specifies a PTE, an object to be accessed, and an intent (read, write, or update). An Open for reading will select the most recent version of those whose write time is less than the timestamp of the PTE specified in the Open. If a new version for the object is being written and its write time is earlier than the PTE timestamp of the Open for reading, the Open will block until the new version is committed or aborted.

An Open for writing will produce a new version of an object. The timestamp of the PTE specified in the Open must be later than the read time of the most recently comitted version. If it is not, the Open will be refused. Otherwise, the Open will be allowed, but will wait until the uncommitted version is either aborted or committed. Open operations that block will be enqueued in order of the timestamps of their associated PTEs.

Our third modification is to delete committed versions when they are replaced by a newer commited version and when they are no longer being actively accessed (i.e., there are no outstanding Opens). For example, when version J of an object is committed, version I of the object will be destroyed if it is not being accessed. Thus, version deletion times are well-defined. This has the disadvantage of reducing potential parallelism and causing transactions to abort. For instance, if an Open for reading is done using an old PTE, the requested object version might have already been deleted. This would cause the Open to fail, and typically, the transaction to be aborted.

Our fourth, and final, modification is to implement sharing between versions of the same object. That is, we exploit the immutability of versions by allowing them to share storage, e.g. disk sectors. When a version is to be deleted, its list of disk addresses can be compared with the subsequent version to determine which disk areas can be freed.

5.4.2. Commit

Committing of object versions in iMAX is simpler than the general scheme of Reed's. In our first release of object filing we are only concerned with atomic actions within a single processing node. In a commit of a transaction that involves a single object, i.e. the typical case, only the POD entry of the object needs to be changed. To make this change atomically and to increase POD reliability, the POD is implemented in **stable storage**. [21] To handle multiple object commits, the value of the POD entries to be changed are first written as a single commit record to stable storage. After this completes, the appropriate POD entries are updated. When this completes, the commit record is deleted. Once the commit record is successfully written, the transaction is committed. If a system crash occurs, the commit records will be processed in order of timestamps as part of the system start-up procedure.

5.5. Data Files

The object filing system incorporates support for simple data files. iMAX supplies a type manager for objects of type **data-file**. Data-file objects exist only in the passive space and do not have the 128K byte limitation as do other 432 objects. Since the data-file type manager is integrated into object filing, a more efficient implementation is possible than if a separate type manager implemented this type.

The opening of a data file is implemented as the opening of the data-file object's passive definition. Such a file open will return a stream-like I/O interface package. [15] File synchronization and consistency are accomplished by atomic actions.

6. An Example of Object Filing on Programming

Object filing provides new programming semantics that can be easily exploited by users through high-level languages such as Ada. [22,23] One example of this is object filing's ability to save the state of a collection of objects, e.g. a computation. This facility can provide a programming workspace construct similar to that supplied by APL and LISP programming systems.

An Ada programmer on the 432 can choose a workspace environment for a data abstraction by properly structuring his Ada program. This is best illustrated by an example. Suppose a user program consists of a package P which contains variable declarations and subprograms, and a procedure Main (the main program). We will examine two possibilities: both P and Main are defined as library units; and, P is defined within Main, i.e. only Main is a library unit.

First suppose that both are library units, Main is invoked, and Main calls subprograms within P which change the variables declared in P. The use of Main and P cause their activation. Sometime after Main completes, Main and P will be passivated. The passivation of Main is a no-op since a procedure does not have any state. P does have state, namely, the variables declared in P. Changes made to these will be reflected in the passive definition of P when passivation occurs. When Main is invoked again, causing a reactivation of Main and P, the state of P will not be the same as when Main was first invoked. This is the workspace form of a data abstraction.

In the second case, package P is defined within the procedure Main. When Main is invoked, a new package P will be created. This package will be deleted when Main is exited. That is, a different instance of P is tied to each invocation of Main. If two processes call Main, each will have a different package P, i.e. they will not access the same variables within P. This does not preclude sharing; i.e. the compiler supports the sharing of constant objects, specifically, the instruction objects that make up instances of P. This form of program structuring provides the programming semantics supplied by traditional systems.

In this example, we showed that the data state of an abstraction can be saved in the object filing store. Object filing is also able to save the control state of an abstraction, i.e. save a process object. This facility is provided by the iMAX process manager, the type manager for process objects.

7. Summary

We have described the key elements of the iMAX object filing system. In this section, we review how these provide solutions to the problems that we stated at the top of this paper.

The first problem was assuring the uniqueness of UIDs. This was solved by guaranteeing the uniqueness of a UID within an object filing device and by reducing the probability to an acceptable level, of any two object filing devices having the same name.

The second problem was providing an object-oriented, logical naming mechanism. We described a link object mechanism that provided for dynamic linking at activation and dynamic unlinking at passivation.

The third problem was maintaining the consistency of a collection of objects. This was solved by: (1) decomposing the object space into two spaces, active and passive, and giving type managers control of the movement of their objects between the two spaces; (2) providing the facility of composite objects which allows a type manager to treat a complex object as a single entity; and, (3) supplying atomic actions to deal with a collection of composite objects.

The fourth problem was the management of the object space. Although other designs have relied solely on a parallel garbage collector to reclaim object filing space, we argued against this approach. In its place, we described an ownership scheme that is more efficient in reclaiming object space.

The last problem we posed dealt with the implicit overhead of an object filing system. To address this problem we introduced the notiqn of composite objects. The use of composite objects reduces the space requirements for the AOD and POD. It also speeds up activation and passivation by involving more objects in each physical I/O operation.

To summarize, we undertook the implementation of an object filing system for iMAX in order to meet our objective of providing to our customers (mainly original equipment manufacturers) an extensible operating system. This is made possible by the architecture of the 432 and by the choice of Ada as our systems implemenation language. Extensibility is achieved by the addition of new object types and their managers. The 432 architecture provides for these in an efficient manner. The iMAX object filing system complements the other parts of the 432 system by providing for the permanent storage of objects and giving each type manager exclusive control over the creation, preservation, and ultimate destruction of the objects it manages.

Acknowledgements

The iMAX design has benefited from the first author's involvement in the design and implementation of the Hydra object filing system. Conversations with Bill Wulf, Guy Almes, and Roy Levin were useful in determining the weaknesses of the Hydra design and in identifying the difficult problems to be solved in the design of an object filing system.

This work has also benefited from the creative and cooperative environment generated by all of the members of Intel's Special Systems Operation. Particular note should be paid to Bill Corwin who provided detailed criticisms of two earlier designs and challenged us to find better solutions. Justin Rattner, George Cox, and Steve Zeigler have supplied motivational support. Steve Tolopka and Karl Deiretsbacher have recently joined us in our implementation efforts.

We would also like to thank Anita Jones, whose many comments and suggestions helped to enhance the clarity of this paper.

References

1. Intel 432 GDP Architecture Reference Manual, Intel Corporation 1981.
2. IBM System/38 Technical Developments, International Business Machines Corporation 1978.
3. D. M. England, "Architectural Features of System 250," pp. 395-428 in *Infotech State of the Art Report 14: Operating Systems*, Infotech International Ltd., Maidenhead, Berkshire, England (1972).
4. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications ACM* **17**(6) pp. 337-345 (June, 1974).
5. W. Wulf, R. Levin, and S. Harbison, *Hydra: C.mmp: An Experimental Computer System*, McGraw–Hill Book Company (1981).
6. M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operating System*, Elsevier North Holland (1979).
7. C. Dellar, "Removing Backing Store Administration from the CAP Operating System," *Operating Systems Review* **14**(4) pp. 41-49 (October, 1980).
8. J. Dion, "The Cambridge File Server," *Operating Systems Review* **14**(4) pp. 26-35 (October, 1980).
9. P. Bishop, "Computer Systems with a Very Large Address Space and Garbage Collection," M.I.T. Laboratory for Computer Science Technical Report TR-178 (May, 1977).
10. R. Halstead, "Object Management on Distributed Systems," *Proceedings of the 7th Texas Conference on Computing Systems*, pp. 7-7:7:14 (October, 1978).

11. J. L. Gula, "Operating System Considerations for Multiprocessor Architectures," *Proceedings of the 7th Texas Conference on Computer Systems*, pp. 7-1:7-6 (October, 1978).
12. G. T. Almes, "Garbage Collection in an Object-Oriented System," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University (June, 1980).
13. A. P. Batson and A. W. Madison, "Characteristics of Program Localities," *Communications of the ACM* **19**(5) pp. 285-294 (May, 1976).
14. K. C. Kahn and F. J. Pollack, "An Extensible Operating System for the Intel 432," *Proceedings Compcon Spring 1981*, pp. 398-404 (February, 1981).
15. K. C. Kahn, W. M. Corwin, T. D. Dennis, H. D'Hooge, D. E. Hubka, L. A. Hutchins, J. T. Montague, F. J. Pollack, and M. R. Gifkins, "iMAX: A Multiprocessor Operating System for an Object-Based Computer," *Proceedings of the 8th Symposium on Operating System Principles*, ().
16. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. M. F. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications ACM* **21**(11) pp. 966-975 (November, 1978).
17. G. W. Cox, W. M. Corwin, K. Lai, and F. J. Pollack, "A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment," *Proceedings of the 8th Symposium on Operating System Principles*, (December, 1981).
18. D. P. Reed, "Naming and Synchronization in a Decentralized Computer System," Ph.D. Thesis, M.I.T. Department of Electrical Engineering and Computer Science (September, 1978). Also available as M.I.T. Laboratory for Computer Science Technical Report TR-205
19. D. P. Reed, "Implementing Atomic Actions on Decentralized Data," *Preprints of Proceedings of the 7th Symposium on Operating System Principles*, pp. 66-74 (December, 1979).
20. D. P. Reed and L. Svobodova, *SWALLOW: A Distributed Data Storage System for a Local Network*, Submitted to the International Workshop on Local Networks in Zurich, Switzerland August, 1980.
21. H. E. Sturgis, J. G. Mitchell, and J. Israel, "Issues in the Design and Use of a Distributed File System," *Operating Systems Review* **14**(3) pp. 55-69 (July, 1980).
22. S. Zeigler, N. Allegre, R. Johnson, J. Morris, and G. Burns, "Ada for the Intel 432 Microcomputer," *Computer* 14(6) pp. 47-56 (June, 1981).
23. Reference Manual for the Ada Programming Language, U.S. Department of Defense Proposed Standard July, 1980.