PROJECT MAC 12/20/75

Computer Systems Research Division Request for Comments No. 101

A CASE STUDY IN OPERATING SYSTEM DESIGN: RECONCILING STRUCTURE WITH EFFICIENCY

by Douglas H. Hunt

In a previous report (RFC 73) the author described an operating system interface which supports extended type objects protected by access control lists. This report describes in more detail how that operating system might be modularized. A major objective of this report is to show that that operating system, with functional capabilities comparable to those of Multics, can be organized as a collection of layered and object-oriented modules. Such an organization is feasible because there are techniques for achieving strict layering without sacrificing economy of mechanism. Some of these techniques are described in this report.

Introduction

In recent years, considerable research activity in the field of computer systems has been directed towards developing systems which are certifiable, or perhaps even verifiable. There are at least two motivating factors for this research; first, the need for more reliable operating systems, and second, the need for greater assurance that the protection policies of a computer system cannot be circumvented. Research into developing certifiable (or even verifiable) systems is occurring on several fronts. One approach is to improve our body of knowledge in the area of module specification techniques, and proofs which demonstrate that programs correspond to specifications [ref. SRI work]. Another approach is to work towards improved modularization of systems, employing techniques such as layering, or levels of abstraction, [ref. "THE" system] and object-orientation [ref. Hydra].

The research described here, which is a part of the author's proposed Ph.D. thesis, is an example of the second approach -- improving the way we modularize systems. In a previous report (RFC 73) the author described an operating system interface which supports extended type objects protected by access control lists. Except for the type extension facility, the system described was similar to Multics [ref. Organick]. This report describes in more detail how that hypothetical operating system might be modularized. A major objective of this report is to show that an operating system with sophistication comparable to that of

Multics can be organized as a collection of layered and object-oriented modules, all the way down to the hardware (or firmware) interface. The reason that such an organization is feasible, in the opinion of the author, is that there are techniques for achieving strict layering without sacrificing economy of mechanism. Some of these techniques are described in this report. (1)

This report is organized as follows. First, an object-oriented model for describing the multiplexing of memory resources is introduced. The objects of the model resemble LISP objects; in particular, the objects have bindings designating other objects. Operations on the objects manipulate the bindings. These objects, together with the procedures which manage them, provide an abstraction of memory called a block space. Second, a particular implementation of the block space model is described. By considering the objects which provide the implementation to be objects of the model themselves, the model can be cascaded to provide a more sophisticated abstraction. Third, a particular implementation of the system "map" -- a data structure which yields attributes of an object given its name -- is described. It is significant that the programs which provide the "map" function make use of the block abstraction, and as a consequence the they are reasonably simple. Fourth, a problem

(1) The layering notion does not imply that there be a strict linear dependency relation between modules; there can be partial orders. In fact, two of the layers described in this report are independent.

encountered in building layered systems -- achieving strict layering without "re-inventing the wheel" -- is cescribeo. A design principle which provides a solution to this problem is suggested and exploited.

Definitions and Assumptions

We assume that every object has a name; e.g. the name of a memory cell is its address. A segment is defined to be like a Multics segment [ref. Bensoussan, Clingen, and Daley], except that it has no attributes and that its name is a system-supplied unique identifier. A principal is an object which is given the authority to reference other objects, and is described further in the paper by Saltzer and Schroeder [ref. Saltzer and Schroeder IEEE paper]. We will almost always use the word "principal" rather than "process" or "domain". The main reason we do so is to defer, until a later stage of the research, any assumptions about the relationship of processes and domains; i.e. whether processes and domains are equal, or whether multiple instances of one might be contained in the other. This issue is discussed in a paper by Lampson and Sturgis [ref. Lampson and Sturgis SIGOPS paper]. Consequently, the mechanics of changing a protection context will not be addressed in this report.

Finally, it is important to emphasize that some of the ideas presented here have been only partially explored. In addition, there are some topics which are definitely relevant to this work but which, for time and space reasons, are not included here.

No attempt is made to determine what the "lowest" or "most primitive" objects in the system ought to be; the most primitive objects to be described here are understood well enough that a consideration of their implementation would not provide any new insights.

A Lisp-like Object World

The first object to be described is called the page object. A page is a fixed-sized collection of bits. The bits are generally grouped into units called bytes or words. Since the name of a page object is not actually used in this model, it would not be necessary to devise a way of naming pages. In order to conform to the view that every object has a name, however, let the contents of a page be its name. If a page contains K bits, then there are 2**K page objects, each with a distinct K-bit name.

A second type of object is the home object. A home object is an abstraction of a storage region in secondary storage which has enough capacity to hold a page object. A home object has (1) a name, (2) a binding, and (3) a property list. As mentioned before, this model bears a strong resemblance to the LISP object world. The name of a home object is its address in secondary storage. For example, in a system which uses disk drives to provide secondary storage, the name of a home object might consist of the concatenation of (1) a controller number, (2) a device number, (3) a cylinder number, (4) a track number, and (5) a record number. The binding of a home object designates a page object. Every home object is bound to some page object; in general more than one home object may be bound to the same page object. Since the (only) binding of the home object designates a page object, the binding is called the page binding of the home object. Properties of home objects and operations on them will be described as they are motivated.

A third type of object is the <u>frame</u>. A frame object is an abstraction for a contiguous region or primary memory which has enough capacity to hold all of the bits in a page object. The name of a frame object is the absolute address of the first addressable unit (usually a byte or a word) of the frame. Frames have bindings and a property list. Unlike homes, frames have two bindings: a page binding and a home binding. The page binding designates a page object. The home binding designates either a home object or a meta-object called NULL. It is possible that the page binding of two or more frame objects may designate the same page; however, non-null home bindings must designate distinct home objects. Examples of binding relationships are

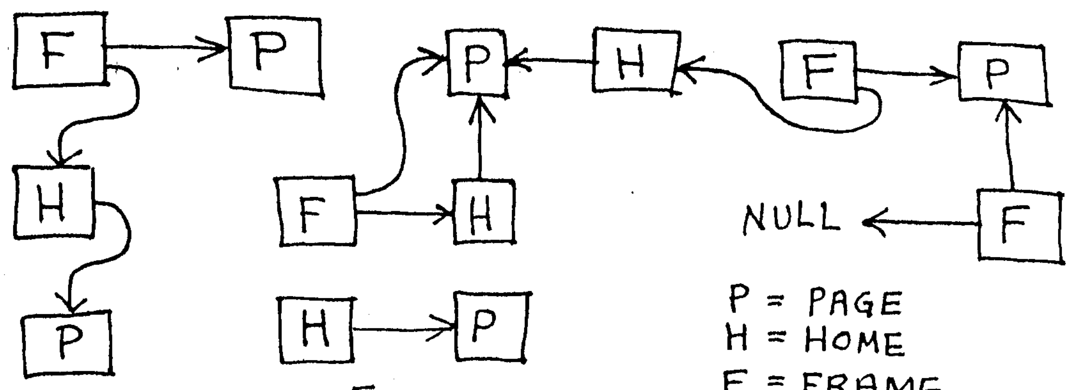

Figure I

depicted in Figure I. When we refer to information "in" a frame or home object, that is an informal way of referring to the page designated by the page binding of the frame or home object.

The page, home, and frame objects which have been described are all used to provide the abstraction of a paged memory. The collection of modules which support paging will multiplex frame objects among various home objects to provide a composite object which retains the advantages of homes and frames, but not their disadvantages (see Table I).

	<u>advantages</u>	<u>disadvantages</u>
homes	many in number	slow access in any implementation
frames	fast access in any implementation	few in number

Table I

This composite object is called a <u>block</u>. The collection of modules which multiplex frames among homes to provide blocks will be called the <u>block layer</u>. Any data (other than that in hardware registers) referenced by programs outside the block layer must be in some block. Blocks are named by nonnegative integers. Blocks have two bindings: a home binding and a page binding. Programs can call the block layer to set the home and page bindings of a block. Thereafter, the programs can reference (load from and store into) the block directly. Thus, programs which execute outside the block layer use two part addresses of the form

    blockname ! offset

This sort of addressing environment is similar to that provided by the TENEX operating system. (1) Programs which use this addressing environment need not perform direct I/O. The set of blocks which a program references which in turn have a non-null home binding are called a block space. Block objects will be described further in a later section.

To see how the virtual address space is supported, we now describe the operations defined on home and frame objects.

The operations on home objects are:

1) read (home_name, frame_name), and

2) write (home_name, frame_name).

The effect of the <u>read</u> operation is to replace the page binding of "frame_name" by the page binding of "home_name". The effect of the <u>write</u> operation is to replace the page binding of "home_name" by the page binding of "frame_name". Thus, the read and write operations are defined jointly on home and frame

---

(1) In TENEX, the blocks are considered to be concatenated so that they form a single linear space. To reference

    blockname ! offset

in TENEX, a <u>single</u> address of the form

    blockname * 1000(octal) + offset

is presented.

objects. The read and write operations can be performed only by programs of the block layer.

Operations defined on frame objects are

1) fetch (frame_name),

2) store (frame_name, binding),

3) assign (frame_name, home_name), and

4) release (frame_name).

The fetch operation returns the current page binding of "frame_name", while the store operation replaces the page binding of "frame_name" by "binding". It should be clear that processor instructions which fetch from and store into frames can be modelled as functions which respectively obtain, and replace, the page binding. The assign operation replaces the home binding of "frame_name" by "home_name". The release operation replaces the home binding of "framename" by NULL. The operations on frame objects can be performed only by programs in the block layer.

Implementing the Bindings of Objects

The description of pages, homes, and frames and their bindings has been given, up to this point, in terms of an abstract model. We now consider how these bindings might be represented. The representation of the page bindings of homes and frames are implicit, in the sense that the bindings are the "contents" of the object. The home bindings for all frames must be represented in some other way. In fact, they are maintained in a table which is itself an object. There is a subset of the set of available frames which is called the set of _interpreted_ frames. There is also a subset of the set of available homes which is called the set of interpreted homes. Informally speaking, interpreted frames and homes are those whose page bindings the block layer cares about; i.e. those which the block layer expects to interpret. There is, for example, a particular sort of interpreted frame in which the block layer expects to find both homenames and framenames. This kind of interpreted frame will be called a _B-frame_ (for "block layer" frame). A B-frame is used to represent the home bindings of frames. Thus a B-frame contains a sequence of ordered pairs of the form

< framename, homename >

in which the first component may be the meta-name NULL. A null framename field means that there is no frame which has the home binding "homename". A given B-frame represents the home bindings of only a subset of all the frame objects. Both framenames and homenames are unique over the set of all B-frames. Figure II illustrates several B-frames which represent home bindings of frames. To find the home binding of a given frame, it would be necessary to search all the B-frames.

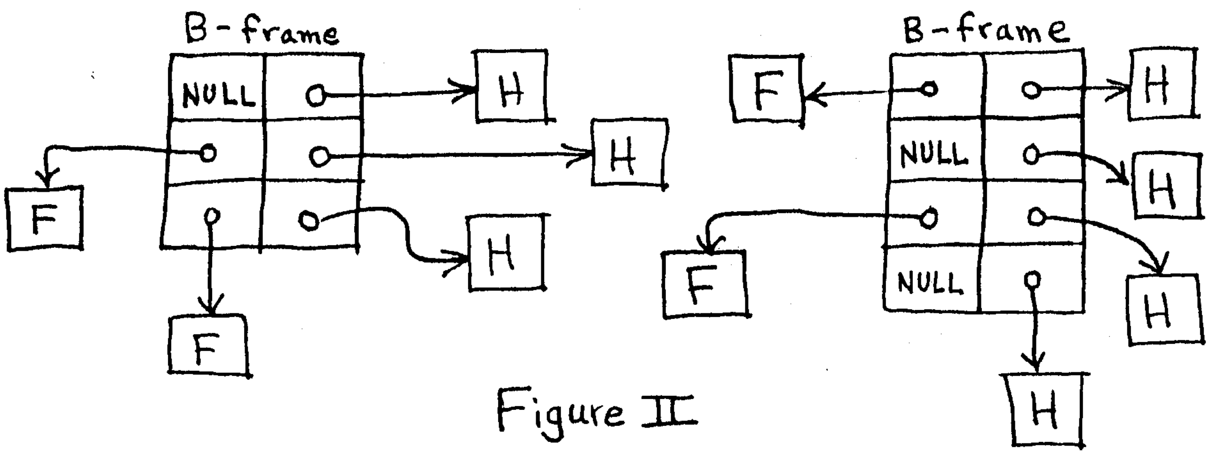

Figure II

We can now describe how the assign and release operations affect B-frames. The effect of

    assign (frame_name, home_name)

can be described as follows.

1. Find the B-frame with "home_name" in it. If "home_name" is found but it already has a corresponding framename, the assign operation returns without changing any bindings.

2. If "home_name" cannot be found, the assign operation returns without having any effect. The homename must exist in some B-frame in order for the assign operation to complete.

3. Set the framename corresponding to "home_name" (it must be null) to "frame_name".

The effect of the operation

    release (frame_rame)

is defined as follows.

1. Find the B-frame with "frame_name" in it. If "frame_name" cannot be found, the operation returns without changing any bindings. Either the frame has already been released or it does not exist.

2. Replace "frame_name" in the framename field by NULL.

Implicit in the description of the assign and release operation is a free list of frames. Frames which have null home bindings are on this free list. Thus, prior to the assign operation, the frame called "frama_name" would have been removed from the free list. After the release operation, "frame_name" would be added to the free list of frames.

Implementing Blocks in Terms of Lower-level Objects

At this point we are ready to describe the operations on block objects and how the page and home bindings of block objects are represented. The operations on a block object are

1) fetch (block_name),

2) store (block_name, binding),

3) initiate (block_name, home_name), and

4) terminate (block_name).

The fetch and store operations on blocks are analogous to the corresponding operations on frames. The initiate operation replaces the home binding of "block_name" by "home_name". The terminate operation replaces the home binding of "block_name" by NULL. These four operations, unlike any of the preceeding operations, can be performed by programs which are outside the block layer.

The versatile B-frames represent not only the home bindings of frames, but also the home bindings of blocks. Any particular B-frame, which holds N ordered pairs as described above, can be used to support a block space which comprises N blocks. Consider one B-frame, and a program executing in the block space supported by this B-frame. Initially, both components of each ordered pair in the B-frame are null. Initiate operations set the homename parts; for example

```
initiate (5, home_name_67)
```

sets the homename part of the 5th ordered pair to "home_name_67"; i.e. the home binding of block 5 is now the home whose name is "home_name_67". The framename part of an ordered pair is set by the assign operation. That is, the block layer does the operation

```
assign (frame_name_12, home_name_67)
```

This sets the home binding of "frame_name_12" to the home named "home_name_67". These relationships are shown in Figure III.

Recall that a home can be designated by the home binding of

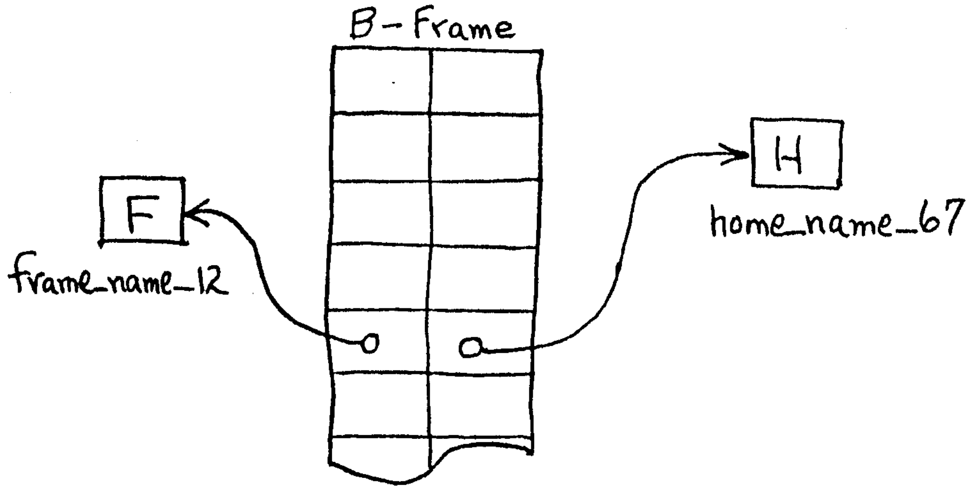

Figure III

at most one frame. Consequently, there is a well-defined concept of a "frame binding" of a block: if the home binding of block B designates home H, and if there is a frame F for which the home binding also designates H, then the frame binding of B is F; otherwise it is null. From the point of view of users of blocks, the notion of a frame binding is hidden. For descriptive purposes, however, such a notion is useful. If the "frame binding" of a block is mentioned, the meaning will be as defined in this section. We use this definition to state that the page binding of a block is guaranteed to be the same as the page binding of the frame binding of a block. For example, if a home is designated by the home binding of both a block and a frame (as "home_name_67" is in Figure III), then the <u>page</u> binding of the block (block 5 in the figure) always tracks the page binding of the frame ("frame_name_12") in the figure.

The page, home, and frame objects which have been described so far are manipulated by programs in the block layer to support the abstraction of a block space. Each principal will have an associated B-frame which represents the bindings of block objects in its block space. A principal can control which homes are bound to blocks in its block space by means of the initiate and terminate operations. (1) The address (framename) of the B-frame associated with an executing principal is stored in a special processor register. Virtual addresses of the form

```
        blockname ! offset
```

are converted to absolute addresses (framenames) by the simple calculation

```
     address = framename_part (B-frame + W*blockname)  +  offset;
```

where W is the number of words in an entry of the B-frame.

If the framename part of the entry is null, then a frame fault occurs. In this model, the handling of a frame fault will be described as though it is being carried out by a separate process, called the frame claiming process (although the claiming of frames could take place in the faulting process). (2) This frame claiming process uses a block space defined by a particular

---

(1) Unlike other principals, the block layer has a block in its block space whose frame binding is its own B-frame. Thus, the block layer can not only manipulate its own block space, but it can manipulate the block space of other principals; i.e. carry out initiate, terminate, assign, and release operations.

(2) All processes involved in implementing the block space are expected to be like the virtual processors described by D. Reed in his Master's thesis.

B-frame, called the basic B-frame, which describes all the procedures and data bases necessary to handle the frame fault. (1) In particular, the framename of the B-frame of the principal which took the frame fault will be contained in the basic

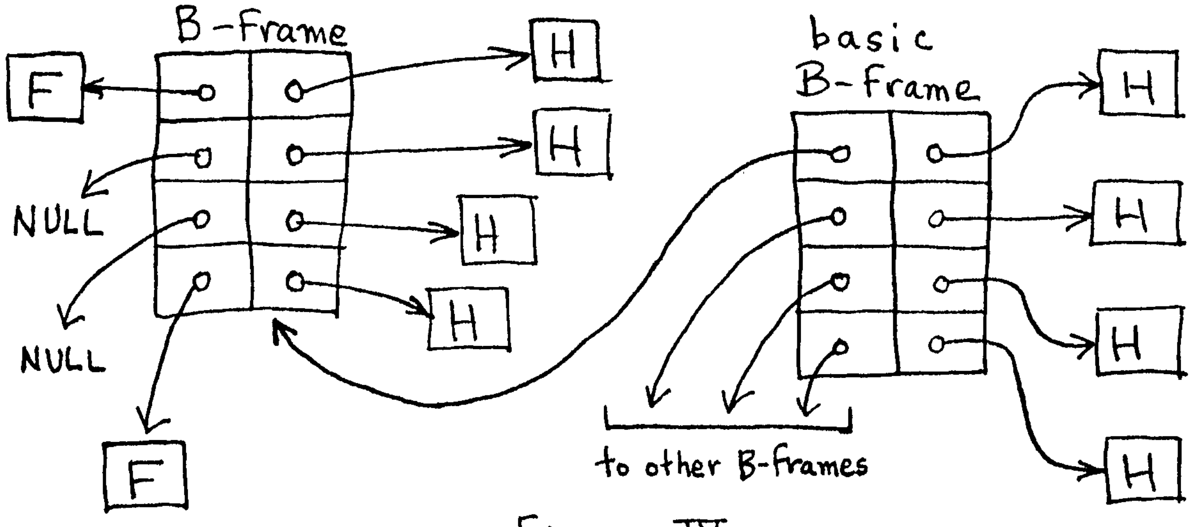

Figure IV

B-frame. These relationships are shown in Figure IV.

The frame claiming process is loosely coupled to another process, called the frame freeing process. The frame freeing process also uses the block space defined by the basic B-frame. It is the responsibility of the frame freeing process to ensure that frames are available for claiming by the frame claiming process. (2)

The frame claiming and frame freeing processes are described briefly. The intent of these descriptions is not to exhibit the

(1) No framename field of the basic B-frame is ever null.

(2) As part of his Master's thesis, [ref. Huber] Huber developed a version of page control for Multics which makes use of a page-freeing process.

algorithms in full detail, but rather to show how they use home and frame objects. Although the descriptions given here suffice only if there is one frame claiming process and one frame freeing process, they could be extended to accommodate multiple processes. First we sketch the steps in the frame claiming process. The arguments are a home, and a B-frame containing the home.

1. frame <-- select (free_frame_list)

   Remove a frame from the free frame list, according to some selection policy.

2. read (home, frame)

   Set the page binding of "frame".

3. assign (frame, home)

   The frame corresponding to "home" in "B-frame" will be set to "frame".

4. enter (assigned_frame_list, frame, B-frame)

   Put the name of the assigned frame and its containing B-frame into a list.

The description of the frame freeing process is similar, except "in reverse".

1. frame, B-frame <-- select (assigned_frame_list)

   Get the frame (and containing B-frame) according to some policy.

2. home <-- release (frame)

   This sets the framename field containing "frame" in "B-frame" to NULL, which will trigger a frame fault on the next reference. The "release" operation returns the home associated with "frame".

3. write (home, frame)

   It is necessary to write only if "home" has beer modified.

4. enter (free_frame_list, frame)

   Enter the frame on the free frame list.

Using the Block Layer to Make Large Blocks

The objects and algorithms which support the block space abstraction are almost sufficiert to support another addressing abstraction which we shall call a "large block space". As the name would indicate, the "large blocks" abstraction is the same as the "blocks" abstraction except that the blocks are larger. In order to implement large blocks, we merely recursively utilize the objects and abstractions already described. That is, let the B-frame which describes a block space of blocks instead describe one large block. Then a collection of such B-frames would describe a block space of large blocks. The collection of B-frames would itself be described by a B-frame. This extension essentially amounts to inserting one more layer of B-frames into the tree shown in Figure IV, producing a tree as shown in Figure V. To distinguish between B-frames which are directly below the

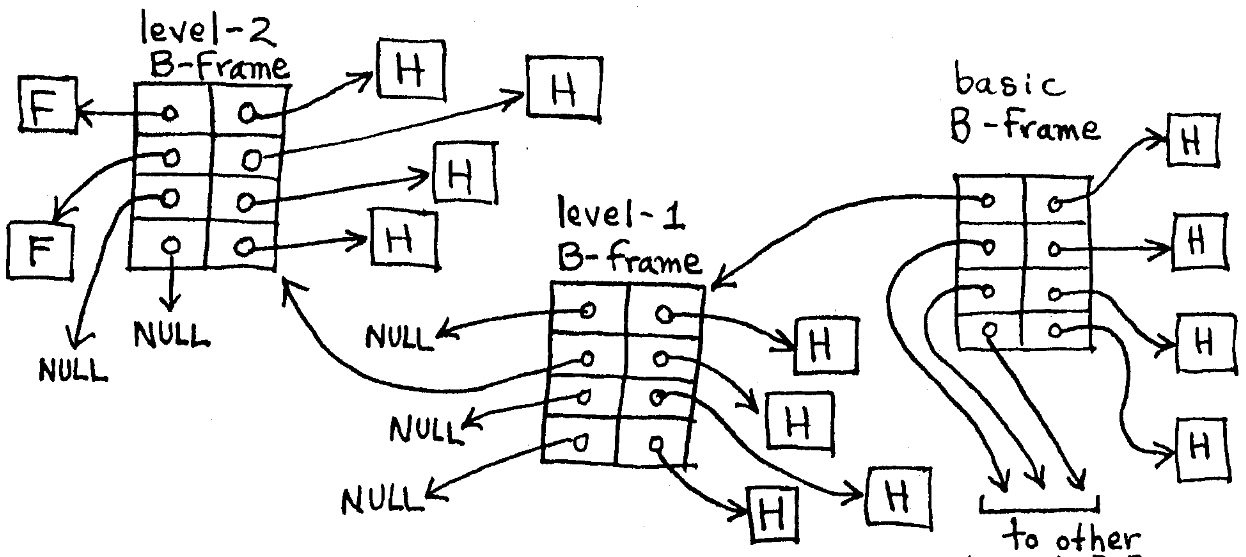

Figure V

root (basic) B-frame and B-frames which are directly above the leaf node frames, we shall call the former group level-1 B-frames and the latter group level-2 B-frames. Unless otherwise specified, it will be assumed that the block space of a "user process" or "user domain" will be described by a level-1 B-frame. Virtual addresses of the form

L ! O

where L is the name of a large block and O is an offset, are translated by the hardware as follows:

1. l2_B-frame = framename_part (l1_B-frame + V * L);

2. frame = framename_part (l2_B-frame + W * [O/F]);

3. address = frame + MOD (O , F);

where "l1" and "l2" stand for "level-1" and "level-2" respectively, V and W are the number of words in an entry of level-1 and level-2 B-frames respectively, and F is the number of words in a frame. In the implementation of large blocks, it is not only possible for leaf node frames to be multiplexed among homes, but also possible for level-2 B-frames to be multiplexed among B-homes. That is, a frame fault (from the point of view of the block layer) may occur in step 1 or in step 2 of the above algorithm for mapping virtual addresses. In a block space described by a level-1 B-frame, there may be a block which has a non-null home binding (designating some level-2 B-home) but which has a null frame binding (that is, no level-2 B-frame is designated). In this case, a fault occurs and is handled by algorithms almost like those given above. To modify the above frame claiming and frame freeing algorithms to handle "level-2 B-frame" faults, first substitute "level-1 B-frame" for "B-frame", "level-2 B-frame" for "frame", and "level-2 B-home" for "home". If we allow non-null entries in the framename part of a level-2 B-home, then no further modifications are needed. Alternatively, all the entries can effectively be null by not storing them in the level-2 B-home at all. In this case, the framename part of each level-2 B-frame can be set to null after a read operation, and it can revert to null (by recursive calls to the frame freer) before a write operation. It is a design objective that the block layer make as little distinction as possible between handling a frame fault for an uninterpreted home and a frame fault for an interpreted home such as a B-home.

Extending Large Blocks to Segments

At this point we have described how the mechanism which implements blocks can, with only a few extensions, implement large blocks as well. This construction could be cascaded further to implement "very large blocks", and so on. However, the goal of this report is to show how this same mechanism, with only a few more extensions, can support a collection of segment objects. Comparing large blocks with segments as we have defined them, we see that large blocks are in fact the same, except that large block names are secondary storage addresses whereas segment names are unique identifiers.

To implement segment objects, there must be a way of assigning unique identifiers (UIDS) to segments as they are created. Further, there must be an efficient way of retrieving the B-home given the UID. Since we must assume that the B-homes of segments cannot be calculated directly from the UIDs, we shall describe a data structure which implements this mapping for all segment objects. Following the terminology of Redell, [ref. Redell] we shall call this data structure the map. A fundamental aspect of the design being presented here is that the programs which search and modify the map are programs which execute above the block layer. The goal is to provide the mapping from UIDs to B-homes while inventing as little new mechanism as possible.

The actual data structure used to implement the map will be a B-tree [ref. Knuth, vol. III]. A B-tree is a balanced n-ary tree which has the property that searching, insertion, and deletion operations all have a guaranteed worst-case efficiency. A B-tree is a structure which is well-suited for external searching; that is, the nodes of the B-tree are implemented as records of secondary storage. The interior nodes of the B-tree are not interpreted homes -- i.e. they are not treated specially by the block layer. However, the collection of programs which manipulate the map (the map layer) does expect the interior nodes to have a format as shown in Figure VI.

```
<homename> <UID> <homename> ... <homename> <UID> <homename>
```

Figure VI

The leaf nodes of the B-tree are level-2 B-homes. The homenames in an interior node are names of other interior nodes, unless the interior node is directly above the leaf nodes, in which case the homenames are names of level-2 B-homes. The UIDs surrounding a homename in Figure VI are lower and upper bounds on all the UIDs reachable in the subtree with the given homename as the root node. Since the program that searches the B-tree executes above the block layer, each time it selects a homename out of one of the interior nodes, it uses the initiate operation to bind a block to the selected home. It then references the block directly, searching for the next appropriate homename. The program can manage its block space so that nodes near the root of the B-tree tend to remain in the block space.

Suppose that the B-tree which provides the mapping from UIDs to B-homes must accommodate 10**8 segment objects. The B-home corresponding to a UID can be located in 3 references to secondary storage if the B-tree is of order 100 (or more), and if the root node is contained in primary memory. The average search time can be reduced if a software-managed associative memory is provided. This associative memory will contain (UID, B-home) pairs. The associative memory will contain enough pairs (probably at least a few hundred) to yield a high hit ratio.

The large block layer and the map layer, both of which depend on the block layer, together provide the segment layer. Above the segment layer, it is possible for programs to perform the operations

initiate (blockname, UID) and

terminate (blockname)

on segments, with "blockname" as an input argument. Initiating or terminating a segment causes the respective operation to be performed on the corresponding large block. For example, the segment layer implements the initiate operation on segments by first mapping the UID argument into a level-2 B-home, and then calling the large block layer to initiate the level-2 B-home.

Two other operations defined on segments are

fetch (UID, offset) and

store (UID, offset, binding).

(In this description we consider "execute" to be a subcase of "fetch".) Since unique identifiers tend to be rather long -- say at least 32 bits -- there is a motivation for referring to segment objects by means of some shorter identifier. Two possibilities are briefly described here.

One possibility is to allow blocknames to be visible above the segment layer. Then the blockname itself could serve as the shorter identifier for the segment. In this case, the segment layer need not re-interpret fetch and store operations. Rather, after a segment has been initiated, the fetch and store operations can be interpreted directly by the large block layer as fetch and store operations on a large block object. In order to relieve user programs of the necessity of managing the available blocknames, a simple layer could be built on top of the segment layer. This simple layer would implement a policy of assigning blocknames to UIDs, and would record these assignments in a per-level-1-B-frame table. Such a table would be similar in function to the Known Segment Table in Multics [ref. Bensoussan, Clingen, and Daley]. In fact, the segment addressing method just described is the same as the method used in the Multics system.

Another possibility for addressing segment objects is to hide the blocknames from any layer which is above the segment layer, and introduce another strategy for assigning shorter names to segment UIDs. The processor could provide a number of UID base registers, which can be loaded with a segment UID and an offset. Then user programs could load segment UIDs into base registers, and refer to segment objects by the register number of the register containing the segment UID. The first reference by a principal to a segment would invoke the map and large block layers to establish a level-2 B-frame corresponding to the segment UID. The name of the level-2 B-frame would be stored in the level-1 B-frame corresponding to the principal, and could be retrieved, given the UID, by a fast search such as a hash lookup. Such an addressing scheme would be supported by a hardware associative memory which returns a level-2 B-frame given a segment UID. This second alternative is similar to the approach adopted for the Plessey 250 system [ref. Plessey system].

An Interaction between Property Lists and Type Extension

Up to this point, we have considered a layered design for implementing "simple" segment objects. As yet, we have not considered attributes of segments. In particular, we now wish to consider access attributes of segment objects. We shall take a list-oriented view of access control, [ref. Saltzer & Schroeder IEEE paper] and therefore consider access control lists. Since it is also our goal to discuss type extension, and since there are interactions between access control (and other) attributes of an object and type extension, we mention this interaction before proceeding.

The basic issue we must address is whether attributes of a segment object are objects in their own right. Specifically, is an access control list (ACL) an object? If so, then a type extension facility seems more fundamental than access control lists. If not, then ACLs can be implemented by some layer which is lower than the type extension facility. There seem to be reasons for favoring both choices. A reason for making the first choice is that an extended type manager (ETM) must be able to determine, using an ACL, if an operation on an extended type object is permitted. On the other hand, an ACL is a special data structure with search, display, and update operations defined on it -- a good candidate for objecthood.

Our solution to this problem is to consider an ACL to be both an attribute of a segment and an object, as it suits our purposes. More precisely, as long as we do not consider the dynamic aspects of access control, it seems quite appropriate to view an ACL as an attribute. Taking the dynamic aspects of access control into consideration, it becomes more appropriate to view an ACL as an object. Thus, we view "static" ACLs as more fundamental than extended types, but "dynamic" ACLs as extended types. This stratification technique, based on the distinction between the static and dynamic characteristics of an object, has been used in other areas of layered system design, as in the work of D. Reed on processes [ref. D. Reed].

Given this approach, the initial description of access control will not characterize an ACL as an extended type object (ETO). Later, after introducing type extension, we will take as second look at ACLs and discover that a change in attitude coupled with minor design modifications allows us to treat ACLs as extended type objects.

The (Static) Access Control List Layer

This section shows how an ACL might be implemented as an attribute of a segment object. To as great an extent as possible, the programs responsible for providing access control for segments will make use of objects which have already been defined. We shall refer to the programs which allocate and search the access control lists for segments and other objects as the access control list (ACL) layer. The ACL layer associates a list of access control attributes with each segment object. A necessary prerequisite for access by a principal to a segment object is that access attributes of the principal be compared with the access control list of the object. For example, the unique identifier of the principal may be compared to a list of principal identifiers associated with the segment. Alternatively, both the principal and the segment may have associated security compartments, which must relate in a particular way (such as set containment) in order that the principal be granted access. In any case, the function of the ACL layer is to provide the following mapping:

(object_UID, operation, principal_UID) --> boolean

which yields "true" only if the accessor (principal UID) is allowed, according to the ACL, to perform "operation" on "object_UID". This mapping is called the "search ACL" operation of the ACL layer.

We propose that the access control lists be implemented in terms of segment objects; that is, the ACL layer will depend on the segment layer. (1) To aid the following description, we introduce some new terms. An ACL-segment is a segment which is used by the ACL layer to contain access control lists. A P-segment stands for "protected" segment, and refers to a segment which has an associated access control list (implemented in an ACL-segment). The ACL layer must maintain bindings between a segment and its associated ACL-segment. Any segment except an ACL-segment may have an access control list. ACL-segments, as well as some other objects, have a degenerate form of access control list to be described later.

The ACL layer must, in effect, search the ACL-segment corresponding to a P-segment on each reference to the P-segment. To provide for efficient operation, part of the ACL layer is implemented in hardware. The level-1 B-frame is extended to contain one more field per entry, which is to contain a bit-encoding of access modes allowed the principal. Then each reference to the segment object will, prior to fetching the level-2 B-home, compare the access mode bits with the type of operation being attempted. The access mode field is initialized the first time a principal references a particular P-segment. If the access mode field is uninitialized, the hardware assists the ACL layer by causing a processor fault. The ACL layer takes the

---

(1) Although this section is devoted to the implementation of access control lists, the design presented here could be used as a basis for providing general property lists for segments or other objects.

UID of the referenced P-segment as an argument and performs the following steps.

1. It finds the UID of the corresponding ACL-segment in its table.

2. It initiates the ACL-segment, if necessary, and searches it.

3. If access is not allowed, it signals an access violation.

4. Otherwise, it sets the access mode field in the level-1 B-home entry to contain the proper mode bits for the referencing principal. (1)

Since access control lists will generally be rather short, the ACL layer may choose to represent the ACLs for several P-segments in the same ACL-segment. The ACL layer can carry out a policy regarding grouping of access control lists into ACL-segments: P-segments whose access control lists are actually contained in the same ACL-segment are all administered by the same <u>office</u>. (2)

One difficulty with the ACL layer implementation, as described so far, has to do with the mapping from P-segments to their corresponding ACL-segments. It is clear that providing this mapping function is the responsibility of the ACL layer; yet

---

(1) These same operations apply if the ACL is a degenerate ACL -- a simple data structure which designates only one principal and a set of modes.

(2) The office object is due to Rotenberg [ref. Rotenberg], and is mentioned in RFC 73.

providing such a map seems to be "re-inventing the wheel", since another layer (the map layer) already has an elaborate mechanism which maps UIDs into attributes. On the other hand, if the map layer were to "know about" any of the data structures of the ACL layer, this would be a violation of layering.

We solve this problem by appealing to a principle, which we shall call the "piggyback principle", which seems fundamental to constructing efficient layered systems. The principle states that there is no violation of layering if a lower layer maintains an uninterpreted data repository for a higher layer. For example, the lower layer can provide a

```
name  -------->  attributes
```

mapping for the higher layer, if the attributes are uninterpreted. Essentially, the only operations which the higher layer is allowed to perform are

```
fetch_attributes (name), and

replace_attributes (name, attributes).
```

The only cause for an error condition is a name unknown to the lower layer. The correct operation of the lower layer does not depend on the correct operation, or even the existence of, the higher layer.

We exploit the piggyback principle in this case by letting the map layer provide the mapping from P-segments to ACL-segments. The map layer will maintain, as uninterpreted data in each level-2 B-home, the UID of the ACL-segment (if any) corresponding to each segment. The map layer provides the operations

get_ACL (segment_UID), and

set_ACL (segment_UID, ACL_UID)

which the ACL layer can invoke. Performing either of these operations would cause the triple (segment_UID, level-2 B-home, ACL_segment_UID) to be placed in the map layer associative memory.

The relationship of the ACL layer to the other layers which

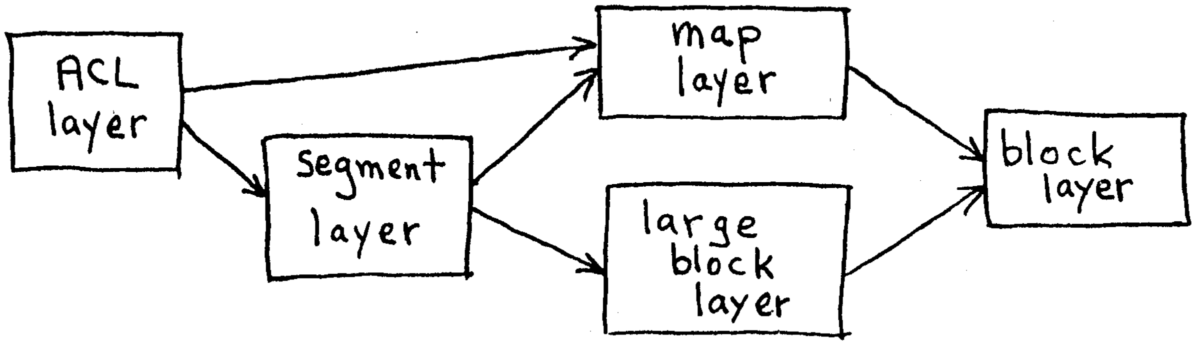

Figure VII

have already been described is shown in Figure VII.

A Layer to Support Dynamic Type Extension

This last section shows how the layers which have been described so far can support a type extension facility, in which the extended type objects (ETOs) are protected by access control lists. In generalizing from one protected object -- the P-segment -- to many, we observe that objects are partitioned into disjoint classes, or types. Extended type manager subsystems effectively define a type as a collection of operations [ref. Jones & Wulf IRIA paper]. All objects, and principals (which are also objects), have UIDs. In our implementation, the type of an object is represented by the UID of the principal which is the extended type manager of the object. The ACL of an ETO is represented by a UID, as is the representation, or REP. The REP of an ETO may simply be another object, or an object which designates a set of other objects and serves as a catalog for them.

We model a call on an ETM as

```
call ETM (operation, object_UID, consumer_UID)
```

in which a consumer, as defined in RFC 73, is merely any principal that wishes to reference the particular object. We assume that the consumer UID is unforgeable. Before actually referencing the representation of the extended object, any ETM would perform the following three functions.

1. It would check the type by the mapping: object_UID --> type.

An ETM should not be able to perform any operation on an object of incorrect type.

2. It would check the ACL by the mapping: object_UID --> ACL. The ETM should not be able to request an ACL search for an object of incorrect type. Also, the ETM should not proceed unless the consumer has the right to do the specified operation.

3. It should obtain the representation of the extended object by the mapping:

   object_UID --> REP.

   The ETM should not be able to get the name of the REP if the extended object is of incorrect type. (Of course, even if the ETM had the name of the REP of an ETO of a different type, it could not access the REP.)

In order to perform the mapping from extended object UIDs to type, ACL, and rep information, we specify a layer called the ETO layer which performs these three mapping functions for ETMs. We elaborate briefly on each of these three functions. First, there is type-checking. Each ETM must be able to determine if an object given it is one of its own. This information could be supplied by a function which maps object UIDs into types, i. e.

(object_UID) --> type.

Another function which still suffices but which provides less information is (object_UID, consumer_UID) --> boolean,

yielding "TRUE" only if the type of the object is the same as the unforgeable consumer UID. We shall choose the first alternative, which allows any principal to determine the type of an object. The second function is ACL searching. We specify that the ETO layer provide the mapping:

(object_UID, operation, consumer_UID, supplier_UID) --> boolean.

The "supplier" of an object is merely that principal which "supplies" it; i.e. the ETM. The supplier UID is assumed unforgeable. If the supplier UID is correct, the ETO layer will pass this request on to the ACL layer as a "search ACL" request. In the case of the third function -- obtaining the rep -- we choose the mapping.

(object_UID, supplier_UID) --> REP

which prevents an unauthorized ETM from getting the name of the representation. These three functions provided by the ETO layer are called "get_type", "search_ACL", and "get_rep" respectively. Each of these functions depends on a function of the same name provided by some lower layer.

Rather than duplicating mechanism, however, we shall rely on the piggyback principle once again and let the map layer provide these three mappings for the ETO layer. The map can be augmented to contain entries which describe ETOs. An entry which describes an ETO has four entries, as shown in Figure VIII.

```
 ---------------
 I_____UID______I
 I_____type_____I
 I_____ACL______I
 I_____REP______I
```

Figure VIII

These ETO entries are essentially the same as level-2 B-homes (which we also augment to contain a type field) except that the rep field is the UID of some other object, instead of a list of B-homes. (1) Any reference to an ETO entry in the map would also update the associative memory of the map layer.

To tie some of these ideas together, we present an example of a user referencing an extended type object. Suppose that Smith wishes to perform an enqueue operation on a particular object of type message queue, namely "MQ_26". The principal "Smith" would invoke the message queue manager (MQ_mgr) as shown.

```
call MQ_mgr (enqueue, MQ_26, message, Smith)
```

We assume that MQ_mgr, the ETM for message queues, has already created the rep of MQ_26, which is the segment SEG_14. The ACL of MQ_26 is represented by the segment SEG_32. Note that "Smith", "MQ_mgr", "MQ_26", "SEG_14", and "SEG_32" are for the convenience of explanation only, and that in fact all of these

---

(1) As an optimization, ETO entries need not be real leaf nodes of the B-tree. Rather, they could be stored in the B-tree nodes immediately above the leaf nodes, which should improve utilization of secondary storage and reduce search time. In fact, this same strategy can be used for segments as well, except that the "REP" field would designate a level-2 B-home.

names are UIDs. The map entries for the objects in this example

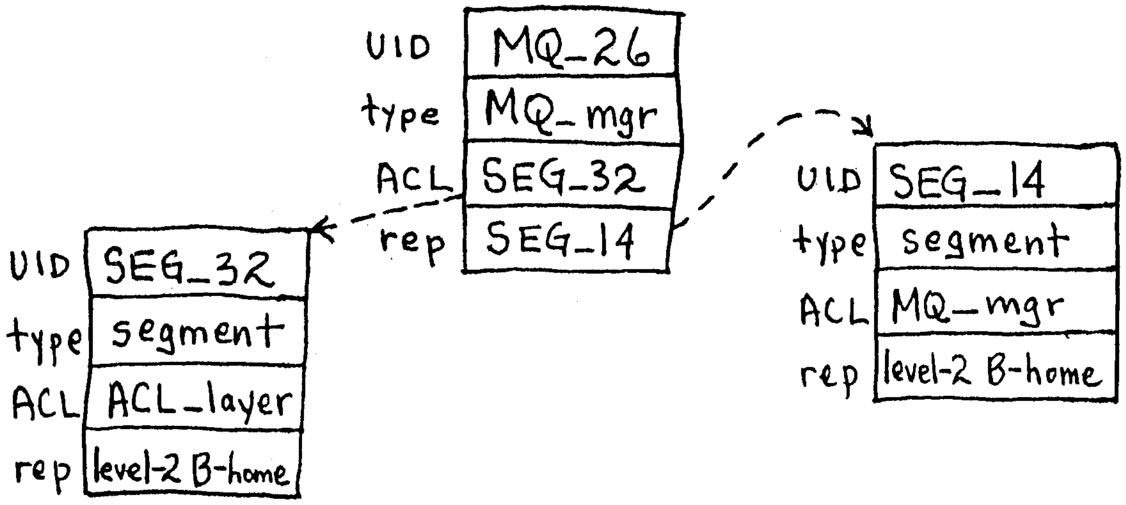

Figure IX

are shown in Figure IX. The ETM MQ_mgr, having been invoked as shown above, first checks the type of MQ_26:

call ETO_layer (get_type, MQ_26, type),

where "type" is an output argument returned by the ETO layer. In the case of the "get_type" operation, the ETO layer merely makes a "get_type" call with the same arguments to the map layer. The map layer finds the type field associated with the object MQ_26 and returns it. It is the responsibility of the MQ_mgr to abort the operation if MQ_26 is not the correct type.

Next, the MQ_mgr checks to see if Smith has "enqueue" access to MQ_26:

call ETO_layer (search_ACL, MQ_26, MQ_mgr, Smith, enqueue, boolean).

The name of the ETM is included an an unforgeable input parameter, since only the correct type manager can request that the ACL be searched. The boolean argument is an output argument which is true only if Smith does have the proper access. The ETO layer checks to make sure that MQ_mgr is the ETM for MQ_26, and if so, passes the request to search the ACL on to the ACL layer. The ACL layer calls the "get_ACL" entry of the map layer to obtain SEG_32, the UID of the ACL. It then searches the ACL and returns the boolean value to the ETO layer, which returns it to the ETM.

Next, the MQ_mgr obtains the representation of MQ_26 :

```
call ETO_layer (get_rep, MQ_26, MQ_mgr, rep).
```

As before, the name of the ETM is an unforgeable input parameter, and "rep" is an output parameter which is to contain the UID of the REP of MQ_26. Again the ETO layer checks to see that MQ_mgr is the ETM for MQ_26, and then calls the "get_rep" entry of the map layer. The map layer finds the rep field associated with MQ_26 and returns it. (1)

Once the MQ_mgr has the name of the rep, i.e. SEG_14, it will reference SEG_14 to effect the "enqueue" operation. While referencing the rep of any of its own objects, an ETM assumes the role of a consumer. Since it is a property of this type extension facility that type managers can be cascaded, and since

---

(1) The map layer actually has a little knowledge of types, since it distinguishes objects of type segment from other objects. It will not return the rep of a segment object.

the rep is a segment in this case, we could imagine the MQ_mgr calling an ETM which manages segment objects. That ETM, in turn, would perform the "get_type", "search_ACL", and "get_rep" operations. Although there is no ETM for segment objects, the ACL_layer, in effect, presents the same interface as a "segment manager" would since all operations on segments are mediated by the ACL layer. The "get_type" and "get_rep" functions for segments are carried out by the segment layer, and the "search_ACL" function is carried out jointly by the ACL and segment layers.

This example also illustrates the two varieties of access control lists. The first variety, or standard ACL, has already been described. It is implemented in terms of a segment object. The second variety of ACL is called a degenerate ACL, since it always contains just one principal. In this example, SEG_14 has a degenerate ACL which contains the principal "MQ_mgr", and SEG_32 has a degenerate ACL containing the principal "ACL_layer". The reason for having only one principal is evident: if an object is part of the rep of some ETO, then only the corresponding ETM should have access to it. Note that the only objects which do not have degenerate ACLs are objects which are not part of the rep of some other object. (1)

---

(1) ACLs are a special case and are described in the next section.

Viewing (Dynamic) ACLs as Extended Type Objects

Having sketched how type extension is provided, we now describe how ACLs may be considered to be ETOs. (1) Our model for administrative control of access includes special subsystems, called offices [ref. Rotenberg], which implement the administrative policy. It is assumed that every object in the system is subordinate to some office, in the sense that the policy embodied in the office ultimately controls which principals may access the object, and in what manner they may access it. The office is designed to allow a selected subset of principals to have some influence over office policy. (2) For example, in a system with non-discretionary controls, the set of principals which can influence office policy is only one principal: the system security administrator.

If we look upon (non-degenerate) ACLs as actual objects, then the natural candidate to be the ETM for an ACL is the office. Since in a computer utility there would typically be more than one office, there would be more than one type of ACL, given that the UID of an ETM indicates the type of an object. However, one reason why ACLs are not genuine ETOs is that there

---

(1) The degenerate ACLs described in the previous section are not intended to be changed, and therefore will not be viewed as extended type objects.

(2) In order to avoid certain anomalies, such as the situation in which all principals authorized to influence office policy suddenly vanish, manual overrides must exist. The administrator who overrides office policy may do so only in an overt, auditable manner; e. g. he may do so only at the system administrator's console in the computer room.

is only one type of ACL object. The representation of an ACL is not accessible to any office, but rather to the ACL layer only.

If some user wishes to inspect or modify the ACL of an object, he must call the office which serves as the ETM for that ACL. To find the right office, it must be possible to obtain the office of an object, given its UID. We assume that the UID of the office of an object is contained in the ACL of the object, and is available to any principal which calls the "get_office" entry of the ACL layer. (The office UID of an object can never be changed.) The user would then call the office:

```
call office (object_UID, operation, area, consumer_UID).
```

The operation would be to inspect the ACL, or to modify it in some way. If the operation is to inspect the ACL, the ACL data is returned in "area". The consumer UID is an unforgeable input argument. In order to validate requests of its callers, an office may interrogate its own data bases as well as inspect the ACL itself.

Offices require two special entries to the ACL layer to perform their role. The first one is

```
call ACL_layer (copy, object_UID, office_UID, area)
```

in which the office UID is unforgeable. The ACL layer gets the ACL of "object_UID", and if "office_UID" is the office in that ACL, it will return a copy of the ACL in "area" (probably in an "unpacked", easy-to-manipulate form).

The second special entry to the ACL layer is

```
call ACL_layer (update, object_UID, office_UID, area)
```

in which "area" is now an input argument. In this case, if the office is the right one, the ACL layer will validate the data in "area", convert it to its internal form, and store it in the ACL, using the "set_ACL" entry of the segment layer. These two entries to the ACL layer allow offices to carry out any user requests to inspect or modify ACLs. Although any principal can invoke the ACL layer, the ACL layer makes sure that only the correct office can perform an operation on an ACL by comparing the UID of the requesting office with the office UID in the ACL. In addition, the ACL layer will not perform a "copy" or "update" operation on a degenerate ACL, since degenerate ACLs are not to be changed.

The division of responsibility between offices and the ACL layer corresponds to the separation of policy and mechanism. The office is the only principal allowed to (directly) inspect or provide updates for the ACL, while the ACL layer is responsible for the format of an ACL and for searching it.

Summary

The primary goal of this report is to show that it is feasible to structure the supervisor of a sophisticated, extensible system without sacrificing economy of mechanism. Since the description of the supervisor which has been given in this report is superficial, this goal is only partially achieved at present.

Two techniques described here for attaining this goal are (1) allowing a lower layer to maintain uninterpreted data for a higher layer (the piggyback principle), and (2) applying the same procedures to "almost identical" data structures, as in the block and large block layers. The author is investigating other generally-applicable techniques.

The advantages of structuring an operating system according to a layered, object-oriented discipline include (1) a progressively nicer set of programming environments, (2) the likelihood that the implementation of any given layer will be relatively straightforward, (3) the ability to change the implementation of an abstraction without affecting other programs which use it, and (4) rendering the system more amenable to correctness proofs. Any system which must serve a general programming community in the real marketplace cannot be justified, however, unless it is efficient enough to be competitive with existent alternative designs. The results presented here should provide some insight into constructing systems which are layered and object-oriented as well as efficient.