MINIMIZING THE NAMING FACILITIES REQUIRING PROTECTION IN A COMPUTING UTILITY

Richard Glenn Bratt

September 1975

The research reported here was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641 which was monitored by ISTAO under contract No. F19628-74-C-0193.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis supervisor, Michael D. Schroeder, for his helpful suggestions and guidance throughout the conception and execution of this thesis.

Thanks are also due many other members of the Computer Systems Research group at M.I.T.'s Project MAC for their helpful comments and suggestions. In particular, I would like to extend my thanks to Doug Wells and David Reed for their help in isolating two programming bugs in the initial implementation of the design presented in this thesis.

I would also like to take this opportunity to thank my girlfriend, Claire, for her kind help and gentle understanding during the past months.

This research was performed in the Computer Systems Research Division of Project MAC, an M.I.T. Interdepartmental Laboratory. It was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641 which was monitored by ISTAO under contract No. F19628-74-C-0193.

MINIMIZING THE NAMING FACILITIES REQUIRING PROTECTION IN A COMPUTING UTILITY*

by

Richard Glenn Bratt

ABSTRACT

This thesis examines the various mechanisms for naming the information objects stored in a general-purpose computing utility, and isolates a basic set of naming facilities that must be protected to assure complete control over user interaction and that allow desired interactions among users to occur in a natural way. Minimizing the protected naming facilities consistent with the functional objective of controlled, but natural, user interaction contributes to defining a security kernel for a general-purpose computing utility. The security kernel is that complex of programs that must be correct if control on user interaction is to be assured.

The Multics system is used as a test case, and its segment naming mechanisms are redesigned to reduce the part that must be protected as part of the supervisor. To show that this smaller protected naming facility can still support the complete functionality of Multics, a test implementation of the design is performed. The new design is shown to have a significant impact on the size and complexity of the Multics supervisor.

*This report is based upon a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, on July 7, 1975 in partial fulfillment of the requirements for the degree of Master of Science.

TABLE OF CONTENTS

List of Figures

Figure Page

Introduction

1.1 Brief Statement of the Problem and Result

This thesis investigates the class of computing utility mechanisms that deal with naming information objects within a computing utility. Our goal is to understand the various functions played by name spaces in contemporary computing utilities and to decide which of these functions must be protected to assure complete control over user interaction. The Multics system, which is a sophisticated computing utility, will be used to test the validity of our conclusions. (1) We will find that Multics protects several mechanisms that we claim need not be protected to assure control over user interaction. To substantiate our claim we will present a redesign of Multics that allows these mechanisms to be unprotected without sacrificing the ability to control user interaction. The resulting reduction in the amount of code that must be protected to assure control over user interaction contributes to defining a security kernel for Multics.

(1) The Multics system was developed as a prototype computing utility by Honeywell Information Systems, Inc., and M.I.T.'s Project MAC. A complete bibliography of the Multics system may be found in [M2].

1.2 Related Work

The Multics system [C1, C2, M2, O1, S3] is an example of a sophisticated state-of-the-art computing utility. As part of a general investigation into how one goes about the task of certifying the security of large systems, the Computer Systems Research Division of Project MAC at M.I.T. is attempting to produce a certifiably secure version of the Multics system, by redesigning Multics to minimize the collection of programs that must be correct to assure complete control over user interactions. As a result, this collection of programs, the Multics security kernel, has been steadily decreasing in size and complexity. A recent masters thesis [J1] describes how a Multics security kernel that does not include a dynamic linking mechanism was developed. This thesis reports the results of another effort to reduce the size of the Multics security kernel.

1.3 Background

A computing utility is any computer system, or network of interconnected computing systems, that provide general computing services to a community of users. Among the most important services provided by computing utilities are facilities that allow users to share, store, retrieve, and process information. To facilitate the manipulation and sharing of stored information, computing utilities must support a multitude of name spaces. These name spaces, which maintain a correspondence between a collection of names and the information they denote, provide organization of the collections of information processed in the system.

We find many name spaces at all levels of a computing utility. The base computers on which a computing utility runs implicitly employ a name space that maps a set of integer names (actually a set of representations of integers) called addresses into a set of words of computer memory. Similarly, direct access mass storage devices such as magnetic disks and drums define a name space that maps physical storage addresses into records of bits. At a higher level, most computer utilities support a name space that allows its users to denote files of information by character string names such as "John's_file". Detailed analysis of most systems reveals many other examples of name spaces.

We have stated that a computing utility provides information processing services to a community of users. Since we have not placed any restrictions upon the composition of this user community, we must assume that these users harbor ill will toward each other or toward the computing utility itself. This ill will can manifest itself in any of three ways. A malicious user might attempt to use, modify, or prevent others from using or modifying information in the computing utility. Even in a computing utility shared by a non-malicious user community, one user might accidently compromise another user's information or computation.

Any general computing utility must prevent such undesirable interactions between its users. To this end it must secure its users against unauthorized use, modification, or denial of use of the information they process in the computing utility. This requires that the computing utility implement an authorization mechanism that allows those user-information interactions that are to be permitted to be specified. The information supplied to the system through this authorization mechanism must then be used by an access control mechanism that intercepts all user-information interactions and verifies that they are authorized.

The presence of access authorization and control mechanisms in a computing utility does not prima facie secure its users from harmful, uncontrolled interactions with other users of the computing utility. It must be established that these protection mechanisms do indeed perform their intended task without error. It further must be established that these information protection mechanisms cannot be subverted, damaged, or circumvented. Only then may users of the computing utility process sensitive, irreplaceable, or timely information with reasonable freedom from fear for its security.

We identify that subset of the mechanisms of a computing utility which must be correct in order to guarantee the security of the information contained in the computing utility as its security kernel.

Clearly the task of establishing the correctness of the security kernel of a computing utility must increase monotonically with its size and complexity. For this reason it would be advantageous to know which computing utility mechanisms need be included in the security kernel for intrinsic reasons. A mechanism has an intrinsic need to be included in the security kernel of a computing system if and only if it can be used by one computation to influence another computation. The access authorization and control mechanisms of a computing utility are the two most obvious examples of mechanisms that must be included in a security kernel. If a computing utility supports a shared name space for identifying stored information, then this mechanism, by virtue of its commonality, also allows one computation to influence another and hence must be considered part of the security kernel of the computing utility.

Mechanisms that have no intrinsic need to be protected often are included in the security kernel of a system. Common reasons for incorporating a mechanism in the security kernel of a computing utility when it has no intrinsic need to be protected include the desire to protect the mechanism from damage, the desire to minimize cross domain calls, and the need to protect the mechanism because some security kernel mechanism happens to depend upon its correct operation. The motivation behind including a mechanism in the security kernel of a computing utility when it has no security-related need to be protected must be carefully analyzed, as the inclusion of the mechanism in the security kernel contributes to the complexity of the security kernel. Removing the mechanism from the security kernel would have the advantage of lessening the task of establishing the correctness of the security kernel. This thesis will evaluate the need for each of the major name spaces supported by a typical computing utility to be included in its security kernel. We will use the knowledge thus accumulated to simplify the Multics security kernel.

1.4 Plan of thesis

In Chapter II we present a model of a computing utility. This model pays particular attention to those mechanisms that are involved in naming information stored in a computing utility. We begin by defining a very simple information storage and protection model. Through successive enhancement of this model we arrive at a model that we feel represents the essence of name space management in a contemporary computing utility. As we add each new name space to our model, we consider its basic raison d'être, the advantages and disadvantages it provides over the previous model, and most importantly its impact upon which name spaces in the model must be protected as a part of the security kernel.

Chapter III begins our case study of name space management in Multics. We identify the major name spaces maintained by Multics that deal with naming stored information and establish a correspondence between these name spaces and the name spaces of our model. Having established this correspondence, we attempt to verify that only the naming functions identified in our model as security sensitive are implemented by the Multics security kernel. This investigation reveals that the Multics reference name space, a name space used in resolving inter-procedure references, is implemented in the Multics security kernel although it has no intrinsic need to be protected. (1) The reasons behind this flaw in the modularity of the Multics system are investigated.

In Chapter IV we develop a design that removes reference name management from the security kernel of the Multics system. In so doing, we also remove several functions related to the management of the Multics global naming hierarchy from the Multics security kernel. The most notable of these are that function which allows the security kernel to name segments by

(1) The research reported in this thesis is based upon the M.I.T. Multics system of December 1974, Multics system 24.2.

hierarchy pathnames and that function which allows multiple paths in the Multics storage system hierarchy to designate the same object. In the course of removing these functions from the security kernel, our design drastically changes the Multics security kernel interface. Finally, we discuss the impact of this design upon the security kernel.

Chapter V discusses the implications of our security kernel design upon code running outside of the Multics security kernel. We discuss the principles involved in designing a reference name manager which runs outside of the Multics security kernel. In the course of this presentation we uncover an important consideration in moving any module out of the Multics security kernel. Specifically, Multics security kernel procedures are guaranteed to run to completion once invoked. This allows them to make assumptions that would be invalid were they to be executed in the interuptable environment outside of the security kernel. Following this discussion, we show how the functions of pathname resolution, and storage system link processing may be implemented outside of the Multics security kernel. Finally, we discuss the need for simulating the old security kernel interface.

In Chapter VI we discuss the results of a test implementation of the security kernel we have designed. This test implementation allowed us to measure of the impact of our design upon the complexity and performance of the Multics system. We report this data along with a description of our test implementation.

We have included nine appendices in this thesis. Appendix A details the structure of the data base for the Multics 24.2 address space manager and reference name manager. Appendix B shows the impact of our design upon the structure and content of this data base. Appendix C summarizes the new address space manager interface proposed in this thesis. In appendix D we present an example of the use of this new interface. Appendix E summarizes the impact of this thesis upon the size of the Multics security kernel. In appendix F we report the details and results of our performance comparison between Multics system 24.2 and our test system. Appendix G summarizes the effect of our thesis upon the complexity of the Multics security kernel interface. Appendix H presents the programs of our redesigned address space manager for the reader's perusal. Appendix I discusses several functions supported by the Multics system 24.2 address space manager that, for the sake of simplicity, were not considered in the body of the thesis.

Chapter II

Name Space Management in a Computing Utility

In this chapter we will develop a model of a computing utility. Our emphasis will be upon the roles played by name spaces in contemporary computing utilities. This model will be developed by adding successive layers to a central model of information storage and protection. After we add each successive mechanism or name space to this model, we will present a graphic representation of the current state of the model. Each node in these illustrations will represent a class of names. The name space binding one group of names to another group of objects or names will be represented by an undirected line. If a name space must be protected to control user interaction, then the line representing it will be constructed from the symbol "+". If the name space need not be protected it will be represented by a line composed of the symbol ".".

2.1 Basic Information Storage and Protection Model

Some basic notion of information storage and protection must be at the heart of any computing utility model. In our model the basic vessel of information storage is a segment. In theory, we do not restrict the amount of information a segment may contain. In practice, the amount of information a segment may hold will be bounded by a combination of hardware and software limitations.

Segments will also serve as our basic unit of information protection. We require that any information protection must apply uniformly to all information stored within a segment. We will choose an access control list (ACL) based information protection scheme for our model. The basic motivation behind this choice is that Multics, our test case system, uses an access control list protection scheme.

We assume that an access control list is associated with every segment. This access control list encodes the authority of each principal in the computing utility to use or modify the contents of the associated segment. (1) We will further assume that the computing utility supports the necessary principal authentication and access authorization mechanisms for maintaining the contents of access control lists. We require that at some point in referencing any segment, its associated access control list be used to mediate that reference.

(1) We assume that the reader is familiar with such computer science concepts as access, capabilities, domains, processes, and principals [S4, F1].

2.2 Global Machine Oriented Names

We will name a segment and its access control list by a name that is unique within the system. This name, which we will call a unique identifier (UID), will be compact, fixed length, and of high information density. The unique identifier naming a segment and its access control list will be assigned when the segment is created and may never be changed. Once assigned, a unique identifier will be valid for all time. If we allowed a unique identifier to be reused after the segment it names is destroyed, then that identifier would not uniquely identify a segment. It would be difficult, if not impossible, for a process to distinguish between different segments, existing at mutually exclusive points in time, named by the same unique identifier. (1)

It should be noted that we have purposely excluded the possibility of having more than one unique identifier bound to the same object. The reason for this is the need to determine if two segments are identical. If we guarantee that no two unique identifiers are bound to the same object, then we can decide if two segments are identical by comparing their unique identifiers. Lacking this guarantee, it is not clear how a process could decide if two segments were the same segment. (2)

(1) A discussion of the need for computing systems to support unique identifier name spaces may be found in Fabry [F1].

(2) By equal we mean the lisp concept of eq [M4].

Due to their compact size, unique identifiers are well suited to efficient implementation and manipulation by computing hardware. We will assume, for the moment, that access control will operate during the translation of unique identifier to object. Certainly this requires that the name spaces that associate unique identifiers with objects and their associated access control lists be protected. Otherwise a process could circumvent the access control mechanisms of the system by causing the unique identifier associated with any segment to name an arbitrary access control list or equivalently, causing the unique identifier associated with any access control list to name an arbitrary segment. It is therefore necessary that the security kernel exercise complete control over the unique identifier to access control list and unique identifier to segment name spaces. Since the security kernel must force these two name spaces to correspond, we will treat them as a single entity. Figure 2-1 illustrates this protected binding mapping unique identifiers into segments and their access control lists.

<UID> +++ <SEG/ACL>

Figure 2-1: Global Machine-Oriented Names

2.3 Global User Oriented Names

From the point of view of a human user, the unique identifier name space which we have defined for naming segments has four major inherent disadvantages. The first disadvantage is that humans are poor at dealing with high information density names. Second, since unique identifiers must be assigned by the system and not the user, they can have no mnemonic significance. Third, the binding or meaning of a unique identifier cannot be changed. The final disadvantage in the usage of unique identifiers by humans is that it is often convenient to allow multiple names in a name space to denote the same object. In our model we have precluded the possibility of having two unique identifiers name the same segment.

For these reasons, any viable computing utility must support a user-oriented name space. Ideally this name space should bind arbitrary length, user-supplied character string names to unique identifiers. In practice, some upper bound is often placed upon the size of user-supplied names. In any reasonable computing utility this restriction must not force users to use difficult-to-remember non-mnemonic names. To promote and encourage information sharing, this name space should be sharable by all processes in the computing utility. If this were not the case, then one user who wished to share a segment with another user would have to communicate the unique identifier of that segment to the other user. A shared user-oriented name space eases this communication problem by allowing users to identify segments in interpersonal communication by human-oriented names.

A well known weakness of such a simple, unstructured, global name space, which results from the need for a name space to define a function, is that two users may not name different segments by the same name. If One user names a segment "square_root_program", then no other user may use this name for another segment. Perhaps the most severe manifestation of this problem is that a user may not choose a name for a segment without knowledge of every name in the global name space.

Another consequence of the global scope of the name space we are defining is that it provides a path of user interaction. One user might intentionally modify a name to unique identifier binding that another user was depending upon. This constitutes an uncontrolled malicious user interaction since it allows one process to cause another process to reference the wrong segment. This in turn may cause an unsuspecting process to fail or compromise the integrity or security of sensitive information to which it has access. It is therefore apparent that the ability to change a global user-oriented name space must be regulated by the security kernel.

One simple authorization scheme a computing utility could adopt for its global user-oriented name space is to allow only the principal who created a name binding to modify that binding. Unfortunately, even such a primitive authorization mechanism is an unwieldy extension to the unstructured name space we have defined. Such an extension would require that every name binding in the name space have an associated principal name used to authorize modifications of that name binding. If the name space were structured into meaningful collections of name bindings, then a more natural authorization scheme based on controlling a process' ability to modify any of a related collection of name bindings could be employed.

Hierarchical name spaces, such as the user-oriented name spaces found in the Multics [B1, O1] and UNIX [R2] time-sharing systems, provide a powerful and natural solution to both the naming conflict and authorization problems outlined above. Since most name spaces found in contemporary computer systems, such as the ubiquitous "two-level" file system [M3], may be described as degenerate fixed-depth hierarchies, our model will support a hierarchical global user-oriented name space.

Hierarchical name spaces provide their users with a powerful organizational mechanism. This mechanism encourages logically related name bindings to be collected in a single directory or directory sub-tree of the hierarchical name space. For instance, each user could place name bindings he creates in distinct sub-trees of the hierarchy. Naming conflicts within a given directory are easily avoided by locally restructuring the hierarchical name space so that the conflicting name bindings occur in different directories. The directory structure of a hierarchical name space can also serve as the basis for a simple, flexible mechanism for controlling the modification of the name bindings in the hierarchical name space. The ability to use and/or change the name bindings in a directory can be specified by an access control list on that directory. Authorization control may also be delegated by allowing the access control lists of a directory to specify which principal may modify the access control lists of its sub-directories. Figure 2-2 extends our model to include both human-oriented and machine-oriented global name spaces.

```
      USER ORIENTED           MACHINE ORIENTED
          NAMES                     NAMES

        <PATHNAME> +++++++++++++ <UID> ++++ <SEG/ACL>
```

Figure 2-2: Global User-Oriented Names

2.4 Local Machine Oriented Names

At this point our model provides two very powerful mechanisms for naming information. One mechanism allows any segment in a computing utility to be denoted by a compact, fixed-length, unique identifier. The other naming mechanism allows segments to be named by arbitrary length character string names indicating the position of a segment in a naming hierarchy. In common to both of these mechanisms is the fact that their scope is global; they are shared by all users of the computing utility.

An obvious implication of the scope of a unique identifier is that it must be capable of representing as many distinct segments as the computing utility could create throughout its entire life. Because the set of segments existing at any one time will be a small subset of all segments that have ever existed or will ever exist, our unique identifier name space will be sparsely populated. For large systems with long lifetimes, this unique identifier name space will also be quite large. Economics demand that such large, sparse mappings be stored in a compact form requiring more sophisticated accessing methods than indexing by unique identifier value. This need for sophisticated retrieval methods in conjunction with the large potential size of the unique identifier to segment mapping tables suggests that this name space is difficult to implement efficiently. As a result, contemporary computing hardware provides a name space for addressing segments that is much smaller and denser than the global unique identifier name space. The increased efficiency of representation and mapping of this name space is achieved by restricting the scope of the machine-oriented segment identifiers.

The local machine-oriented name space in our model is patterned after the Multics _segment_ _number_ name space. Like unique identifiers, segment numbers are compact, fixed-length, machine-oriented names. Unlike unique identifiers, relatively few segment numbers are supported (1) and segment numbers are locally dense so that simple, efficient hardware translation techniques can be used. Since segments will be identified to the base level of the computing utility by segment number, we will call a segment number name space an address space.

There are many possible choices for the scope of segment numbers. A cooperating collection of processes could share a common segment number address space. Segment numbers could be private to a process, shared by all domains in that process. Conversely, the scope of a segment number could be a domain. It is even possible to imagine a system in which the scope of a segment number is temporally restricted. The choice of which of these or other possible schemes for limiting the scope of segment numbers is appropriate for a given computing utility depends upon both the hardware on which it must run and the desired patterns of interaction within the computing utility. The larger we allow the scope of a name space to be, the greater the cost of translating names in that name space. Conversely, the smaller we make the scope of a name space, the fewer the naming needs it can satisfy.

If we desire inter-domain communication to be efficient, then it would be inappropriate to restrict the scope of segment numbers to a domain. Were this done, segments could

(1) Multics supports a local, machine-oriented name space of about four thousand segment numbers.

only be named in inter-domain communication by unique identifier or, worse still, pathname. Since these names are not directly usable by the base level hardware of the computing utility, they would have to be mapped by the receiving domain into its segment number address space before the segment named could be referenced. By similar reasoning, if inter-process communication occurs with high frequency in a particular computing utility then that computing utility might choose to share a segment number address space among a group of cooperating processes.

The choice of the scope of segment numbers represents an engineering trade-off. We must limit the scope of segment numbers so that they may be efficiently implemented in hardware. Additionally, the smaller the scope of a segment number the less its need to be protected. If an address space is local to a protection domain, then it may be freely manipulated by that domain without compromising security. In opposition to the efficiency considerations that weigh in favor of reducing the scope of segment numbers is the desire to make the scope of a segment number as large as possible so as to make communication between different computer systems, processes, domains, and moments in time as efficient as possible. The desired characteristics and resources available to each computing utility must be carefully evaluated to determine the largest group of interacting objects that can share an address space without making the address space unacceptably large.

Routine communication between the security kernel domain and other protection domains in a computing utility should probably, for performance and modular programming reasons, be performed by using segment numbers to denote segments. This requires that the ability to manipulate the segment number name space we have just defined be controlled by the security kernel. This need for the security kernel to control the manipulation of an address space would not arise if address spaces did not span protection domains. The reader should take note of the fact that since segment numbers do not have global scope, our global user-oriented name space cannot be implemented by binding names to segment numbers. Figure 2-3 extends our model to include the protected binding of segment numbers to segments and their access control lists. We also include a protected binding between segment numbers and unique identifiers. This binding allows the identity of a segment named by a segment number to be established.

```
                     USER ORIENTED    MACHINE ORIENTED
                         NAMES              NAMES

  PER-SYSTEM         <PATHNAME> ++++++++++ <UID> +++ <SEG/ACL>
                                              +     +
                                              +    +
  PER-ADDRESS SPACE                        <SEGNO>
```

Figure 2-3: Local Machine-Oriented Names

2.5 Local Descriptors

Economics require that we refine the segment number to access control list and segment translations depicted by our model. These translations must be performed upon every reference to a segment. It is thus essential that they be efficiently implemented. In light of current computing technology, these translations must be performed in hardware if we desire our computing utility to be economically feasible.

Contemporary computing hardware supports neither the ability to address arbitrary amounts of storage nor the ability to perform the necessary access control list search upon every reference to a segment. To solve these problems one frequently finds two high-speed, hardware look-aside memories aiding the processors that implement a computing utility. One associative memory maps a segment number and domain identifier into a hardware interpretable representation of the domain's access to the segment specified by that segment number. We will call the entries in this associative memory _protection descriptors_ (PDS). The other associative memory maps a segment number into an _addressing descriptor_ (ADS) that allows the hardware to address the representation of a segment.

The processors we have described look up the address of a segment in their addressing descriptor associative memory and validate their authority to reference the segment with respect to the appropriate protection descriptor found in their protection descriptor associative memory. When one of these descriptors is not found in its associative memory, a hardware fault will be recognized. At this point software may intervene and take the appropriate steps to load the necessary descriptors and restart the faulted program.

Clearly the security kernel must control the manipulation of the protection descriptor and addressing descriptor name spaces. This is necessary since there exists a one-to-one correspondence between addressing descriptors and protection descriptors which must be maintained to preserve the integrity of the system's access control mechanisms. Figure 2-4 refines our previous model by supplanting the protected segment number to segment and access control list mapping by the four protected mappings described above.

```
             USER ORIENTED    MACHINE ORIENTED
                  NAMES              NAMES

PER-SYSTEM     <PATHNAME> ++++++ <UID> +++++++ <SEG/ACL>
                                   +            +     +
                                   +            +     +
PER-ADDRESS SPACE               <SEGNO> + <ADS> +     +
                                   +                  +
                                   +                  +
PER-DOMAIN                       <PDS> ++++++++++++++++
```

Figure 2-4: Local Descriptors

2.6 Local User Oriented Names

We have seen that efficiency considerations require our model to support a limited-scope, machine-oriented name space. It is only natural to consider whether there would be any advantages in our model also supporting a user-oriented name space of limited scope. The answer is, quite emphatically, yes.

Like the segment number name space we have defined, a user-oriented name space of local scope would be easier and faster to search than its global counterpart. But more important, it would provide a private name space that could be manipulated arbitrarily without worrying about interactions with processes outside of the scope of the name space. This latter ability is necessary in providing modular programming facilities.

It is clear that a program should not code into itself the unique identifier or even the pathname of another program, such as a square root program, that it wishes to call. This premature binding between modules would require that the first program be changed and recompiled if a new and better square root program was added to the computing utility. The caller of a square root program does not, in general, wish to be bound to a particular square root program. If the choice of which routine a procedure is to call can be delayed until the call is made, then we gain much flexibility.

We call a name that one program uses to refer to another program a <u>reference name</u> [O1] if its meaning is only defined in relation to a local name space. Such a local user-oriented name space is called a reference name space. One way to implement a space of reference names is to maintain a list of reference name to segment associations [O1]. Another mechanism for realizing a reference name space, found in many contemporary computer systems [J1, I1], involves searching an ordered list of specified directories, called search rules, to resolve inter-program references. Reference names provide a very useful mechanism for combining separately conceived subsystems and testing new subsystems all of whose components have yet to be written by allowing reference name to segment binding to be defered until the components of a subsystem are combined for execution.

In our model, each domain will have a private reference name space. This minimizes the problem of naming conflicts and allows each protection domain to operate without regard to the reference names used in other domains. A further advantage of per-domain reference names is that they need not be explicitly protected or controlled by the security kernel. Since reference names are private to a protection domain, each domain may freely manipulate its own reference name space. All that is required is that the reference names of each protection domain be stored in a segment accessible to only that protection domain. If reference names spanned protection domains, it would be necessary for a security kernel mechanism to control the manipulation of reference names to prevent one domain from exerting uncontrolled influence over another domain through the manipulation of reference names. Figure 2-5 shows the relationship of the unprotected reference name space to the other name spaces described so far.

```
             USER ORIENTED    MACHINE ORIENTED
                  NAMES              NAMES

PER-SYSTEM     <PATHNAME> ++++++ <UID> +++++++ <SEG/ACL>
                                   +            +     +
                                   +            +     +
PER-ADDRESS SPACE               <SEGNO> + <ADS> +     +
                                 .   +                +
                                 .   +                +
PER-DOMAIN     <REFERENCE NAME> ..   ++ <PDS> +++++++++
```

Figure 2-5: Local User-Oriented Names

2.7 Summary

In this chapter we have investigated the basic roles played by name spaces in a typical computing utility. Of the eight name spaces we have described, only the per-domain reference name space may be excluded from the security kernel without jeopardizing the ability of the computing utility to control user interactions. The critical difference between the reference name space, which can be uncontrolled, and the other seven name spaces we have considered, which must be controlled, is that the reference name space is not common to multiple protection environments. Since it cannot be used by one protection domain to exert influence over another protection domain, it need not be implemented in the security kernel.

Chapter III

A Model of the Multics System

Before approaching the specific problem of defining a security kernel for the Multics system that does not support unnecessary name space management mechanisms, we will present a detailed model of the Multics system and show its correspondence with our general computing utility model. Our Multics model contains four components: a storage system model, an information protection model, an address space model, and a reference name model. These models will contain sufficient detail to allow the reader who is unfamiliar with the implementation of Multics to comprehend the important details of the design we will present.

3.1 Storage System Model

The Multics storage system (1) manages two distinctly different types of objects called segments and directories. These objects are organized into a single system-wide tree structure that is known as the storage system hierarchy. This hierarchy implements the system's human-oriented global name space. The internal nodes of this hierarchy are directory objects. Each directory object is itself composed of a named

(1) A more complete description of the Multics storage system than will be presented in this section may be found in Organick [O1] and Bensoussan [B1].

collection of entries, one for each immediately inferior segment or directory in the hierarchy and one for each link in the directory. Links are psuedo-objects in the hierarchy that allow an object to appear to reside at several distinct nodes in the hierarchy. To accomplish this, the directory entry of a link contains the pathname of another object or link in the hierarchy that is to be considered as the target object of the link. The directory entry of a segment or directory object contains many important attributes of the object. Among these attributes are: a system-wide unique identifier, a collection of human-readable names for the object that are unique within the directory, an access control list, and a file map for the object that allows the system to access the object.

Each directory in the Multics hierarchy is stored in a separate segment. Many advantages accrue from supporting a hierarchical name space whose directories are implemented in separate segments. These advantages are closely interrelated. First, since each directory contains only a small fraction of the total name bindings represented by the hierarchy, it may be searched much more quickly than a corresponding single segment implementation of the whole hierarchy. Finding a name in a hierarchically organized name space requires searching only those directories defined by the prefixes of the name. In general, this will represent a substantial savings in search time. Second, the component names in a directory may be viewed as uniform, unstructured names. Finally, the names in a directory can be relatively small and yet still be unique.

As we have mentioned, a practical computing utility cannot assume that all users will be benevolent with respect to their manipulation of a global, shared name space. We must assume that some user, through malice or accident, will attempt to delete or modify name bindings that other users are depending upon. If this global name space is to be useful, then users must be able to control or at least know who may change the name bindings that are of interest to them. Controlling who may read the name bindings in a particular directory of a shared name space is also desirable since the names in a directory might themselves constitute sensitive information.

Since segments are the basic unit of access control in Multics, it is only natural to control the manipulation of the names in a directory by the Multics segment access control mechanisms. This approach is quite attractive since it allows the name bindings in a name space to be protected without introducing any new, special purpose access control mechanisms. The access control list of a directory specifies which principals may read and write its representation. In this way, the normal access control and authorization mechanisms of Multics automatically provide a certain degree of control over the manipulation of names in its hierarchical name space. Multics actually provides finer access control on directories than is afforded by its hardware enforced access control mechanism by encapsulating directories and a set of system-supplied procedures which manipulate directories in a protected subsystem [S1]. The procedures in this protected subsystem, which must be a part of the security kernel, exercise control over the use and manipulation of the name bindings in a directory.

If we assume that the root directory of the hierarchy is its own parent, then every object in the Multics storage system has a unique parent directory. Furthermore, since the hierarchy has the structure of a tree and names of directory entries are unique within that directory, we can specify an arbitrary object in the hierarchy by an ordered list of entry names. Such a specification is called a pathname. The first component of a pathname names an entry within the root directory, and each additional name specifies an entry within the directory specified by the list of names that preceeded it. By convention we take the name of the root to be the null name, and we write the pathname a, b, ... q as >a>b>...>q.

A leaf node of the Multics hierarchy can be either an empty directory, a link, or a segment. Segment objects, which are implemented directly by the Multics hardware, are primitive objects in which programs and data are stored.

In our general computing utility model a directory entry consists of one name to unique identifier mapping stored in a directory of the user-oriented hierarchical name space. The issue of where to store the access control list and other attributes of a segment or directory, which was not addressed by our general model, was resolved in Multics by merging this information with the entries of its hierarchical name space. This scheme has three important consequences. First, because a directory entry contains the attributes of the segment it names, no two directory entries in the hierarchy are allowed to describe the same segment. (1) This requires that an entry contain all synonyms of the object it describes. In our general computing utility model this was not necessary since there was no penality associated with allowing multiple entries (single name to unique identifier mappings) to denote the same object.

Second, the unique identifier to segment name space of our general computing utility model exists in Multics only as a collection of individual mappings scattered throughout all directory segments in the hierarchy. This renders the task of locating a segment given its unique identifier prohibitively expensive. However, Multics does use unique identifiers to facilitate the determination of whether two objects denoted by different pathnames are in fact the same object.

(1) If this rule were not obeyed, then the system would be faced with the error-prone task of maintaining identical, but separate, copies of the attributes of a segment.

Third, because the access control list of an object is stored in the object's superior directory, it is not possible to have the access control list on that object arbitrate access to the object independent of the access control lists on the object's superior directories. To see that this is true all we need do is consider the following scenario of a process attempting to reference a segment. Assume that the access control list of the segment specifies that the process is authorized to reference the segment, but that the segment's directory entry resides in a directory to which the process has no access. The system is faced with a paradox. If it allows the process to reference the segment, then it must allow the process to use information in the segment's directory entry. But the process is not authorized to use information in the directory containing the entry. Thus, if the system permits the process to reference the segment, then it must violate the authorization specified in the access control list of the containing directory. Conversely, if the system does not permit the process to reference the segment, then it must violate the authorization specified in the access control list of the segment. This dilemma will be discussed in detail in the next chapter.

3.2 Information Protection Model

The active agent of computation in Multics is a process. A process may execute instructions in any of eight protection domains, numbered from 0 to 7. These domains have the property that a process' access rights to objects in the storage system while executing in domain n are a subset of its access rights while executing in domain n-1. Domains that are so constrained have been named rings [S2]. To identify the user on whose behalf a process is executing instructions, the system associates with each process an unforgeable principal name. This access control name is used to establish a process' rights to access directories and segments in the storage system hierarchy.

Associated with each segment and directory in the storage system hierarchy is an access control list which, in conjunction with the access control name and ring of execution of a process, completely determines the access rights of that process to the object. The access control list in the directory entry of an object encodes the access mode or rights each principal is to have to the associated object in a given protection ring. (1)

(1) In the current Multics implementation both a segment's access control list and its ring brackets must be considered to determine the access rights of a principal to the segment in a given ring. Since this level of detail is unimportant for our purposes, we will imagine that a segment's access control list alone is sufficient to determine access.

When a process attempts to reference a segment or directory, the system evaluates the process' access modes to the target object. Conceptually, this involves searching the access control list of the object. This information is used to validate the process' right to perform a given operation upon the segment or directory. In the case of evaluating access to segments, Multics relies upon the hardware associative memories described in our general model to make access validation efficient.

For segments the valid access modes are read, write, and execute. These access modes are enforced directly by the Multics hardware. The valid access modes for directories are status - the right to read the attributes of the entries in the directory; modify - the right to change the attributes of the entries in the directory; and append - the right to add new entries to the directory. Directory access modes are interpretively enforced by the Multics security kernel.

Links, which are not full fledged objects in the Multics hierarchy, are not given an access control list. Instead, access to read the contents of a link is granted to any process that has status permission to the link's containing directory.

The process of a normal user executes in protection ring four. This allows the process to access only those segments and directories to which it has non-null access in ring four or some higher numbered ring. In order to access a storage system object accessible to the process only in rings numbered lower than four, a user process must enter an appropriate lower ring. This may be done only by calling a procedure which is designated, by its access control list, as a gate into that ring. When such a gate procedure is called, the process enters the inner ring. By virtue of its having entered an inner ring, the access rights of the process may increase. When the process returns from the gate procedure, it reenters its previous ring of execution and relinquishes the access rights it gained on entry to the lower ring. To put teeth into this protection mechanism, the storage system manager will not allow a process to create a gate into a lower ring than the ring the process is currently executing in. This insures that only procedures authorized to run in an inner ring may create gates into that ring. (1)

The Multics system takes advantage of this ring protection mechanism to protect its security kernel programs and data bases from tampering by non-kernel procedures. This is accomplished by setting the access control lists of security kernel procedures and data bases to indicate that they may be

(1) More complete descriptions of the Multics protection mechanisms may be found in Saltzer [S3], Schroeder [S2], and Organick [O1].

accessed only by processes executing in protection ring zero. Entry points in the security kernel which are callable by non-kernel procedures are declared to be gates into ring zero.

3.3 Address Space Model

The Multics system associates an address space with each process [B1]. The function served by this address space is to provide a mapping from a small set of virtual addresses, called segment numbers, that can be directly translated by the Multics hardware, onto the much larger set of objects in the Multics hierarchy. This segment number address space corresponds to the local machine-oriented name space defined in our general computing utility model. In the Multics system every process has a potential address space of several thousand segment numbers.

The binding of a segment number to a storage system object, which incorporates a storage system object into an address space, is called initiation. The effect of initiating a storage system object is to make the representation of that object appear directly addressable by the hardware of the Multics machine. Since Multics relies upon addressing and protection descriptors, such as those described in our computing utility model, to implement hardware references to segments, only a fraction of the hardware segment number to segment mappings implied by a process' address space need exist at any given instance. As in our computing utility model, the Multics security kernel handles faults caused by attempting to use missing descriptors by reloading the missing addressing or protection descriptor and restarting the faulted process. The unbinding of a storage system object from a segment number, which removes the object from the process' address space, is called termination.

Our discussion may have lead the reader to the conclusion that a process may have several segment numbers bound to the same storage system object. Actually, this is not permitted by the address space manager. During the initiation of an object, the address space manager locates the directory entry of the object from which it fetches the system-wide unique identifier of the object. This identifier is looked up in a per-process table (1) that maps unique identifiers into segment numbers. If the unique identifier is found in this table, then the object is already in the address space of the process. This being the case, the initiation primitive returns an indication to this effect as well as the segment number that is bound to the object. This scheme has several advantages. First, it helps a process conserve its segment numbers - a very scarce resource. Second, it permits a process to test the identity of two objects in its address space by comparing the segment numbers assigned to

(1) See appendix A.

these objects. Finally, it simplifies the management of the Multics virtual memory.

3.4 Reference Name Space Model

We have asserted that local user-oriented name spaces in a computing utility need not be part of its security kernel. This claim not withstanding, the Multics supervisor implements a reference name space for every ring of every process. These name spaces provide a mechanism for mapping character string names into segment numbers and vice versa. In the current Multics implementation only segments may be assigned reference names. The security kernel itself does not use reference names for normal segments. It does however misuse its unique ability to assign reference names to the segments with which it implements directory objects. (1) Specifically, the Multics supervisor uses the reference name manager to associate the hierarchy pathnames of initiated directories with the segment number of the segment containing the representation of the directory. As we will see in the next chapter, this presents problems when directory objects are renamed. This problem will be discussed in great detail in the ensuing chapters.

The address space manager and reference name manager share a common data base in the current Multics implementation.

(1) In non-kernel domains directory objects are sealed and may not be accessed as segment objects.

This combined data base is called the Known Segment Table and is documented in appendix A. The reader who is unfamilar with the structure and contents of the KST is urged to review this material. Additional information on the Multics reference name manager may be found in Organick [O1] and Bensoussan [B1].

Chapter IV

Design

The Multics designers recognized the advantages of building a computing utility on top of a central security kernel. As a consequence, Multics is more fortunate than most existing computer systems as regards its securability. By construction most modules of the Multics system are not permitted to execute in protection ring zero. This bulk of code is thus prevented by the Multics protection mechanisms from tampering with those programs and data that are only accessible from protection ring zero. These protected programs constitute the Multics security kernel. Although the portion of the Multics supervisor that lies outside of the security kernel dwarfs the security kernel in comparison, the modules of the Multics security kernel are still quite numerous as well as complex. The object modules of the Multics security kernel presently represent approximately one hundred and fifty thousand machine instructions. These instructions implement in excess of two hundred user callable functions as well as a host of implicit system services such as demand paging.

We will present a redesign of the current Multics security kernel that will enhance its certifiability by reducing its size and number of external interfaces. As a side effect, we will also improve the modularity and coding of the area of the system we will investigate. Our design will eliminate the need for the Multics security kernel to support reference name management. This requires that we carefully redesign and remodularize ring zero so that it is independent of the reference name manager. This is necessary since a security kernel must not depend upon the correctness of procedures outside of the kernel. Before getting into the details of our design, we will investigate the reason behind ring zero's current dependence on the reference name manager.

4.1 Security Kernel Dependence on Reference Name Management

While there does not appear to be any intrinsic need for the Multics security kernel to support reference name management, its removal from ring zero is complicated by the fact that the Multics address space manager uses the facilities of the reference name manager to maintain an association between the pathnames of directories it has initiated in a process and the segment numbers of these directories. The address space manager uses these associations to avoid having to repeatedly resolve identical directory pathnames into segment numbers. Since the security kernel must not depend upon a mechanism outside the security kernel, it is necessary to decouple the address space manager from the reference name manager before the latter can be removed from ring zero.

The dependence of the address space manager upon the reference name manager manifests itself in the recursive procedure find_ which the address space manager uses to resolve directory pathnames into directory segment numbers. This resolution is necessary since the hardware base of the system only implements references to storage system objects by segment number. When find_ is invoked to determine the segment number for a directory, it calls the reference name manager to map the pathname it is given, interpreted as a reference name, into a segment number. If the pathname is a reference name known in ring zero of the process, then find_ returns the associated segment number as the segment number of the directory. (1) If the pathname is not a known reference name, then find_ splits the pathname into a pathname of the parent directory of the target directory and the directory entry name of the target directory. It then calls itself recursively to obtain a segment number for the parent directory. Using this segment number to reference the parent directory, find_ attempts to initiate the target directory. If it succeeds, it calls the reference name manager to bind the pathname of the target directory, as a reference name, to the segment number assigned to the target directory.

(1) As we will see later, this can cause problems since this segment number may no longer be bound to the directory specified by following the pathname find_ was given step by step through the directory hierarchy.

This thesis suggests a radical change in the ring zero address space manager. The essential result of this change is that find_, as described above, need no longer be called by ring zero. This allows both find_ and reference name management to be removed from ring zero.

4.2 Source of the Dependence

One of the basic goals of the Multics protection mechanism is that a process should be unable to detect the existence of a storage system object to which it has no access. (1) A second basic goal of the Multics protection mechanism is that the access control list of an object should be the sole specifier of access to the object. (2)

(1) We will consider that if a process has access to the parent of an object then it has sufficient access to determine the existence of the object. The reason for this will be discussed later.

(2) This goal was not originally embodied in the Multics design. Originally a process' access to an object was a function of three different access control lists. The first list was part of the directory entry of the object and corresponds to the access control list we now have. The second list was part of the object's parent and was common to all entries in the directory. The last list was a one per system master access control list. The result was a very complex access evaluation mechanism that allowed an unwary user to increase a principal's access rights to an object by removing that principal from one access control list when his intention was actually to deny the principal access to the object. The complexity of this mechanism so confused users that many of them did not attempt to use the system provided protection mechanism. With the current Multics design a user needs only review one access control list to determine who has access to a given segment.

These goals have made the determination of whether a process should be permitted to initiate an arbitrary directory quite difficult. This difficulty stems from the fact that the access control list of an object and its physical storage map reside in its parent. Since we wish the access control list of an object to exercise complete control over access to that object, we must permit a process to initiate all superiors of accessible segments independent of access to these superiors. But this violates our second goal.

Multics attempts to resolve the conflict outlined above by not permitting a process running outside of ring zero to initiate a directory. Since a process cannot read the access control list of a segment until its parent is known, the system still must permit processes, while executing in ring zero, to initiate directories that they may not have the right to know exist. By causing the initiation of these superior directories to occur in a single, indivisible ring zero call, the system could, in principle, prevent security leaks. This could be accomplished by terminating those intermediate directories that had to be initiated only to find that the process had no access to the terminal segment, before returning to the caller. Unfortunately, Multics system 24.2 does not do so. As a result, any process can determine the existence of any postulated directory by attempting to initiate any arbitrarily named descendent (which need not exist) of that directory and observing how many segment numbers were allocated by ring zero. This is possible because all rings share a common address space.

It would be relatively easy to correct the implementation flaw in the Multics address space manager pointed out above. However, the system would still have to be very careful to avoid compromising information. For example, suppose a process filled up its address space intentionally and then called ring zero to initiate >secret>x. If ring zero was not very careful, it might cause the process to die due to its inability to find an unused segment number to bind to >secret, if and only if >secret existed. This would allow the existence of >secret to be inferred by whether or not the process died.

The inability of a process to initiate directories in outer rings directly has led to many needlessly complex mechanisms for manipulating directories. In addition, it has forced us always to refer to directories by pathname in the security kernel interface. Not only is this inefficient, but it has led to ring zero's dependence upon find_. If we could initiate directories directly outside ring zero, then the ring zero interface could take a segment number instead of taking a pathname as a directory specifier. Since ring zero would no longer need to call find_, it could move out of ring zero, along with reference name management, without compromising the security of ring zero.

4.3 Removal of the Dependence

4.3.1 Overview of the Design

We propose allowing directories to be initiated by processes executing in all rings. As was noted earlier, the basic problem to be solved is that of deciding whether a process should be allowed to initiate a directory to which it has no explicit access. (1) There are essentially four schemes for making this decision. The first scheme involves recognizing that if the access control list of a directory is to completely express access to that directory, then we must make explicit the now "hidden" permission to initiate a directory if some descendent of the directory is accessible to the process. The obvious way to accomplish this is to invent a new directory access mode called "initiate". This mode would allow the named principal to initiate a directory and to use the information it contains that is relevent to accessing descendents of that directory. This makes the decision of whether or not a process should be allowed to initiate a directory quite simple. If the process has non-null access to the directory, then it may initiate it. Otherwise, it may not.

(1) The reader should note that we are ignoring, for the purposes of this thesis, the possibility of solving the problem outlined above by removing the attributes of a segment from the directory hierarchy. Removing the attributes of a segment from its parent directory, which may be the best long term solution, seems very attractive but requires a fairly extensive overhaul of the system. This thesis will investigate less drastic solutions to the directory initiation problem which do not disturb the structure of the Multics hierarchy.

This scheme does not meet the goal that the access control list of an object completely express which processes may access that object. While explicit initiate permission is probably a workable solution, and its simplicity is appealing, adoption of such a solution would produce a quite noticable change in the system's functionality. We choose to explore alternative solutions that maintain the current system's functionality.

A way to maintain the current functionality of Multics using explicit initiate permission is to couple the access control list on an object with the access control lists on all superior directories, so that when a process is given access to an object it is also given initiate access to all superior directories of that object. When a process subsequently is denied access to an object, the security kernel must remove any initiate permission that the process had to the superior directories of the object and that resulted solely from its having access to the object. Determining which initiate permissions should be removed is very difficult, potentially requiring that the entire directory hierarchy be examined.

A second way to decide whether a process may initiate a directory is to search the hierarchy subtree rooted at that directory. If the process has non-null access to any member of this subtree then the process should be allowed to initiate the directory in question. Naturally, this scheme is far too inefficient to consider seriously.

A third method of deciding whether a process may initiate a directory is to require non-null access to the directory. This scheme has the disadvantage, shared by the first scheme discussed, of preventing the access control list of a directory or segment from being the sole arbiter of access to that directory or segment. In order to initiate a segment, a process would need non-null access to the superiors of that segment.

We propose a fourth solution to the problem of initiating directories. Instead of worrying about whether or not a process has the right to initiate a directory, let us allow all processes to initiate any directory - whether or not it exists. The key to this scheme is preventing the process from detecting any difference between an initiated directory that does not exist and an initiated directory that exists but that the process has not proven its right to know exists. How this is to be done will be discussed later.

The ring zero address space manager interface resulting from this approach seems quite natural. Ring zero no longer concerns itself with pathnames. Instead, it accepts directory segment numbers for directory specifiers. To allow this scheme to bootstrap itself, we will define the segment number of the parent of the root to be zero. Initiation of segments and directories will be controlled by the procedure initiate_ that will accept a parameter specifing whether a segment or directory is to be initiated.

The rationale behind distinguishing directory and segment initiation is that a process usually has a preconceived idea about the type of the object it wishes to initiate. When reality does not support this preconceived idea, the process is usually in error. Forcing the process to make explicit the type of object it is expecting allows ring zero to immediately catch many such errors, preventing a careless process from bumbling along thinking all is well only to die when it attempts to access a directory as a segment or vice versa. Naturally, it would be a security violation for the kernel to report a type violation to a process that has no right to know whether the directory or segment named actually exists. If a segment or directory should be undetectable to a process, then the security kernel must treat it in a manner consistent with the type specified in the initiate call regardless of its actual type.

To complete our new ring zero address space manager interface we must define a new termination primitive. This primitive will accept two arguments. The first argument specifies the segment number to be terminated. The final argument is a status code. It should be noticed that this primitive may be called with either a segment or directory segment number. In the case of terminating a directory, one constraint is enforced. Since the system requires that a known segment's parent also be known, terminate_ will not terminate a directory with known inferiors.

4.3.2 Details of the Design

So far everything seems rosy. This scheme seems to remove many functions from ring zero and to simplify the ring zero interface in the bargain. Where is the hitch? Do we get all this for free? The answer is, of course, no. We have glossed over one important point. In order to decouple directory and segment initiation we must be able to successfully cloak the physical initiation of directories from a process' detection until it has established its right to know of the existence of the directory. As was pointed out earlier, this need for deception is intrinsic to the hierarchy structure and functionality of the Multics system. While this design makes the system's need to deceive the user more obvious, it is not responsible for the required deceit.

We will call a directory detectable if a process has established its right to know that the directory exists. Detectability may be established either by having non-null access to the directory, by having non-null access to its parent, or by establishing the detectability of an inferior of the directory.

The reason that non-null access on the parent of an object establishes its detectability is that either status, modify or append permission to a directory is sufficient to allow a process to detect if a postulated entry in that directory actually exists. It should be noted that the detectability of a directory is dependent on the process' history and the ring of execution.

A directory is detectable by a process in rings zero through the highest ring in which it has detectably initiated some member of the tree rooted at that directory. This highest detectable ring number of a directory is kept in its KSTE. (1) We will not attempt to reset this field should a once detectable directory subsequently become undetectable. Not attempting to reset the highest detectable ring field in the KSTE of an object when it becomes undetectable to the process makes sense since the system has already admitted the existence of the directory to the process. The process could have stored this information elsewhere, so it would be of little use to deny the existence of the directory. The record kept in the KST of the existence of the directory will naturally vanish when the directory is terminated or when the process is destroyed.

We must prevent a process from detecting any difference between an initiated directory that does not exist and an initiated existing, but undetectable, directory. If a process

(1) See appendices A and B.

could detect a difference in these two cases then it could establish the existence of <u>any</u> postulated path in the hierarchy. This would constitute a clear violation of security. To accomplish this means abandoning the current one-to-one mapping that exists between occupied segment numbers and initiated segments and directories. Although we will still only allow one segment number to be bound to a segment, we must allow multiple segment numbers for the same directory.

The reason for this dichotomy between segments and directories is simple. Since the access control list of a segment completely controls the right to initiate that segment there is no need to allow a process to initiate a segment to which it has no access. This allows us to hide the physical existence of a segment from a process that has no right to know of its existence by returning the ambiguous status code "noinfo" in response to an initiate request. This simple mechanism fails for directories since we must always allow a process to initiate an existing directory in case it has access to some inferior of that directory. This forces us to return more than one segment number for a directory in some cases in order to prevent the process from detecting the existence of physically initiated but logically undetectable directories.

There are two characteristics of Multics that necessitate our abandonment of the current one-to-one mapping between directory segment numbers and directories. First, directories can have multiple entry names. If initiate_ returned the same segment number for two different entry names within a given directory, then the process would know that these names both named the same directory. This coincidence of names would establish the existance of the directory (if the directory did not exist, then how could it have two names?). To prevent the coincidence of multiple names on a directory from revealing the existence of the directory, we must return a new segment number if a process reinitiates a directory that is still undetectable with a new name. In fact, we will even return a new segment number if it tries to initiate an undetectable directory with the same name twice. If we returned the same segment number, then in order for directories that do not physically exist to appear the same to the user ring, ring zero would have to remember the name of every phoney directory. This is a needless complication of ring zero.

The second characteristic of Multics that forces our abandonment of the one-to-one mapping between directory segment numbers and directories is that the segment numbers of a process are a finite resource shared among all protection rings of that process. As we have commented earlier, the finite size of the Multics shared segment number address space allows one ring to detect the number of segment numbers being used by all other rings. This makes it necessary to assign a new segment number whenever an attempt is made to initiate an undetectable directory. This segment number must not be shared with another ring so long as the directory remains undetectable. The need for assigning private, per-ring segment numbers to undetectable directories may be seen in the argument that follows.

Assume the system returned the same segment number when asked to initiate a directory in two different rings. Assume also that the directory is undetectable in the upper of the two rings. What is the system to do when asked to unbind the segment number from the directory by the upper ring? It cannot unbind the segment number and return it to the list of free segment numbers since a lower ring is using the segment number. Unfortunately the ring that requested the system to terminate the segment number can detect whether or not the system actually returned the segment number to the free list so the system cannot just pretend to honor the termination request. If the segment number is not freed then the ring can deduce that some other ring has the directory initiated. By an argument similar to the one given in the previous paragraph the ring can conclude, from the coincidence of two rings having the directory initiated, that the directory actually exists. Since segment numbers are a scarce resource, the system cannot take the easy out of not allowing undetectable directories to be terminated. As a result, initiate_ must assign a new segment number whenever it initiates an undetectable directory.

The reader should note that we have ignored, up to now, the problem of preventing a process from distinguishing between a non-existent directory and an existent but undetectable directory through observation and analysis of second order effects such as the time required to initiate or terminate a directory. It is hard to predict in advance of installation in the standard system what sort of second order effects might be observed. The plan is to investigate this problem following actual installation. Timing differences can be easily hidden by inserting extra code in the shorter path. Other differences also probably are disguisable.

This scheme will merrily allow a process to initiate vast trees of directories that do not exist. These directories will be indistinguishable from real undetectable directories. The potential multiplicity of segment numbers for directories implies that if we compare two directory segment numbers and find them to be not equal, then we cannot conclude that the objects to which they are bound are not one and the same. Since processes running outside ring zero cannot currently obtain segment numbers for directories, no user code can be affected by this new restriction. To allow processes to quickly determine if two segment numbers are bound to the same object, the system should support a function for mapping a segment number into the unique identifier of the object to which it is bound. Naturally, this function must return an error if the object is not detectable to the process. The system must also assure that if a process attempts to reference through any directory pointer in an outer ring, it will get the same access violation whether or not the segment number it referenced corresponded to a real or phoney directory.

Figure 4-1 summarizes the actions performed by initiate_ when mapping a directory into a process' address space. The reader should note that a target object within a phoney directory is considered a priori undetectable and a non-existent target object is considered detectable by a process if the process has non-null access to the containing directory. The abbreviation "hdr" used in figure 4-1 stands for the contents of a KSTE's highest detectable ring field. We have omitted the case where the target is a link as this case will be discussed later.

.target is detectable in ring of caller	.target exists in hierarchy	.target already has a segment number	return values		internal state
			status code	segment number	hdr
0	-	-	"no_info"	new	0
1	0	-	"noentry"	none	-
1	1	0	0	new	ring of caller
1	1	1	"known"	old	max(hdr,ring of caller)

Figure 4-1: Action of Initiate_ for Directories

Two possible objections we can see to this scheme are that it can potentially waste segment numbers and it requires inspecting the parent's access control list. A close examination of figure 4-1 indicates that there are only two ways to assign multiple segment numbers to a directory. The first way is to reinitiate an undetectable directory. The second is to initiate a phoney directory. Neither of these operations should occur in normal operation. They could, however, arise in an attempt to use a misspelled pathname. To control this problem, the outer ring variant of find_ could terminate those directories that might be phoney if the terminal segment could not be initiated. This would prevent a habitual misspeller from cluttering his address space. It seems that with this addition a process would be obliged to go out of its way in order to clutter its address space. If that is what it wants fine. Even if a process wastes all its segment numbers, it can recover by terminating no longer needed segment numbers.

The apparent inefficency of inspecting the access control list of the parent of a directory during its initiation is not serious since it is normally not required. Only when a process has null access to an object and has not previously established detectability for that object is it necessary to inspect the access control list of the parent. (1)

(1) In fact, the frequency with which a process initiates a directory to which it has has no access is low enough in Multics that our test implementation does not check to see if a process has previously established detectability for a directory to avoid

In Multics system 24.2 the address space manager and the reference name manager share a data base. (1) The address space manager takes advantage of its ability to access the reference name manager's data base by scanning the per ring, per segment number, list of reference names kept by the reference name manager to determine which rings of a process are still using a particular segment number. This information is used to prevent one ring from terminating a segment number that is still in use by another ring. (2) Only if all rings that initiated the object have terminated it, can the segment number be unbound from the object. Thus, we have the concept of initiating an object in a particular ring rather than the concept of initiating an object globally in all rings of a process. This scheme is desirable since all rings share the address space of segment numbers.

inspecting the access control list of the parent of the directory. If the process has null access to a directory, then we always check the process' access to the parent of the directory.

(1) See appendix A.

(2) Since the address space manager uses the presence of reference names in a given ring for a segment number to detect that the ring is still using the segment number, the current initiation primitive must call the reference name manager to give a segment a reference name in the appropriate ring each time the segment is initiated. The current initiate interface supplies the address space manager with a reference for this purpose. A more complete description of the relationship between the address space manager and reference names in system 24.2 may be found in Organick [01].

Since reference names will no longer be kept in the KST, some new mechanism must be invented to supply information about which rings of a process are still using a given segment number. This is easily accomplished by adding an eight bit field, called rings, to each KSTE. If the i th bit(0 origined) in this field is on then the corresponding ring has the segment number initiated. This allows ring zero to detect when a segment number may be physically terminated, thereby preventing one ring from terminating a segment or directory that is being used by another ring. (1)

Our termination primitive marks the segment number it is given as free in its caller's ring of execution. If the segment number is initiated in no other rings and its inferior count is zero, then the segment number is unbound from the object and its KSTE is placed on a list of free KSTEs. It should be carefully noted that the termination primitive terminates a single segment number; it only removes an object from the process' address space if the last segment number for the object is terminated. The reader should notice that because initiate_ always assigns a private segment number when a directory is undetectably initiated, terminate_ need not worry about revealing the existence of an undetectable directory.

(1) Appendix B summarizes the content of the known segment table as we have redefined it.

4.4 Removal of Pathname Processing

Ring zero's ability to resolve a pathname into a segment number has been severely impaired by our design. This ability, which was embodied in the ring zero procedure find_, depended upon ring zero's ability to call the reference name manager. Specifically, find_ depended on the reference name manager to maintain an association between pathnames of objects and the segment number bound to the object. Fortunately, this association was only used to make find_ more efficient. As a result, we could redefine find_ in such a manner that it would still operate correctly but would not take advantage of such an association between pathnames and segment numbers.

To make find_ independent of the reference name manager, all we would need to do is redefine find_ to inspect the pathname it was given to see if it specified the root, i.e. ">". If it did, then find_ would initiate the root, and return its segment number. (1) Otherwise find_ would strip off the last component of the pathname and call itself recursively with the pathname of the parent of the target object to get its segment number. Given this segment number, find_ would call initiate to initiate the entry named by the component which was previously

(1) The system treats the root directory as a special case. The location of its physical object map as well as the rest of the information that would reside in its directory entry, if it had a parent, is embedded in the programs of the system. This guarantees that the root may always be initiated.

removed from the pathname. For example, if find_ were called with >a>b it would call itself recursively to get a segment number for >a. It would then call initiate to get a segment number for the object named b in the directory >a.

While the procedure we have described is correct, it appears to be quite inefficient. This inefficiency suggests that we should either give find_ a new associative memory or move it out of ring zero so that it can once again use the reference name manager. Since giving find_ a new associative memory would add code to ring zero which has no protection reason to be in the security kernel, this alternative is untenable. Our approach will therefore be to remove find_ from ring zero.

The actual removal of find_ from ring zero is, of itself, trivial. In the outer rings it can access the reference name manager directly once again. It can also access our new initiation primitive through a standard gate into ring zero. The problem is that numerous programs in ring zero depend upon find_ to map pathnames into segment numbers. Unfortunately, they cannot be allowed to call our new find_ in the outer ring. To do so would jeopardize the security of ring zero. To get ourselves out of this dilemma, we will have to remove almost all uses of pathnames from ring zero. This in itself represents a substantial simplification of ring zero. To accomplish this task we will consider the four major uses of pathnames in ring zero.

4.4.1 Parameters to Ring Zero

The first class of pathnames used in ring zero that we will consider consists of those pathnames that were passed into ring zero as an argument to a gate procedure. This class represents the major use of pathnames in ring zero. Fortunately, it is also the easiest class to remove from ring zero. Since find_ now resides in the outer ring, we will make the outer ring responsible for translating all pathnames that are currently passed into ring zero into segment numbers. We will then redefine all ring zero gates that accept pathnames as object specifiers to accept segment numbers as object specifiers instead.

4.4.2 Links

The second class of pathnames used in ring zero are the pathnames contained in links. Many ring zero programs, when they discover that the object they are to act upon is a link, are defined to act instead upon the target of the link. An example of a ring zero function that is defined to follow this rule is the segment initiation primitive.(1) We propose that primitives

(1) To prevent a process from causing ring zero, which is masked against interupts, from looping indefinately following a circular chain of links, each program that follows links keeps count of the number of links it traverses during each invocation. If this number exceeds a certain system-specified threshold, then the computation is aborted.

which are defined to follow links return a status code indicating that a link has been encountered as well as the contents of the link itself, upon discovering that their target is a link.

This scheme requires that links be readable in the outer rings which raises the question of what, if any, access control should be placed on reading links. The approach taken in Multics system 24.2 is to make links effectively readable by any process that has non-null access to the terminal target of the link. This scheme has an inherent security flaw and is therefore unacceptable. If some process can guess the pathname of an existing link to whose target the process has access, then it can prove the existence of the parent directories of that link by initiating the target object through the link. To eliminate this security flaw we could place access control lists on links, thereby explicitly naming those processes which may read the link. The complexity of such a mechanism seems unwarranted when weighed against its benefits. The only access control on the target object of the link that is guaranteed is specified by the access control list of that object. Any access control specified on a link may be avoided by referencing the target object directly and thus serves only to protect the contents of the link itself.

The reasons that access to links must be controlled is that the existence of a link implies the existence of its superior directories and suggests the existence of its target. We have chosen a simpler mechanism for controlling access to links which, although not as comprehensive as a mechanism that associates a private access control list with each link, meets both of the needs for protecting links. We consider a link to be part of its containing directory, readable only by processes having status permission on that directory. This scheme has the virtues of being simple, easy to implement, and plugging the information hole that uncontrolled access to links provides in system 24.2. While this scheme does make one class of currently legal uses of links illegal, this restriction does not seem too severe.

To illustrate the scheme we have proposed, we will outline the redesign of link processing by the ring zero initiation primitive. When initiate_ encounters a detectable link, it will return the link and a status code that informs the outer ring procedure that a link was encountered. (1) The outer ring procedure may then try the new path specified by the link. Since this is happening in an outer ring, we need no longer have a standard interpretation of links. Since link processing will be done in the user ring, the process may interpret links in any manner it chooses. Why not let links contain relative pathnames, offsets, or even arbitrary character strings? A link might even

(1) As we have mentioned earlier, if an undetectable link is encountered while attempting to initiate a directory, the system must treat that link as an undetectable, phoney directory.

specify a file residing in another computer system. The important point is that while the kernel may be the keeper of links, it does not interpret them. Naturally, the restriction on link depth, which was intended to keep ring zero from getting into trouble, vanishes.

4.4.3 Internally Generated Pathnames

In a few cases, ring zero generates and uses pathnames internally. These generated pathnames constitute the third general class of uses of pathnames in ring zero. We will further partition this class into those pathnames that are generated only during system initialization and those pathnames that are generated during normal system operation.

During the initialization of the Multics system, the need arises to initiate on the order of one hundred or fewer segments. The reason the system must initiate these segments is of little interest to our thesis. We observe that since system initialization is an infrequent operation (hopefully once a day or less) and the number of pathnames to be resolved is quite small, we need not feel remorse at proposing a very inefficient mechanism to resolve these pathnames. In fact, as the reader has undoubtedly guessed, we propose that these pathnames be resolved by calls to the inefficient version of find_ that we described earlier.

In the case of pathnames generated by ring zero during normal system operation, we cannot be quite so cavalier. Or can we? In fact, we can. A careful examination of ring zero reveals that ten is a reasonable upper bound on the number of generated pathnames that must be resolved in ring zero in the life of any given process.

In fact, these internally generated pathnames are so restricted that we have no need to even call our inefficient find_. Since they all are of tree depth at most three and all components of these pathnames except possibly the last component are constant for all time, we could expand the code of find_ in line in the programs that use these pathnames. For example, if a program needed to initiate >pdd>my, then it would first initiate the root. Then, given the segment number of the root, it would initiate pdd. Finally, given the segment number of pdd, it would initiate my.

4.4.4 Error Conditions

The last and perhaps most troublesome class of pathnames used in ring zero are pathnames that are used to report error conditions. There exist numerous instances in the system where a procedure detects an inconsistency or error condition associated with some segment or directory. For instance, the system may detect an unrecoverable error while reading the contents of a segment. Another example would be the detection that the doubly threaded list which chains the entries in a directory together is misthreaded. In error conditions such as these, the system writes a message into the system log explaining the problem. This message often contains a pathname that was generated from the virtual address of the segment or directory in which the error occured. While the exact algorithm for generating a pathname from a virtual address is of little interest to us, this algorithm did depend upon the reference name manager's ability to map a directory segment number into a pathname of the object it was bound to.

Since we have argued that ring zero must not call the outer ring name space manager, we must propose a new algorithm for mapping a segment number into a pathname. Many schemes are possible. However, since the error conditions we are talking about may be presumed to be quite rare, we will suggest a very simple, but inefficient, algorithm. This algorithm relies on the fact that any virtual address may be mapped, by the known segment table, into the virtual address of its directory entry. A name for the segment can be found in the directory entry. This name is the last component name in a valid pathname of the object. To get the other components of a pathname of the object, we recursively apply this technique to the virtual address of the directory entry which is, of course, within the parent directory.

4.5 Summary of the Design

This chapter has presented a design that allows directories to be initiated in all rings. As a consequence, the need for the Multics security kernel to maintain reference names has been eliminated. The key feature of this design is that the security kernel maintains, for each process, the illusion that any postulated directory exists unless the process has sufficient access to prove otherwise. This permits the security kernel to allow a process to initiate a directory to which it has no access without disclosing the existence of that directory. The address space manager interface presented in this design is summarized in appendix C. Appendix D contains an example of the use of this interface.

Chapter V

New Non-kernel Functions

As a result of our design, the interface to ring zero has been modified quite extensively. We have eliminated three major functions that were supported by the old ring zero: reference name management, pathname resolution, and storage system link indirection. If the non-kernel portion of the Multics supervisor is to use these services or provide them to the users of the system, then we must design modules capable of providing these services that run outside of ring zero. We have already explained, to a degree which we hope is sufficient to convince the reader, how the last function may be trivially performed by outer ring modules. In this chapter we will discuss the important issues involved in resolving pathnames in the outer ring and designing an outer ring reference name manager. In addition, we will address ourselves briefly to the problem faced by user programs that depend upon now obsolete ring zero interfaces.

5.1 Reference Name Manager Design

We have seen that the Multics reference name manager provides four primitive functions on name spaces. These functions provide a process with the ability to: bind a name to a segment number, unbind a name, determine the segment number that a name is bound to, and obtain a list of the names bound to a segment number. Actually, the Multics reference name manager provides a larger set of functions. However, the additional functions all can all be expressed in terms of the four primitives we have described.

It is not our intention to actually design a reference name manager. We trust that the reader will accept our assurance that it can be done and that it is in fact straightforward. We must, however, comment on one consideration that the design of an outer ring reference name manager must recognize. When the name space manager resided in ring zero it was operating in an environment in which it was guaranteed to run to completion once invoked. An outer ring name space manager is not afforded this luxury.

Executing in the outer ring environment, the reference name manager may be stopped at any instant. This of little consequence except when it is stopped by the Multics "quit" mechanism. In this case, the system suspends the process' current computation and then restarts the process. The process may then reinvoke the reference name manager and at a later time resume the suspended computation having potentially totally rearranged the reference name manager's data base.

Luckily the system provides a mechanism that allows a process to inhibit or "mask" quit signals. By masking quits on entrance to the reference name manager and unmasking quits upon exit the problem can be eliminated. Actually, it is highly unlikely that the entire computation performed by the reference name manager need be masked. We should design the reference name manager so that it has as small a "critical" section or sections as possible. In other words, we should try to isolate the code that might malfunction if it were not masked against quits. We can then mask and unmask quits only when we enter and exit a critical section.

Before leaving the topic of name space management, we should comment on one consequence of allowing processes to initiate directories directly. This ability allows a process to use the reference name manager to bind an arbitrary name to a directory. One immediately obvious use of this new facility is to replace the current special purpose mechanism that identifies a process' per ring working directory and search directories [O1]. All we need to do is bind the appropriate name, i.e. "working_dir" or "search_dir_n" to the correct directory segment number.

5.2 Pathname Resolution

We have commented that reference names are per ring. This prevents programs executing in one ring from causing programs executing in another ring to malfunction by tampering with shared reference names. As a result, ring four could bind the name "sqrt" to one procedure and ring one could bind the same name to an entirely different procedure. While this multiplicity of name spaces per process is desirable for protection and modular programming reasons, it partially defeats find_'s purpose in using the reference name manager to bind pathnames to segment numbers. Since each ring has a different name space, associating the pathname >a>b with segment number 401 in one ring will not help another ring resolve >a>b. The result is that many redundant pathname resolutions will occur and many name spaces will contain identical entries.

We suggest that find_ not use the reference name manager to associate pathnames with segment numbers. In fact, it was never correct for it to have done so. A name space just associates an arbitrary name with a segment number. However, pathnames are not just arbitrary names. Consider, for instance, what happens when we remove the name b from the directory >a>b and then add the name b to the directory >a>c. The result of this change in the environment is external to the reference name manager and yet it has invalidated a mapping the reference name manager was keeping. The pathname >a>b no longer refers to the object that is bound to segment number 401, but the reference name manager has no way of knowing this.

There are potential advantages to binding pathnames to directories once per process, as is done in Multics system 24.2. Consider the problem of installing a new version of a multi-component subsystem, such as the Multics PL/I compiler, while Multics is running. In Multics system 24.2 we could store the components of the compiler in a single directory. To install a new version of the compiler all we would need to do is build the new version in a brother directory of the current compiler. When the new compiler is ready for installation all that would be necessary is to exchange the names on the new and old compiler directories. Processes that had already started to use the compiler would remember the segment number of the old directory as the compiler directory and would continue to use the old compiler and satisfy new dynamic linkage faults to components of the compiler from the old directory. In this way a process always gets a consistent copy of the compiler. A process that had not yet used the compiler would initiate the directory containing the new compiler when it attempted to invoke the compiler. It would then remember this new directory as the compiler directory and satisfy all linkage faults for pieces of the compiler from this directory.

If a process does not "freeze" a directory sub-tree, as is done in system 24.2, when it initiates that directory, then it becomes very difficult to do on line installations of multi-component subsystems. A process could easily get half of an old multi-component subsystem and half of a new version of that subsystem when an online installation of the subsystem is done. On the other hand, a process often wants to use the actual hierarchy, not a "frozen" image of the hierarchy. Our design allows a process to choose between these two alternatives by supplyiing an appropriate version of find_ in the outer ring.

We suggest that the system supplied find_ opt for solving the "directory renaming problem" rather than the "online installation problem". The easiest and most attractive approach to solving the directory renaming problem is to not allow find_ to use a pathname, segment number associative memory. Instead, find_ will always recurse to the root when resolving a pathname. While this might seem unattractive for efficiency reasons, direct measurement of the impact of such a scheme upon system performance reveals that system throughput would only be degraded by a small fraction of a percent. In addition, our proposed address space manager will drastically reduce the number of pathname resolutions that occur within the system. This reduction in pathname resolutions should render the difference between find_'s having and not having a pathname associative memory almost immeasurable. This slight performance degradation seems a small price to pay for the elimination of the directory renaming problem outlined above.

5.3 Compatibility

The final topic we wish to discuss in this chapter is that of compatibility. A basic responsibility of any computing utility is to minimize the effect of internal changes upon its user community. If a major change must be made in the interfaces between user written programs and the system, or in the semantics of these interfaces, then the system must support both the new and old interfaces for a sufficiently long period of time to allow users to convert their programs to use the new interfaces. A suitable meaaure of this period of time would probably be measured in months or even years, not hours, days, or weeks.

We have made substantial changes to the ring zero interface and thus must address the compatibility issue. Fortunately, it is quite simple to preserve compatibility without keeping the old find_ and name and address space managers. This is possible for two reasons. First, we can simulate the old ring zero interface by interposing a ring four procedure between the caller of an obsolete ring zero interface and our new ring zero interface. Second, it is possible to interpose such simulation procedures between the user and the new ring zero interfaces without recoding or even recompiling any user programs.

Consider how we would simulate the old interface to initiate. The outer ring interposing procedure would call the outer ring reference name manager to map the pathname directory specifier of the old interface into the segment number required by the new interface. It would then call the new initiation primitive. If this returned a link, the outer ring interposing procedure would start over again.

This simulation procedure would be difficult to implement if it were not for the fact that Multics now has an interposing procedure on all calls to ring zero. This procedure is a ring four transfer vector that normally transfers the call to the appropriate ring zero gate. (1) This transfer vector can be modified so as to call an appropriate interposing interface simulation procedure for the interfaces we have changed.

(1) This transfer vector, which was discussed in a previous masters thesis by Janson [J1] has not yet been installed in the current Multics system.

Chapter VI

Implementation

We have coded a test implementation of the essential features of our design. This test implementation, which is based on Multics system 24.2, was undertaken for four major reasons. First, a working implementation of our ideas serves as an existence proof of the basic claim of our thesis. Second, a working implementation helps us demonstrate the practicality of our design. Third, the actual task of implementing our design helps insure that we have not neglected any important details in our design. Finally, a test implementation of our design helps us to quantify the impact of our design upon the system.

6.1 Plan

We have indicated that our new design requires an extensive overhaul of ring zero. The pervasiveness of the modifications necessary to ring zero is largely a result of the removal of pathnames from ring zero. While the removal of pathnames from ring zero is essential to our design, it is a time consuming, straightforward, and intellectually unrewarding task.

Instead of undertaking this drudgery, we have devised a scheme that allows the essential ideas of our design to be implemented while avoiding most of the uninteresting work. The implementation we will describe does not affect any code outside of ring zero, nor does it affect the syntax or semantics of the interface to ring zero. As a result of this feature, our test implementation provides the first step in an orderly transition from the current Multics system to the system we have described. The implementation we will describe could be immediately installed in the standard Multics system without substantially affecting users.

What we elected to do was to implement our new initiation, termination, and name space management primitives inside ring zero. We then reimplemented, inside ring zero, the old initiation, termination, and name space management primitives using our new primitives. This scheme allowed us to concentrate upon the key issues of our design without getting bogged down in the mechanics of converting thirty or more large complex programs from using pathnames to not using pathnames.

The strength of this approach is that the modules in ring zero may be slowly weaned away from using pathnames or now obsolete interfaces. Also, by supplying gates to our new primitives, users of Multics can start converting their programs to take advantage of the new ring zero interface. When ring zero has been completely converted, all we need do is throw away the code that implemented the old primitives in terms of the new primitives and move the reference name manager out of ring zero.

6.2 Impact on System Complexity

Reducing the complexity of a system certainly increases its certifiability [D1, D2, D3, L1, M1, P1]. In order to substantiate the hypothesis that our design results in a system that is more certifiable than Multics system 24.2, we will look at two measures of the complexity of the security kernels of the two systems. These measures are the difference in size of the old ring zero and our new ring zero and the difference in the number and complexity of gates into the old ring zero and our new ring zero.

Appendix E summarizes the size comparison data between the old ring zero and our new ring zero. As it reports, the address space manager was reduced in size by seventy-seven per cent. This correaponds to a two and a half per cent reduction in the size of ring zero. In fact, the address space manager that we designed was so small that we have presented it in appendix H for the reader to peruse. This sizeable reduction in the size of the address space manager is quite encouraging and substantiates our claim that we have produced a more certifiable ring zero. What is even more encouraging is that while this figure is in itself substantial, it only represents a partial implementation. Several modules in ring zero accept both pathnames and segment numbers as storage system object specifiers. In a complete implementation of our design many of these modules would only accept segment numbers. This would allow the code that handled the pathnames in these modules to be thrown out of ring zero, further decreasing its complexity.

The old ring zero supports about two hundred gates. Our design clearly removes the necessity of having gates into ring zero which call the reference name manager. It also removes a whole class of gates that allow an object to be specified by pathname. Many gates into the old ring zero came in pairs. One gate would specify the target object by segment number. The other gate would specify the target object by pathname. With the ability to initiate directories in the outer rings, this multiplicity of gates becomes unnecessary. As a result, only the gates that take a segment number as object specifier would be retained in the ring zero of a complete implementation of our design. When we add up the number of gates that a full implementation of our design would remove from the current ring zero interface, we find that we would remove about five per cent of the gates. In addition to reducing the number of gates into ring zero, we have significantly simplified the interface to over fifty of the gates that must remain in ring zero. (1) This reduction in interface complexity also lends credibility to our claim that we have made ring zero, and hence Multics, more certifiable.

(1) See appendix G.

6.3 Impact on System Performance

To help assess the impact of our design upon the performance of the Multics system, we developed a small benchmark that tests the speed and paging behavior of the most used system functions that our design affected. This benchmark was run on both Multics system 24.2 and our test implementation. The results of these runs indicated that the virtual cpu time to initiate and then terminate an object dropped from 11.002 milliseconds in the standard system to 10.226 milliseconds in our test system, a reduction of eight per cent. (1) This is especially gratifying since the test name space manager we implemented was not in the least optimized for running speed. In addition, our test implementation was unfairly penalized by having to converse with our benchmark through a simulation of the old interfaces.

We attribute this speed up to many factors; not the least of which is the fact that we greatly simplified the structure of the known segment table. We also make the somewhat immodest claim that our initiation, termination, and reference name management primitives were simply coded better than those in system 24.2. But this is not surprising; most things are done better the second time around. It should also be noted that the

(1) A description of our benchmark as well as a brief summary of the performance data can be found in appendix F.

smaller and less complex a module is, the easier it is to program that module efficiently and correctly. Unless a programmer can hold all of the relevent details and specifications of a program in his head at one time, it is very difficult to perform global optimizations or simplifications of the program.

Our working set performance data indicates that our system referenced two more pages running the benchmark than system 24.2. This did not come as much of a surprise. One of these extra page faults resulted from splitting the code of the reference name manager and address space manager apart and the other resulted from splitting apart their shared data base. We anticipate that when programs are converted to use the new interfaces directly the extra page fault that was caused by splitting the code apart will be compensated for. We expect that since our code is smaller in total, by eliminating the simulation code we will decrease the working set by a least a page. This will make up for the extra page fault caused by splitting the reference name manager and address space manager apart. The increase in working set due to splitting apart the known segment table cannot in itself be avoided. However, this increase in working set is only on the order of a half of a page and is independent of the combined size of the new data bases.

We have not really put much effort into the performance arguments above. We feel that the performance data which we have reported above is not, in fact, a good measure of the performance of a full implementation of our design. We claim that there is a hidden performance factor which will easily swamp out the performance effects we have been discussing. Fortunately, this hidden performance factor is in our favor. The effect to which we are alluding will not be seen immediately but will slowly assert itself. This effect has to do with the gradual conversion of major supervisor and user programs to use segment numbers as directory specifiers. Since pathname resolution is fairly expensive (even when find_ is given a pathname - segment number associative memory), the use of segment numbers as directory specifiers will save an average process a substantial amount of computation.

Chapter VII

Conclusion

We have argued that reference name management need not be supported by the security kernel of a computing utility. In particular, we have demonstrated a transformation on the Multics system that removes reference name management from its security kernel. Our design has further simplified the Multics security kernel by allowing directories to be initiated outside of ring zero, and removing the concept of a storage system link from ring zero. In the process, we have repaired an inherent security flaw in the current Multics design that allowed processes to detect the existence of objects in the storage system hierarchy to which they had no access. This flaw resulted from having insufficient access control on links and from ring zero's failure to terminate undetectable directories. Finally, we have provided a solution to the problem of clearing find_'s pathname associative memory when a directory is renamed.

We have used a technique in our redesign of the Multics system that we feel deserves special mention. This technique involves constructing a careful lie to maintain the security of a piece of data. In our case, we constructed a security kernel that lies about the existence of a directory until the caller proves its right to know of the existence of the directory. This lie, which was actually quite easy to maintain, prevents a process from detecting directories that should be undetectable by pretending that all possible pathnames correspond to an existing directory unless the process has sufficient access to the object specified by the pathname to prove otherwise.

We have implemented and tested the key points of our design. This implementation has shown that our design is both simpler and more efficient than Multics 24.2. More details of our design than were presented in the body of the thesis may be found in the appendices that follow. In particular, appendix H presents the actual programs of the address space manager designed in this thesis.

In conclusion, we would like to note three observations we made while designing a new address space manager for Multics. First, our address space manager, which is far simpler than the current Multics address space manager, also is more efficient than the current address space manager. The complexity of the current address space manager cost Multics both space and performance. (One is tempted to believe that, in general, complexity added to improve performance is frequently counterproductive.) Second, because Multics is an existing system, the functionality and use patterns of the Multics address space manager were thoroughly understood when we began our research. A large part of the simplification achieved is the direct result of insight extracted by observing the existing implementation of these mechanisms. Finally, we noticed an impressive threshold effect. As our design progressed it got simpler and simpler. At a certain point, when our design was simple enough so that all of the relevant details of the design could be considered simultaneously, our design underwent a further drastic simplification. This simplification was only discovered when the mechanism became simple enough and small enough to be kept in the head of one designer all at one time.

APPENDIX A

Structure of the Multics Known Segment Table

The main data base for the Multics system 24.2 ring zero address and reference name manager is the Known Segment Table. The KST is a per-process, ring zero segment. Logically it contains three items. First, it contains an array of KST Entries. KSTEs are indexed by segment number and contain all per-process information necessary for the proper care and feeding of the segment or directory associated with the indexing segment number. Second, it contains a hash coded mapping from the space of Unique IDentifiers onto the space of segment numbers, or equivalently the space of KSTEs. This mapping provides the means of locating the KSTE of an already initiated segment should it subsequently be initiated by a different name. Third, it contains a hash coded mapping from the space of names onto the space of segment numbers. This association is mainly of use to the dynamic linking mechanism. The current contents of a KSTE and their major usages are given in the following table.

KSTE Field	Use
forward pointer, backward pointer	These pointers are used to chain the KSTE onto a list of free KSTEs when it is not in use.
unique identifier	The unique identifier of the segment is used to validate UID hash searches and to properly identify the corresponding directory entry after an on-line salvage.
name pointer	This pointer chains together a list of the reference names associated with this segment or directory. Stored with each reference name is the number of the ring in which the name is known.
inferior count	The inferior count records the number of inferiors of a directory that are in the process' address space. This information is used to prevent a directory from being terminated while it has known sons.
parent segment number	This entry records the segment number of this segment's parent. It is used at segment fault time to help locate this segment's directory entry. It also is used to translate segment numbers into pathnames.
entry offset	This entry, which records the offset of this segment's directory entry within its parent, is used in conjuction with parent segment number to locate the segment's directory entry.
directory switch	This flag, which is set to indicate that the segment implements a directory object, is used to special case access setting for directories at segment fault time.

APPENDIX B

Structure of the Proposed Known Segment Table

Our redesigned KST has been simplified and contains only two components: a KSTE array, and a UID hash table. The contents of each KSTE and their major uses are summarized below.

KSTE field	Use
forward pointer, backward pointer	Used to thread KSTE onto free or hash class list as required.
unique identifier	Unchanged (a phoney directory will have a uid = 0).
inferior count	Unchanged.
entry pointer	A pointer to the directory entry for this segment.
directory switch	Unchanged.
rings	An eight bit field containing one bit per ring. Whenever ring i has this segment number initiated then bit i of this field is on.
highest detectable ring	A number that specifies the highest ring in which this process has established its right to know of the existence of this directory.

APPENDIX C

Proposed Address Space Manager Interface

The proposed ring zero address space manager interface is as follows.

initiate_ (dirsegno,ename,dirsw,link,segno,code)

dirsegno	segment number of the parent (input)
ename	entry name of target (input)
dirsw	directory switch (input)
link	link (output)
segno	segment number of target (output)
code	status code (output)

possible status code values:

error_table_$segknown --- segment already known to process
error_table_$invalidsegno --- parent is not a directory
error_table_$noinfo --- insufficient access to return any information
error_table_$nrmkst --- no more room in known segment table
error_table_$no_entry --- entry does not exist
error_table_$wrong_type --- entry is of the wrong type
error_table_$link --- entry is a link

terminate_(segno,code)

segno	segment number to be terminated(input)
code	see above

possible status code values:

error_table_$invalidsegno --- segment number is not bound to an object
error_table_$infcnt_non_zero --- can't terminate due to active inferiors
error_table_$known_in_other_rings --- can't terminate due to segment number being used in other rings

APPENDIX D

Example

To help clarify the ideas presented in this thesis, let us consider the following scenario in which a process tries to initiate the segment >a>b>c>d>e>f in ring four. We will assume that directory e and segment f do not exist and that the process has no access to a, b or d, and append permission to c in rings zero through four. We have presented below a representation of this path through the hierarchy along with the process' access rights to each object in ring four.

```
"root" <-- status
   |
   a    <-- null
   |
   b    <-- null
   |
   c    <-- append
   |
   d    <-- null
```

To simplify matters we will ignore the existence of the outer ring reference name manager and we will assume that we are operating in a virgin environment. What follows is how the outer ring find_ would proceed in this case.

step 0 call initiate_(0,"",1,link,segno_of_root,code)

The root directory will be initiated, its detectable field in the KSTE will be set to four, and a status code of zero will be returned. (all processes have status permission to the root directory)

step 1 call initiate_(segno_of_root,"a",1,link,segno_of_a,code)

The directory will be initiated, its detectable field in the KSTE will be set to four, and a status code of zero will be returned.

step 2 ll initiate_(segno_of_a,"b",1,link,segno_of_b,code)

The directory will be initiated , its detectable field in the KSTE will be set to zero, and the status code noinfo will be returned.

step 3 ll initiate_(segno_of_b,"c",1,link,segno_of_c,code)

The directory will be initiated, its detectable field in the KSTE will be set to four, and a zero status code will be returned. In addition this initiation establishes the process' right to know of the existence of superior directories at least in rings zero through four. This is reflected, in this case, by setting the detectable field in the KSTE of >a>b to four.

step 4 call initiate_(segno_of_c,"d",1,link,segno_of_d,code)

The directory d will be initiated, its detectable field in the KSTE will be set to four, and a zero status code will be returned.

step 5 call initiate_(segno_of_d,"e",1,link,segno_of_e,code)

The non existent directory e will be assigned a KSTE which will be marked as phoney and the status code noinfo will be returned.

step 6 call initiate_(segno_of_e,"f",0,link,segno_of_f,code)

No KSTE will be assigned and the status code noinfo will be returned.

step 7 call terminate_(segno_of_e,code)

The segment number assigned to e will be released on the grounds that e may not really exist.

APPENDIX E

Size of Programs

In this appendix we summarize comparison data between the size of the Multics system 24.2 security kernel and the size of our proposed Multics security kernel. We have only included data for the major programs that were affected by our design. As a basic measure of the size of a procedure we have chosen the number of words of text in its Multics object code module. This corresponds roughly to the number of machine instructions in the module. We notice that in most cases the procedures in our system are markedly smaller then their counterparts in system 24.2. Our reduction of the security kernel by 3345 words or about two and a half per cent may not appear spectacular, but the reduction in size of the address space manager is seventy-seven per cent. This has substantially reduced the complexity of the security kernel. The reason we can make this claim is that while the reference name manager in system 24.2 is not that large, it is complex far out of proportion to its size.

old procedure	size		new procedure
find_	791	128	find_entry
makeknown	732	164	makeknown_
kstsrch	440	103	kstsrch
kst_man	45	34	get_kstep
makeunknown	1044	123	terminate_
initialize_kst	667	82	initialize_kst
initiate	698	288	initiate_
kst_entry_check	112	88	kste_info
		84	kste
		86	validate_segno
	4529	1184	

APPENDIX F

Performance Data

In order to measure the change in overall performance between our system and Multics system 24.2, we developed a special benchmark program. This benchmark was designed to evaluate only the most commonly used features that we modified in our design: segment initiation, reference name management, and segment termination. Specifically, our benchmark called the old ring zero initiation interface (1) to initiate a segment and give it a reference name. It then used the terminate by segment number primitive of the old interface to terminate the segment and unbind the reference name. This was repeated one hundred times. The virtual cpu time in microseconds to complete the benchmark was then divided by one hundred to obtain a normalized performance timing datum. The total number of page faults for the run was also recorded.

The benchmarks for both systems were run on December 10, 1974 within ten minutes of each other on a dedicated computer. The standard Multics system used was designated as Multics system 24.2. Our test system was identical to system 24.2 except as it implemented our design. Three runs were made on each system. The first run served only to cause dynamic linking to occur and to bring the pages that our benchmark

(1) The old ring zero interfaces were simulated in our system.

touches into primary memory. The second run, which took no page faults, was used to obtain our timing data. (1) Multics system 24.2 averaged 11002 microseconds for each iteration of our benchmark. Our test implementation was actually seven per cent faster, taking 10226 microseconds per interation. The final run was made after the contents of primary memory were flushed. This run established the size of the working set of our benchmark since each page touched while running our benchmark produced a missing page fault. The working set of our benchmark in Multics 24.2 was five pages. Our test implementation had a working set of seven pages.

(1) Prior testing had shown that multiple runs of the benchmark, under identical conditions, produced times within one hundredth of one per cent of each other. As a result one timing run was all that was required.

APPENDIX G

Ring Zero Interface Complexity Data

This appendix lists briefly the changes we have made in the ring zero interface of Multics system 24.2. We have excluded from this appendix the changes we have made to the ring zero address space manager interface as these changes have been documented in appendix C.

Obsoleted Interfaces

```
hcs_$chname_file
hcs_$fs_get_path_name
hcs_$delentry_file
hcs_$fs_get_ref_name
hcs_$fs_get_seg_ptr
hcs_$status_minf
hcs_$terminate_file
hcs_$terminate_name
hcs_$terminate_noname
hcs_$truncate_file
hcs_$set_bc
```

Interfaces Converted To Specifying Their Target Object

By Segment Number Rather Than

By Directory Pathname and Entry Name

hcs_$add_acl_entries
hcs_$add_dir_acl_entries
hcs_$add_dir_iacl_entries
hcs_$add_iacle_entries
hcs_$del_dir_tree
hcs_$delete_acl_entries
hcs_$delete_dir_acl_entries
hcs_$delete_dir_iacl_entries
hcs_$delete_iacle_entries
hcs_$get_author
hcs_$get_bc_author
hcs_$get_dir_ring_brackets
hcs_$get_max_length
hcs_$get_ring_brackets
hcs_$get_safety_sw
hcs_$get_user_effmode
hcs_$list_acl
hcs_$list_dir_acl
hcs_$list_dir_iacl
hcs_$list_inacl
hcs_$quota_move
hcs_$replace_acl
hcs_$replace_dir_acl
hcs_$replace_dir_inacl
hcs_$replace_inacl
hcs_$set_copysw
hcs_$set_dir_ring_brackets
hcs_$set_max_length
hcs_$status_
hcs_$status_long
hphcs_$add_acl_entries
hphcs_$add_dir_acl_entries
hphcs_$delete_acl_entries
hphcs_$delete_dir_acl_entries
hphcs_$replace_acl
hphcs_$replace_dir_acl
hphcs_$set_act
hphcs_$set_auth
hphcs_$set_bc_auth
hphcs_$set_dates
hphcs_$set_dir_ring_brackets
hphcs_$set_ring_brackets
hphcs_$status_backup_info

Interfaces Converted To Specifying Their Target Object

By Segment Number Rather Than

By Directory Pathname

```
hcs_$append_branch
hcs_$append_branchx
hcs_$append_link
hcs_$quota_get
hcs_$star_
hcs_$star_list_
hphcs_$quota_reload
hphcs_$quota_set
hphcs_$salvage_dir
hphcs_$star_no_acc_ck
```

APPENDIX H

The Address Space Manager Programs

We have claimed that the address space manager we designed is simple, small and easy to certify. To substantiate this claim, we are including in this appendix the source code of our address space manager for the reader's perusal. These programs differ from the actual programs that ran in our trial Multics system only in a few minor details. (1)

We will divide this appendix into three sections. The first section contains a declaration for the KST. This declaration is used by programs that contain a "%include kst;" statement. The second section contains the PL/I source programs that constitute the address space manager. Finally, the third section describes the calling sequence and functionality of system modules called by the programs presented in section two.

The baseno and ptr PL/I builtin functions used in the programs in this appendix are non-standard PL/I functions used in Multics to manipulate pointers. A Multics pointer may be viewed as a pair of integer values. The first component of a pointer is interpreted as a segment number by the Multics hardware. The second component of a pointer is interpreted as a word offset within the segment specified by the first component. The baseno

(1) See appendix I.

builtin function constructs a pointer to the first word in a segment given a segment number for that segment. The ptr builtin function constructs a pointer from the segment number in its first argument, which must be a pointer, and the integer offset which is its second argument.

```
/*  BEGIN INCLUDE FILE  - - - kst.incl.pl1 - - - */

dcl  kst_seg$ ext;                                     /* kst segment */

dcl 1 kst aligned based (addr (kst_seg$)),             /* KST header declaration */
    2 lowseg fixed bin,                                /* lowest segment number described by kst */
    2 highseg fixed bin,                               /* highest segment number described by kst */
    2 free_list,                                       /* free list */
      3 (fp, bp) bit (18) unaligned,
    2 uid_hash (0: 127),                               /* uid hash table  */
      3 (fp, bp) bit (18) unaligned,
    2 entry (lowseg:highseg) like kste;

dcl  kstep ptr;                                        /* pointer to entry */

dcl 1 kste based (kstep) aligned,                      /* KST entry declaration */

    (2 fp bit (18),                                    /* forward rel pointer */
    2 bp bit (18),                                     /* backward rel pointer */

    2 segno fixed bin (17),                            /* segment number of kste */

    2 rings bit (8),                                   /* rings in which this segment is known */
    2 hdr fixed bin (3),                               /* highest detectable ring */
    2 dirsw bit (1),                                   /* directory type switch */
    2 unused bit (5),                                  /* unused bits */
    2 infcount fixed bin (17),                         /* inferior segment count */

    2 entryp ptr) unaligned,                           /* ptr to dir entry */

    2 id bit (36) aligned;                             /* unique identifier */

/*  END INCLUDE FILE - - - kst.incl.pl1 - - - */
```

```
initialize_kst:
     proc (lowseg, highseg);

/*

   initialize_kst is called during process initialization to build a virgin kst
   USAGE: call initialize_kst (lowseg, highseg);

   lowseg fixed bin (17) - - - lowest segment number described  by kst
   highseg fixed bin (17) - - - highest segment number described  by kst

*/

dcl (lowseg, highseg, i) fixed bin (17),
     thread$in ext entry (ptr, ptr);

% include kst;

          kst.lowseg = lowseg;
          kst.highseg = highseg;
          kst.free_list = "0"b;
          kst.uid_hash = "0"b;
          do i = lowseg to highseg;
               call thread$in (addr (kst.free_list), addr (kst.entry (i)));
               kst.entry (i).segno = i;
          end;
     end initialize_kst;
```

```
initiate_: proc (a_psegno, a_ename, a_dirsw, a_link, a_segno, a_code) ;
/*
   ---> initiate_ is the ring zero gate which allows an object to be mapped
   into a process' address space. This module only validates its caller's
   right to initiate the object in question. If the request is valid then
   makeknown_ is called to actually map the object into the process' address space.
   USAGE: call initiate_(psegno, ename, dirsw, link, segno, code);

   psegno fixed bin(17) --- segment number of parent directory (input)
   ename char(*) --- name of entry in directory to initiate (input)
   dirsw bit(1) --- set if entry is a directory (input)
   link char(*) varying --- link (output)
   segno fixed bin (17) --- segment number of target (output)
   code fixed bin(35) --- status code (output)

   possible status code values:
   error_table_$segknown --- segment (or directory) already known to process
   error_table_$noinfo --- insufficient access to return any information
   error_table_$nrmkst --- no more room in known segment table
   error_table_$no_entry --- entry does not exist
   error_table_$link --- entry is a link
   error_table_$invalidsegno --- invalid parent segment number
   error_table_$notadir --- parent is not a directory
   error_table_$wrong_type --- target object is of the wrong type          */

dcl  a_ename char (*),
    (a_dirsw, dirsw, noinfo) bit (1),
     a_link char (*) varying,
     link char (168) varying,
    (segno, a_segno, psegno, a_psegno) fixed bin (17),
    (code, a_code) fixed bin (35),
     ep ptr,
     uid bit (36) aligned,
     type fixed bin (17),
     ename char (32) aligned,
     phoney_uid bit (36) aligned static init ("0"b),
     accessible bit (1) static init ("1"b),
     no_entry fixed bin (17) static init (0),
     segment fixed bin (17) static init (1),
     directory fixed bin (17) static init (2),
     lnk fixed bin (17) static init (3);

dcl (error_table_$noinfo, error_table_$segknown, error_table_$nrmkst, error_table_$no_entry,
     error_table_$invalidsegno, error_table_$notadir, error_table_$link,
     error_table_$wrong_type) ext fixed bin (35);

dcl  get_branch_info entry (fixed bin (17), char (32) aligned, fixed bin (17), bit (36) aligned,
     ptr, char (*) varying, fixed bin (35)),
     validate_segno$inuse ext entry (fixed bin (17)) returns (ptr),
     makeknown_ ext entry (ptr, bit (36) aligned, bit (1), fixed bin (17), bit (1), fixed bin (35));

%include kst;
/*
```

```
    */
     psegno = a_psegno;                                      /* copy input arguments */
     dirsw = a_dirsw;                                        /* so our caller cant change them */
     ename = a_ename;
     kstep = validate_segno$inuse (psegno);
     if kstep = null () then call return_code (error_table_$invalidsegno);
     if ^kste.dirsw then call return_code (error_table_$notadir);
     if kste.id = phoney_uid
     then if dirsw
          then call MKN (baseptr (psegno), phoney_uid, ^accessible);
          else call return_code (error_table_$noinfo);
     else do;
          call get_branch_info (psegno, ename, type, uid, ep, link, code);
          if code = error_table_$noinfo
          then if dirsw
               then if type ^= directory
                    then call MKN (baseptr (psegno), phoney_uid, ^accessible);
                    else call MKN (ep, uid, ^accessible);
               else call return_code (code);
          else do;
               if type = no_entry then call return_code (error_table_$no_entry);
               if type = lnk
               then do;
                    a_link = link;
                    call return_code (error_table_$link);
               end;
               if (type = segment) & dirsw | (type = directory) & ^dirsw
               then call return_code (error_table_$wrong_type);
               else call MKN (ep, uid, accessible);
          end;
     end;
MKN: proc (ep, uid, accessible);
dcl  ep ptr,
     uid bit (36) aligned,
     accessible bit (1);
          call makeknown_ (ep, uid, dirsw, segno, accessible, code);
          a_segno = segno;
          call return_code (code);
     end MKN;
return_code: proc (code);
dcl  code fixed bin (35);
          a_code = code;
          go to return_to_caller;
     end return_code;
return_to_caller: return;

     end initiate_;
```

```
makeknown_:
     proc (ep, entry_uid, dirsw, segno, accessible, code);

/*

   ---> makeknown_ maps a segment or directory (specified by dirsw) into its
   caller's address space. This module assumes that the process' right to
   initiate the segment specified has already been established. It further
   assumes that its input arguments will not be modified while it is executing.
   This assumption requires its callers to be sure that arguments passed
   to makeknown_ are not accessible to outer ring procedures.
   USAGE: call makeknown_ (ep, entry_uid, dirsw, segno, accessible, code);

   ep ptr --- pointer to the object's branch (input)
   entry_uid bit(36) aligned --- unique identifier of the object (input)
   dirsw bit(1) --- set if object is a directory (input)
   segno fixed bin(17) --- segment number bound to the object (output)
   accessible bit(1) --- set if process has access to the object or its parent (input)
   code fixed bin(35) --- status code (output)

*/

dcl  ep ptr,
     entry_uid bit (36)aligned,
     dirsw bit (1),
     segno fixed bin (17),
     accessible bit (1),
     code fixed bin (35);

dcl  ring fixed bin (3),
    (error_table_$segknown, error_table_$noinfo) ext fixed bin (35),
    (pkstep, hashp) ptr;

dcl  level$get ext entry () returns (fixed bin (3)),
     get_kstep ext entry (fixed bin (17)) returns (ptr),
     kstsrch ext entry (bit (36) aligned, bit (1), ptr, ptr),
     thread$in ext entry (ptr, ptr),
     kste$reserve ext entry (fixed bin (17), ptr, fixed bin (35));

dcl (baseno, fixed, null, substr) builtin;

          %include kst;

/*
```

```
*/
          ring = level$get ();
          call kstsrch (entry_uid, accessible, hashp, kstep);
          if kstep ^= null ()
          then do;                                          /* object already has a detectable KSTE */
               code = error_table_$segknown;
               segno = kste.segno;
          end;
          else do;                                          /* must allocate a new KSTE */
               if ^accessible then code = error_table_$noinfo;
               call kste$reserve (segno, kstep, code);
               if code ^= 0 then return;
               call thread$in (hashp, kstep);               /* thread KSTE into hash class */
               if ep ^= null ()
               then do;                                     /* increment parent's inferior count */
                    pkstep = get_kstep (fixed (baseno (ep), 17));
                    pkstep -> kste.infcount = pkstep -> kste.infcount+1;
               end;
               kste.dirsw = dirsw;                          /* fill in KSTE */
               kste.infcount = 0;
               kste.entryp = ep;
               kste.id = entry_uid;
          end;
          substr (kste.rings, ring+1, 1) = "1"b;            /* mark kste as known in proper ring */
          if accessible
          then do while (ring > kste.hdr);                  /* update hdr of superiors */
               kste.hdr = ring;
               if kste.entryp ^= null ()
               then kstep = get_kstep (fixed (baseno (kste.entryp), 17));
          end;
          return;
     end makeknown_;
```

```
terminate_: proc (a_segno, a_code);

/*

   ---> terminate_ is the gate into ring zero which allows a process to unbind
   a segment number from the object to which it was bound. If the KSTE has no
   inferiors and the segment number is not in use by other rings then the
   segment number is physically disconnected from the object to which it was bound
   and the segment number is returned to the free or reserved pool as specified
   by the reserved switch argument. If these conditions do not obtain then the
   segment number is not disconnected. The KSTE is merely marked as no
   longer in use in the caller's protection ring.
   USAGE: call terminate_ (segno, code)

   segno fixed bin(17) - - - segment number of the segment
   code fixed bin (35) - - - error code (output)

   possible status code values:

   error_table_$invalidsegno --- segment number is not bound to an object
   error_table_$infcnt_non_zero --- can't terminate due to active inferiors
   error_table_$known_in_other_rings --- can't terminate due to segment number being used in other rings
   */

dcl  a_segno fixed bin (17),
     a_code fixed bin (35);

dcl  kstep ptr,
     ring fixed bin,
     segno fixed bin (17);

dcl  disconnect ext entry (fixed bin (17)),
     get_kstep ext entry (fixed bin (17)) returns (ptr),
     kste$free ext entry (ptr),
     thread$out ext entry (ptr),
     validate_segno$inuse ext entry (fixed bin (17)) returns (ptr),
     level$get ext entry returns (fixed bin);

dcl (error_table_$known_in_other_rings, error_table_$infcnt_non_zero) ext fixed bin (35);
dcl  error_table_$invalidsegno ext fixed bin (35);

dcl (baseno, fixed, null, substr) builtin;

          % include kst;
/*
```

```
*/
                                                                    /* copy values of input arguments */
          segno = a_segno;                                          /* so our caller can't change them */
          kstep = validate_segno$inuse (segno);                     /* make sure call is legal */
          if kstep = null ()
          then call abort (error_table_$invalidsegno);
          ring = level$get ();
          substr (kste.rings, ring+1, 1) = "0"b;                    /* make unknown in this ring */
          if kste.rings  = "0"b
          then call abort (error_table_$known_in_other_rings); /* can't terminate in another ring */
          if kste.infcount  = 0
          then call abort (error_table_$infcnt_non_zero);      /* can't terminate if infcount non zero */
          if kste.entryp  = null ()
          then do;                                                  /* decrement parent's inferior count */
               pkstep = get_kstep (fixed (baseno (kstep -> kste.entryp), 17));
               pkstep -> kste.infcount = pkstep -> kste.infcount-1;
          end;
          call disconnect (segno);
          call thread$out (kstep);
          call kste$free (kstep);                                   /* deposit kste in free pool */
          a_code = 0;
          return;

abort:    procedure (status_code);
dcl  status_code fixed bin (35);
               a_code = status_code;
               go to return;
          end abort;

return:   return;

     end terminate_;
```

```
kstsrch: proc (uid, accessible, hashp, kstep);

/*

   ---> kstsrch searches the KST unique identifier hash table and returns pointers
   to the KSTE desired and the hash class thread word. Only if the process has established
   its right to detect the existence of the object bound to the KSTE will a match be found.
   The conditions required for kstsrch to return a given segment number are:
   1) the segment number must be bound to the correct object (as identified by uid),
   2) the segment number must be detectable in the caller's ring, and
   3) no higher ring may have the segment number initiated. At the expense of assigning multiple
   segment numbers to an object when not necessary for protection reasons, kstsrch could
   use a weaker matching algorithm such as matching only if the caller has access to the target
   object or the parent of the target object.
   USAGE:  call kstsrch(uid, accessible, hashp, kstep);

   uid bit(36) aligned ---- unique id of object searched for (input)
   accessible bit(1) ---- set if the process has any access to the object or its parent (input)
   hashp ptr ---- pointer to the hash class thread word (output)
   kstep ptr ---- pointer to the desired KSTE if found else null (output)

*/

dcl  uid bit (36) aligned,
     accessible bit (1),
    (ring, hdr) fixed bin (3),
     hashp ptr,
    (addr, ptr, null, mod, dimension) builtin,
     level$get ext entry () returns (fixed bin (3));

          %include kst;

          ring = level$get ();
          hashp, kstep = addr (kst.uid_hash (mod (fixed (uid), dimension (kst.uid_hash, 1))));
          do while (kste.fp ^= "0"b);
               kstep = ptr (kstep, kste.fp);
               if match () then return;
          end;
          kstep = null ();
          return;
match:    proc () returns (bit (1));
               if uid = kste.id & (accessible | kste.hdr >= ring)
               then do;
                    if accessible
                    then hdr = max (kste.hdr, ring);
                    else hdr = kste.hdr;
                    if substr (kste.rings, hdr + 2, 7 - hdr) = "0"b then return ("1"b);
               end;
               return ("0"b);
          end match;
     end kstsrch;
```

```
kste: proc ();

/*
   kste provides the functions of freeing and reserving segment numbers

   ---> kste$reserve extracts a kste from the free list
   USAGE: call kste$reserve (segno,kstep,code);

   ---> kste$free frees a segment number given a kst entry pointer
   The kste is threaded onto the free list.
   USAGE: call kste$free (kstep);

   segno fixed bin (17) - - - segment number (output)
   kstep ptr - - - pointer to the kstep (input/output)
   code fixed bin(35) - - - error code (output)
   */


dcl  code fixed bin (35),
    (segno, save_segno) fixed bin (17) ;

dcl  thread$in ext entry (ptr, ptr),
     thread$out ext entry (ptr);

dcl (addr, ptr, unspec) builtin;

dcl  error_table_$nrmkst ext fixed bin (35);

          % include kst;
reserve:
          entry (segno, kstep, code);
          if kst.free_list.fp = "0"b
          then do;
               code = error_table_$nrmkst;
               return;
          end;
          kstep = ptr (addr (kst), kst.free_list.fp);
          call thread$out (kstep);
          segno = kste.segno;
          kste.fp, kste.bp = "0"b;                         /* terminate chains */
          code = 0;
          return;
free:
          entry (kstep);
          save_segno = kste.segno;
          unspec (kste) = "0"b;
          kste.segno = save_segno;
          call thread$in (addr (kst.free_list), kstep);
          return;

     end kste;
```

```
get_kstep: proc (segno) returns (ptr);

/*
   --->  get_kstep translates a segment number into a pointer to the associated KSTE
   USAGE:  kstep = get_kstep (segno);

   1) segno fixed bin(17) ---- the segment number
   2) kstep ptr ---- pointer to a KSTE

*/

          % include kst;

dcl  segno fixed bin (17),
    (null, addr) builtin;

          if segno < kst.lowseg ¦ segno > kst.highseg
          then return (null ());
          return (addr (kst.entry (segno)));
     end get_kstep;
```

```
validate_segno:
     proc ();

/*

   validate_segno provides generally useful kste validation functions
   Each entry returns a pointer to the associated kste if a particular conditions holds.
   If the stated condition does not obtain then the null pointer is returned.

   ---> validate_segno$free checks to see that the segment number is free
   USAGE: kstep = validate_segno$free (segno);

   ---> validate_segno$inuse checks to see that the segment number is bound to an object
   USAGE: kstep = validate_segno$inuse (segno);

   segnc fixed bin (17) - - - segment number (input)
   kstep ptr - - - pointer to the kstep (output)

   */


dcl  segno fixed bin (17);

dcl  get_kstep ext entry (fixed bin (17)) returns (ptr);

dcl (null, unspec) builtin;

          %include kst;

free:     entry (segno) returns (ptr);
          return (eval ("1"b));

inuse:    entry (segno) returns (ptr);
          return (eval ("0"b));


eval:     proc (unassigned) returns (ptr);
dcl  unassigned bit (1) aligned;
               kstep = get_kstep (segno);
               if kstep = null () then return (null ());
               if unassigned ^= (unspec (kste.entryp) = "0"b) then return (null ());
               return (kstep);
          end eval;

     end validate_segno;
```

```
kste_info:  proc (segno, uid, branchp, code);

/*
   ---> kste_info returns the uid of the object bound to a segment number
   as well as the address of the object's branch. This information is used
   to lock the parent directory and locate the desired branch.
   USAGE: call kste_info (segno, uid, branchp, code);

   ---> kste_info$update_branch_offset is called by the file system when it notices that
   the online salvager has moved an entry in a directory.
   It updates the pointer in the kste to reflect the new location of
   of the branch within the directory.
   USAGE: call kste_info$update_branch_offset (segno, branch_offset);

   segno fixed bin (17) ---- segment number of the object (input)
   uid bit (36) aligned ---- unique identifier of the object (output)
   branchp ptr ---- branch pointer (output)
   branch_offset bit (18) aligned ---- offset of branch of object in parent (output)
   code fixed bin(35) ---- status code (output)
*/

dcl  segno fixed bin (17),
     code fixed bin (35),
     branchp ptr,
     branch_offset bit (18) aligned,
     uid bit (36) aligned;

dcl (error_table_$invalidsegno, error_table_$noentry) ext fixed bin (35);
dcl  validate_segno$inuse ext entry (fixed bin (17)) returns (ptr);

          %include kst;

          kstep = validate_segno$inuse (segno);
          if kstep = null ()
          then do;
               code = error_table_$invalidsegno;
               return;
          end;
          uid = kste.id;
          if kste.entryp = null ()
          then do;
               code = error_table_$noentry;
               return;
          end;
          branchp = kste.entryp;
          code = 0;
          return;
update_branch_offset:
          entry (segno, branch_offset);
          kst.entry (segno).entryp = ptr (kst.entry (segno).entryp, branch_offset);
          return;
     end kste_info;
```

---> get_branch_info

This file system routine is called by initiate_ to get the attributes of a named entry in a directory. If the caller has no access to the named object (if it exists) or to the parent directory then the status code error_table_$noinfo is returned. The reader should note that get_branch_info must read the access control list of the directory containing the named entry if the entry does not exist or if the process has no access to the entry. To locate the access control list of the containing directory, get_branch_info must call the kste_info module of the address space manager, a recursive invocation of the address space manager.

```
Usage: call get_branch_info (psegno, ename, type, uid, ep,  link,
        code);

  psegno fixed bin (17) --- directory segment number (input)
  ename char (32) aligned --- name of entry in directory  (input)
  type fixed bin (17) --- type of the object (output)
          0 -- no entry
          1 -- segment
          2 -- directory
          3 -- link
  uid bit (36) aligned --- unique identifier of object (output)
  ep pointer --- pointer to the entry of the object (output)
  link char(*) varying --- contents of the link (output)
  code fixed bin (35) --- error code (output)
```

---> thread$in

This routine adds an element to a two way linked list of elements. The first word of each element contains the necessary forward and backward pointers.

Usage: call thread$in (where, what);

where pointer --- pointer to an element in the list after which the new element is to be threaded.
what pointer --- pointer to the element to be threaded into the list.

---> thread$out

This routine threads an element out of a two way linked list built by thread$in.

Usage: call thread$out (what);

what pointer --- pointer to the element to be threaded out of the list.

---> level$get

This routine returns the validation level of the calling procedure. In all cases considered in this thesis the validation level of a process is equal to the number of the ring in which the process was executing when it called into ring zero.

Usage: ring = level$get ();

ring fixed bin (3) --- validation level of the process.

---> disconnect

This routine physically removes a segment number from a process' address space by zeroing the segment descriptor word for that segment number in the process' virtual address translation table.

Usage: call disconnect (segno);

segno fixed bin (17) --- segment number to be disconnected.

APPENDIX I

Unimplemented Address Space Manager Functions

In our discussion of the Multics address space manager we omitted three mechanisms that it currently supports. These mechanisms, which are non-essential to our design, were omitted to simplify our presentation and avoid confusion. In this appendix we will briefly describe these mechanisms and show how they fit into our design.

I.1 Reserved Switch

The Multics initiation and termination primitives take a reserved switch argument. In the case of initiation, this switch specifies, if set, that the caller wishes to specify what segment number to bind to the object when it is initiated. Naturally, ring zero must check that the caller has in fact reserved the segment number. When the ring zero initiation primitive is called without the reserved switch set, then ring zero chooses a segment number from a list it maintains of free segment numbers. This segment number is bound to the object and returned to the caller. In the case of termination, the reserved switch specifies whether the freed segment number is to be eligible for assignment when a free segment number is needed.

The reserved switch must clearly remain a protected security kernel mechanism in our new address space manager. Were this not the case, one protection domain could cause another protection domain to malfunction by using a segment number that the first protection domain had reserved.

I.2 Copy Switch

During the process of initiating a segment, an attribute in its directory entry called a copy switch is examined. If the segment has the copy attribute, then a copy of the segment is made and this copy is made accessible to the process instead of the original.

We can use the mechanism of reflecting information out to an outer ring by setting a status code to remove copy switch processing from ring zero. This is possible since the current initiation primitive takes an argument that allows a process to bypass copy switch processing. Together with the fact that no ring zero procedures or data bases have their copy switch set, this insures that the protection mechanisms of the system do not depend upon the segment copy on initiation facility. To take advantage of this, our new initiate primitive will not process the copy switch. Instead, it will always initiate the target segment and return a status flag indicating whether or not the segment's copy switch was set. The outer rings can then worry about creating a copy of the segment, terminating the original, and returning the segment number of the copy if the copy switch was set. This allows the concept of a copy switch to move out of ring zero.

I.3 Transparency Switches

When a segment is initiated in the current Multics system, the address space manager sets two switches, called the transparent usage switch and the transparent modification switch, in its KSTE. These switches determine whether this process' usage and modification of the segment is to be detectable to other processes in the system. These transparency switches have no influence upon our design except that in an implementation of our design (as in our test implementation) these switches would be kept in the KSTE of a segment and the address space manager would retain the two lines of code from the current address space manager that sets these switches.

Bibliography

B1 Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics Virtual Memory: Concepts and Design," *CACM 15*, 5 (May 1972), pp. 308-318.

C1 Corbatô, F. J., J.H. Saltzer, and C.T. Clingen, "Multics -- The First Seven Years," *AFIPS Conf. Proc. 40* (1972 SJCC), AFIPS Press: Montvale, N.J.

C2 Corbatô F.J., and Vyssotsky, V.A., "Introduction and Overview of the Multics System," *AFIPS Conf. Proc. 27* (1965 FJCC), Spartan Books: Washington, D.C.

D1 Dijkstra E.W., "Complexity controlled by hierarchical ordering of function and variability," *Software Engineering* (P. Naur and B. Randell, eds.), NATO Scientific Affairs Division: Brussels, January 1969, pp. 181-185.

D2 Dijkstra E.W., "The structure of the "THE"-multiprogramming system," *CACM 11*, 5 (May 1968), pp. 341-346.

D3 Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., *Structured Programming*, Academic Press: New York, N.Y., 1972.

F1 Fabry, R.S., "Capability-Based Addressing," *CACM 17*, 7 (July 1974), pp. 403-412.

J1 Janson, P.A., "Removing the Dynamic Linker from the Security Kernel of a Computing Utility," MIT Project MAC Technical Report TR-132, 1974.

I1 IBM, "IBM OS Linkage Editor", IBM Systems Reference Library, GC 28-6538, January 1972.

L1 Liskov, B. H., "A design methodology for reliable software systems," *AFIPS Conf. Proc. 41* (1972 FJCC), AFIPS Press: Montvale, N.J.

M1 Mills, H.D., "On the development of large reliable programs," *Proceedings of the IEEE Symposium on Computer Software Reliability*, 1973.

M2 M.I.T. Project MAC, *Introduction to Multics*, MIT Project MAC Technical Report TR-123, 1974.

M3 Madnick, S.E., "Design Strategies for File Systems," MIT Project MAC Technical Report TR-78, 1970.

M4 McCarthy, J., Abrahams, P., et al., Lisp 1.5 Programmer's Manual, MIT Press: Cambridge, Mass., 1965.

N1 Naur, P. and B. Randell (Eds.), Software Engineering, report by the NATO Science Committee, Garmisch, Germany, 1968.

O1 Organick, E.I., The Multics System: An Examination of its Structure, MIT Press: Cambridge, Mass., 1972.

P1 Parnas, D.L., "A technique for software module specification with examples," CACM 15, 5 (May 1972), pp. 330-336.

R1 Rotenberg, L.J. "Making Computers Keep Secrets," MIT Project MAC Technical Report TR-115, 1974.

R2 Ritche, D.M. and Thompson K., "The UNIX Time-Sharing System," CACM 17, 7 (July 1974), pp. 365-375.

S1 Schroeder, M.D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," MIT Project MAC Technical Report TR-104, 1972.

S2 Schroeder, M.D. and J.H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," CACM 15, 3 (March 1972), pp. 157-170.

S3 Saltzer, J.H., "Protection and the Control of Information Sharing in Multics," CACM 17, 7 (July 1974), pp. 388-402.

S4 Saltzer, J.H., and M.D. Schroeder, "The Protection of Information in Computer Systems," IEEE Proc., 63, 9 (Sept. 1975), pp. 1278 - 1308.

W1 Wirth, N., "Program Development by Stepwise Refinement," CACM 14, 4 (April 1970), pp. 221-227.

W2 Wirth, N., Systematic programming introduction, Prentice-Hall: Englewood Cliffs, New Jersey, 1973.

CS-TR Scanning Project
Document Control Form

Date : 11 / 16 / 95

Report # LCS-TR-156

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR)   ☐ Technical Memo (TM)
☐ Other: ____________________

Document Information

Number of pages: 130 (134-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:

☐ Single-sided or

☒ Double-sided

Intended to be printed as :

☐ Single-sided or

☒ Double-sided

Print type:

☐ Typewriter   ☐ Offset Press   ☐ Laser Print
☐ InkJet Printer   ☒ Unknown   ☐ Other: ____________

Check each if included with document:

☐ DOD Form   ☐ Funding Agent Form   ☐ Cover Page
☐ Spine   ☐ Printers Notes   ☐ Photo negatives
☐ Other: ____________________

Page Data:

Blank Pages(by page number): FOLLOWING LAST PAGE 129

Photographs/Tonal Material (by page number): ____________

Other (note description/page number):

Description :   Page Number:

IMAGE MAP! (1-130) UN#'ED TITLE PAGE 2-129, UN# BLANK
(131-134) SCANCONTROL, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 11 / 16 / 95   Date Scanned: 12 / 1 / 95   Date Returned: 12 / 7 / 95

Scanning Agent Signature: Michael W. Cook