MITRE Technical Report
MTR-2934

The Design and Specification of a Security Kernel for the PDP-11/45

W.L. Schiller

March 1975

CONTRACT SPONSOR ESD
CONTRACT NO. F19628-75-C-0001
PROJECT NO. 572B
DEPT. D73

THE MITRE CORPORATION
BEDFORD, MASSACHUSETTS

ABSTRACT

This paper presents the design of a kernel for certifiably secure computer systems being built on the Digital Equipment Corporation PDP-11/45. The design applies a general purpose mathematical model of secure computer systems to an off-the-shelf computer. An overview of the model is given. The paper includes a specification of the design that will be the basis for a rigorous proof of the correspondence between the model and the design. This design and implementation has demonstrated the technical feasibility of the security kernel approach for designing secure computer systems.

This work was carried out by The MITRE Corporation under contract to the United States Air Force Electronic Systems Division, Contract F19628-75-C-0001.

PREFACE

The security kernel design given in this paper is a major revision of a kernel design described in [Schiller]. In the original design a distinction was made between the information and control structures of a computer system, and the access controls dictated by our mathematical model of secure computer systems were only applied to the information structure. To protect the control structure we stated that "it is the responsibility of the system designer to systematically determine all possible channels through the control structure . . . (and prevent) the associated state variable from being controlled and/or observed". After that design was published it became obvious that the approach to protecting the control structure was not adequate. The systematic determination of channels was equivalent to having a model that protected the control structure.

Consequently, refinements were added to the model to allow the same mechanisms to protect both the information and control structure objects of a system. The basic technique used is to organize all of the data objects in the system into a tree-like hierarchy, and to assign each data and control object explicit security attributes. The major difference between the revised design given in this paper and the original design is the incorporation of the model refinements. In addition, this paper benefits from an additional year's study and understanding of the computer security problem. Familiarity with the original design is not required.

ACKNOWLEDGEMENTS

Many individuals have contributed to the ideas in this paper. I would like to give special thanks to Ed Burke, Steve Lipner and Roger Schell for many helpful discussions over the past two years. Jon Millen contributed to the form and content of the specification. And finally, I am indebted to Betty Aprile for spending many hours with a recalcitrant computer system to help produce this paper.

TABLE OF CONTENTS

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I

BACKGROUND

INTRODUCTION

The PDP-11/45 Secure System Design is intended to provide a design, based on a general purpose mathematical model of secure computer systems, for building secure systems on the Digital Equipment Corporation PDP-11/45, an off-the-shelf computer. The primary goal of the design is to bridge the gap between the abstract secure system defined by the model and the elements of state-of-the-art hardware and software systems. A secondary goal is to develop a design that applies to specific systems to be implemented on the PDP-11/45. The approach taken has been to apply the model, which is completely general in its nature, to a design that will support non-trivial systems with security requirements. The model is applied to the PDP-11/45 hardware which is sufficient to support the model and secure systems, but not as complex as other available hardware. The technical issues of special interest are the mathematical modeling of secure systems, the secure system software design, and the impact of hardware on the design. This report presents the design and discusses the decisions made in generating the design. This section provides background for understanding the general problem area and the approach taken in attacking it. (Some of the material in this section has been taken from [ESD] and [Lipner].)

THE PROBLEM OF COMPUTER SECURITY

As larger, more powerful computers are employed for Air Force information systems, the desirability of operating in a "multi-level security" mode increases. A computer operating in such a mode performs simultaneous processing of data having different levels of classification and provides simultaneous (typically on-line) support to users with differing clearance levels. This mode of operation is desirable because it is often impractical to clear all system users for the highest level of data, or to separate the processing of different levels by time of day. The most severe multilevel security problem is presented by an "open" system - one in which uncleared users have access to a computer that is processing classified data. As recently as 1970 experts in the field felt that the provision of security for a general purpose computer system operating in an open environment was beyond the state-of-the-art.

The primary problem of computer security is that of certification: how can one assert that a system provides adequate security for a given application. The problems of certification range from specification through the production of correct hardware and software to testing. Previous work in this area has convinced us that security cannot be "added onto" existing computer systems. Current systems (IBM's OS/360/370 and Honeywell's GCOS, for example) are notoriously easy to penetrate. Attempts to "repair" contemporary systems are expensive and increase the malicious user's cost to penetrate by a negligible amount. The selective reimplementation of contemporary systems would cost even more and would at best serve to increase the cost of penetration [Anderson]. The only feasible approach to providing security (and therefore completely blocking penetration attempts) is to consider the problem of security and certification throughout the whole system development process - from specification to design, implementation, and testing.

BASIC CONCEPTS

The ESD computer security panel [Anderson] identified the concepts of a reference monitor and security kernel as fundamental to a secure computer system. The reference monitor is that portion of a computer's hardware and software which enforces the authorized access relationships between subjects and objects. Subjects are system entities such as a user or a process that can access system resources, and objects are system entities such as data, programs, and peripheral devices that can be accessed by subjects. The security kernel for a specific computer is the software portion of the reference monitor and access control mechanisms. The reference monitor must meet three essential design requirements:

First, the reference monitor must be tamperproof. It is obvious that if the reference monitor can be tampered with, its ability to protect programs and data can be destroyed. In the most elegant case, the reference monitor can protect itself with the same mechanisms it uses to protect other information.

Second, the reference monitor must be invoked on every attempt to access information. This requirement does not mean that the LOAD and STORE instructions of a user's process must be executed interpretively by kernel software with extensive checks. Rather, every reference must be checked by either software or hardware that is provided with sufficient information to make the correct decision on granting or denying access.

Finally, the reference monitor must be subject to certification. "Subject to certification" implies that the reference monitor's

correctness must be provable in a rigorous manner using a mathematical model as the basis for the criteria to be met.

In addition to meeting the above requirements, a reference monitor must also implement a well defined set of access control rules. In the case of a secure computer system for military use, these rules are defined by military security regulations. Basically, they require that a user be cleared to the proper level, have any formally defined special access permissions (categories) that may be required, and have a "need-to-know" before he is allowed to access information.

The approach to obtaining a secure system involves first defining the security requirements, and then creating a conceptual design that can be shown to provide the required protection (i.e., a model). The model formally defines an ideal system (in our case one that complies with military security requirements), and provides a basis for testing a subsequent implementation. Once a reference monitor that meets the requirements previously described has been implemented, computer security has been achieved. Of the software in the system, only the kernel (the software portion of the reference monitor) need be correct. The access controls and all of the other features of the hardware on which the kernel depends must be correct. The operating system proper and/or applications software can contain inadvertently introduced bugs or maliciously planted trap doors without compromising security.

SUMMARY

In this section we have presented the problem of computer security and an outline of an approach for solving it. The remainder of the report will present a design for a kernel which will serve as the basis for secure systems to be built on the PDP-11/45. The first application of the 11/45 kernel will be to support a file system for a multilevel data base.

SECTION II

TECHNICAL APPROACH

INTRODUCTION

This section presents the technical approach to designing a security kernel for the PDP-11/45. The subsections discuss the objectives of the design effort, how the design relates to the design of the Multics system, the mathematical model that is the basis for the design, hardware requirements for secure systems, and finally, some of the special problems presented by I/O (input/output processing) in a secure system.

OBJECTIVES

A long range goal of our work in the area of computer security is to solve the complete security problem. We would like to build a completely general (i.e., "computer utility") system that can be certified (proven) secure. This work includes developing mathematical models of secure computer systems to serve as a basis for subsequent designs, and identifying appropriate characteristics that the supporting hardware should have.

As a subgoal we want to build a prototype secure system to verify our ideas about computer security and apply them to perform useful work in the near term. This prototype system will not have all of the capabilities of a general purpose system and will be built on hardware that is less complex than the hardware required to support a computer utility. The advantages of building a prototype are: 1) it presents a problem of reduced complexity and therefore increases the likelihood of success in the near term, and 2) the cost of implementing a prototype system in terms of time, manpower, and equipment is much less than that of a general system. Although the initial system to be built on the 11/45 will be of limited generality in its functional characteristics, the mechanisms for achieving security will be based on completely general principles. Much that is learned will be applicable to the solution of the general computer security problem. Not only will the prototype development investigate problem areas related to the security of general purpose systems, but the resulting kernel design should be applicable to mini and medium sized computer systems with a need for multilevel security.

RELATION TO MULTICS

While most contemporary general purpose systems have notoriously ineffective security controls, there is one system, Multics [Organick], that is far superior with respect to security controls. This superiority is no accident - protection of user information was a key design goal from the inception of the Multics effort. Multics has, however, been penetrated [Karger and Schell]. One could argue that the lack of security in Multics is due to the design methodology - its design is not based on a model of secure systems and no attempt has yet been made to certify the Multics security controls. Nevertheless, Multics is the prime system that a prospective secure system designer can look to for positive guidance. Since the structure of Multics has influenced the PDP-11/45 kernel and our mathematical model of secure computer systems, a brief overview of it is given here. The material that follows is taken from [Bensoussan, Clingen, and Daley].

The key feature of Multics is its virtual memory. Multics uses segmentation to satisfy two design goals: 1) to allow all on-line information stored in the system to be directly addressable by a processor and hence available for direct reference by any computation; 2) to control access, at each reference, to all on-line information. The basic advantage of direct addressability is that the copying of data is no longer mandatory. Many users can share a single copy of a compiler or other system procedure, and users need not have an I/O system read portions of data files into main memory and then write the data back out.

If all information in the system may be directly addressable, then there is an obvious need to control access to this information both for the self-protection of a computation from its own mistakes, and for the mutual protection of users sharing the system from each other. The technique for achieving protection is to compartmentalize all information into segments, and to associate with each segment a set of access attributes for each user who may access the segment. Segments are directly addressable and the access attributes are checked by the hardware upon each segment reference by any user.

In nonsegmented systems, the use of core images makes it nearly impossible to control shared information in core. Even if the nontrival problem of addressing the shared information in core were solved, access to this information could not be controlled without additional hardware assistance. The different parts from which the core image is synthesized are indistinguishable in the core image; they have lost their identity and thereby lost all their attributes, such as length, access rights, and name. Thus, nonsegmented hardware is inadequate for controlled sharing in core memory.

In segmented systems, hardware segmentation can be used to divide a core image into several parts - segments. Each segment is addressed through a segment descriptor containing the segment's attributes; if these attributes include access at each reference, then the hardware can control access to the information in the segment at each reference.

If the number of segments that a user wishes to reference exceeds the number of segment descriptors available to him, then segmentation loses some of its effectiveness. The user may be forced to call the supervisor to free segment descriptors so that they can be reused to access other segments. This form of user controlled segment descriptor allocation can require a significant amount of pre-planning by the user. Alternatively, the user can choose to collect the information from several different segments into a single segment. This approach is a form of buffering - it requires that information be copied and lose its original identity. Multics avoids these problems by providing a number of segment descriptors sufficiently large to allow, in most cases, a segment descriptor for each segment required for a computation. The Multics supervisor also automatically associates a descriptor with a segment when the segment is first referenced by a computation. Thus, Multics users need not concern themselves with the allocation and deallocation of segment descriptors, nor need they resort to buffering information.

In a system where the maximum size of any segment is small compared to the size of main memory, it is possible to move complete segments into and out of main memory. If, in this type of system, different segments can have different current sizes, then the allocation of main memory to segments can be a difficult problem. Furthermore, if, as is the case with Multics, segments can become sufficiently large so that only a few can be entirely main memory resident at any one time, then memory allocation is made even more difficult.

The allocation of main memory is vastly simplified by dividing segments into equal-size parts called pages. Allocation of space to a segment is made on a per page basis, and all pages are the same size. In addition to simplifying allocation, paging also permits large segments to be handled with no problems because only those pages of a segment that are currently being referenced need be in main memory.

An address space in Multics is the set of segments that a process can reference with a segment number - the set of segments for which a process has descriptors. In general, each process has a unique address space. A key aspect of Multics is that its supervisor

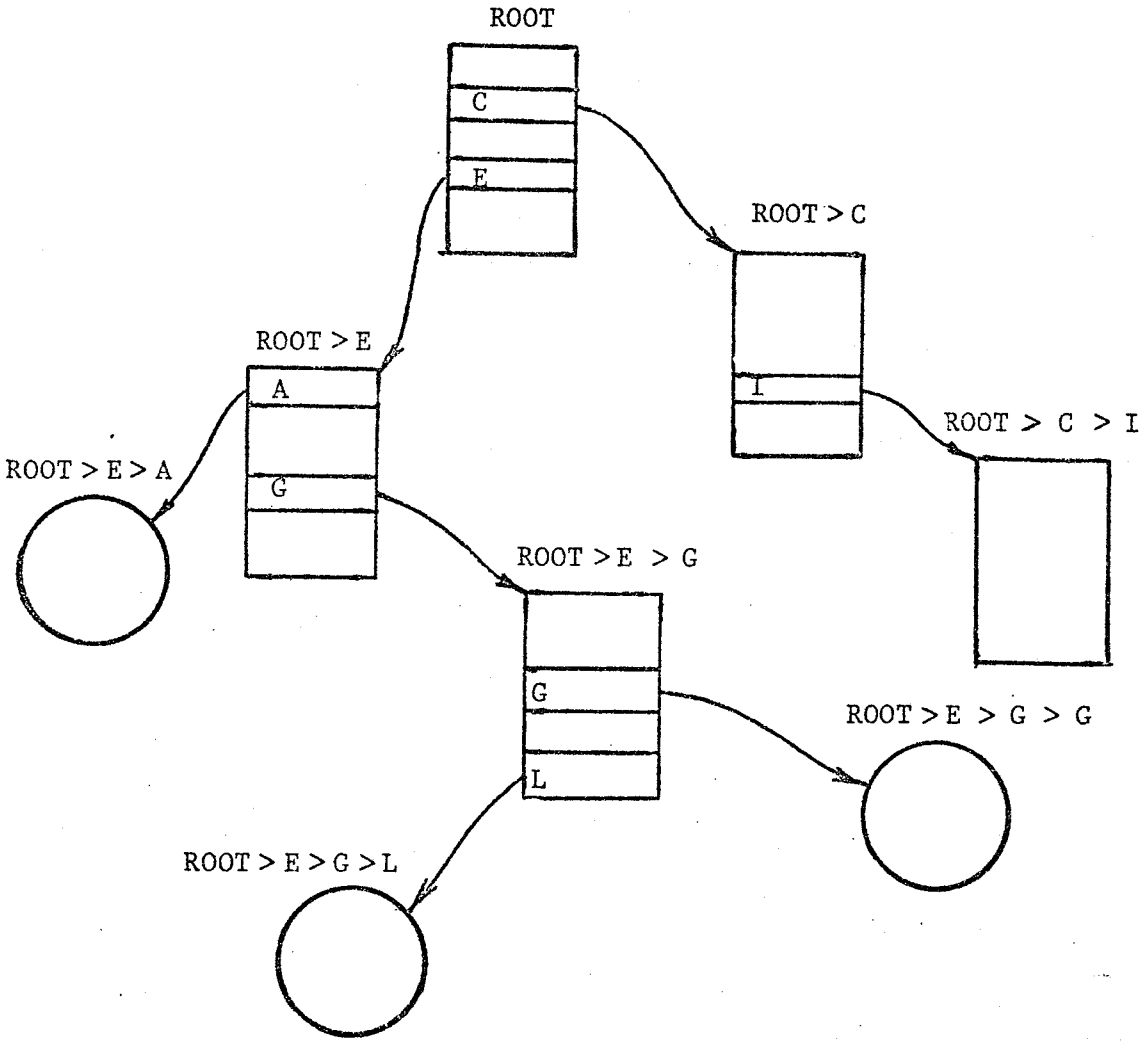
does not operate in a dedicated process or address space. Instead, the supervisor is "distributed" - its procedure and data segments are shared among all Multics processes. The execution of the supervisor in the address space of each process facilitates communication between user procedures and supervisor procedures and the simultaneous execution, by several processes, of supervisory functions. Since supervisor segments are in the address space of each process, they must be protected from user programs. This protection is achieved by having the supervisory and user procedures execute in separate domains (in Multics, protection rings).

The name of a segment and its other attributes (length, memory address, list of users allowed to access the segment, time of creation, etc.) are kept in an entry in a catalogue. In Multics, this catalogue is implemented with several segments of a special type - directory - organized into a tree structure. The name of a segment is a list of subnames that reflect the position of the segment in the tree. The base directory of the tree is called the ROOT, and subnames are separated by ">". Figure 1 shows a possible directory hierarchy.

Comparing the Multics supervisor and the PDP-11/45 security kernel is somewhat like comparing apples and oranges. Multics is a prototype computer utility that provides a variety of user-oriented services. It is supported by a powerful and complex multiprocessor; it has been operational since 1969; consists of about 300,000 lines of source code [Organick]; and it is part of the product line of a major computer system vendor (Honeywell Information Systems). The 11/45 kernel simply provides security controls for a reasonably complex general purpose environment, but it does not support user-oriented features. The 11/45 kernel is built on a straightforward, medium-sized computer. The initial implementation, which contains about 900 lines of source code, has only been operational since mid-1974, and it has not yet performed any useful work.[1] It might be appropriate to compare the Multics kernel with the 11/45 kernel, or a general purpose operating system built on the 11/45 kernel with the complete Multics supervisor, but in both cases the first item in the comparison does not currently exist. Nevertheless, it is interesting to make some comparisons between Multics and the 11/45 kernel.

The basic similarities between Multics and the 11/45 kernel are that they both implement a one level, segmented virtual memory with a directory hierarchy, and both distribute the supervisor/kernel across

[1] A simple file system to run on the 11/45 kernel is currently being implemented.



- Squares are Directory Segments
- Circles are Data Segments

Figure 1. A Simple Hierarchy

each process in the system.[2] The supporting mechanisms are similar to the extent that they can be, but differences in the supporting hardware have an impact in this area. The basic difference is that the 11/45 kernel implements a set of primitives that allow algorithms to operate within the virtual memory environment, whereas Multics provides much more than a set of primitives. While protection of information was a primary design goal from the inception of the Multics effort, and an attempt was made to isolate the protection mechanisms from the rest of the supervisor by having them execute in the most privileged domain (ring 0), the Multics system has been penetrated - the Multics protection mechanisms are not effective [Karger and Schell]. This lack of effectiveness may be due to two causes: 1) ring 0 is rather large (about 60,000 lines of source code [Saltzer (2)]) and extremely complex; and 2) the protection mechanisms are not based on a model - there was no criteria for what belonged in ring 0 and what did not. Thus the fundamental difference between the 11/45 kernel and Multics is that the 11/45 kernel is based on a model of security and Multics is not. The use of a model makes it possible to precisely define what compromises the 11/45 kernel and to rigorously prove assertions about its behavior.

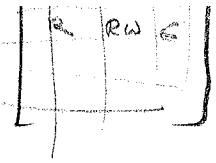
THE MATHEMATICAL MODEL

One of the key aspects of the security kernel design is that it is based on a mathematical model of secure computer systems ([Bell and LaPadula], [LaPadula and Bell], and [Bell]). The development of the model is a reaction to the inadequacy of contemporary systems. Without models for guidance, system designers are forced to apply ad hoc security-related techniques throughout the design and implementation of a system. Designers use their intuition to determine the methods of would-be penetrators and attempt to block them appropriately. But, just as testing can only indicate the presence, and not the absence, of bugs in software, penetration attempts can only demonstrate that a system is non-secure, not that it is secure. Consequently, certification of systems designed on the basis of intuition and ad hoc techniques, and tested to the point of unsuccessful penetration attempts, is not technically justified.

The model, in contrast, rigorously and precisely defines the notions of "security" and "compromise", and identifies elements that correspond to those in real systems. The model is a finite state

---

[2] Clearly, these concepts were not developed independently by both systems. The 11/45 kernel design evolved towards the Multics design as the utility of its structure as a framework for protection mechanisms became apparent.



machine model and gives a set of rules of operation for making state transitions. If the system is initialized to a secure state (and again, the notion of secure state is rigorously defined), then the rules of operation guarantee that all subsequent states are secure. These rules can be transformed into algorithms suitable for implementation on a digital computer.

Two of the basic elements of the model are subjects and objects. Subjects are active system entities such as users or processes that can access system resources, and objects are passive system entities such as data and program segments, and peripheral devices, that can be accessed by subjects. The model defines types of access that a subject may have to an object. These access types include read-only access, append access, execute access, and write/read access. For the 11/45 kernel, only read/execute access (abbreviated read access) and write/read/execute access (write access) are used in accessing segments. ~~A write only access mode is used in interprocess communication.~~

State of the System

The state of the system with respect to security is represented by four sets - b, M, f, and H [Bell]. The set b indicates the current access relationships between all subjects and objects - that is, for each subject b identifies the objects that the subject can currently access and it also indicates the permitted mode of access. Thus b is a set of triples of the form (subject identifier, object identifier, access mode).

M corresponds to an access matrix and is used by the model to implement "need-to-know" security. Elements of M are accessed by subject identifier and object identifier, and each element of M indicates in what mode, if any, the specified subject may access the specified object. Thus, M represents the potential access of subjects to objects.

The set f gives the security level of all subjects and objects in the system. A security level is composed of two parts - a classification (or clearance) and special access categories. Classifications are strictly ordered - a subject cleared to secret may access unclassified, confidential and secret objects. Categories are not strictly ordered, but are partially ordered by set inclusion. A subject with categories x and y may observe an object with category x or categories x and y, but not an object with categories y and z. The combination of strictly ordered classifications and partially ordered category sets gives a security level that is partially ordered. Thus it is meaningful to say that one security level is greater than, less than, equal to, or isolated from another security

level. (Two security levels are isolated from each other if one is neither greater than, less than, or equal to the other.)

Finally, the set M indicates how the objects are hierarchically organized in a directory-tree structure.

The system satisfies the basic military security requirements if all triples in b (subject, object, access mode) are such that the security level of the subject is greater than or equal to the security level of the object. The basic rules of the model allow for changes to b. If a subject wishes to add an object to its portion of b in some mode, it invokes the model rule that governs the particular state change. The algorithm of the rule consults f and M in determining whether or not the state change will be permitted, and adds the new triple to b, if the change is permitted. The model assumes that subjects can and will access objects as permitted by b. There are no security constraints on the removal of triples from b.

Potential Compromise

In addition to preventing explicit security compromises, the model also prevents potential security compromises. Potential compromise is a meaningful situation within a computer system but it has no analogy in the external "people/paper" system. If an individual has a secret clearance he may read documents classified secret, but he may also write documents classified confidential. By virtue of his clearance he is trusted not to include secret information in the confidential document, in the same sense that he is trusted not to disclose secret information in any other unauthorized manner. When this individual is using a computer system the situation changes, because programs that he has little knowledge of will be executing on his behalf. For example, he may invoke a compiler to translate a PL/I program into machine language. One could assume that the compiler performs the required language translation and nothing else, but in building a secure computer system we cannot assume that a program behaves properly (with respect to security requirements). Rather, unless a program is proven to behave in a certain fashion as described by a mathematical model or formal specification, we cannot make any statements about its behavior and must make the assumption that the program attempts to violate security regulations. If in fact the program does act in a malicious manner, then we say that it contains a "Trojan horse" [Branstad]. Continuing with our compiler example, in addition to doing the translation, the compiler may copy some of the invoking user's secret information into an unclassified file. At a later time, an uncleared user may read the unclassified file, thus gaining access to secret information. The compiler had access to the secret information because it was running on behalf of a user cleared to

secret. It acted the way it did because the compiler writer wanted to penetrate the system.

We can now restate the problem of potential compromise in terms of the model. We say that the potential for a security compromise exists if, for example, a subject simultaneously has read access to a secret object and write access to an unclassified object. The potential for compromise is realized if two events occur: 1) the subject, either inadvertently or deliberately, reads secret information from the secret object and writes it into the unclassified object, and 2) a second subject whose clearance level is unclassified gains access to the unclassified object and reads the secret information in it.

At least two ways of preventing this type of situation from occurring are known. The first is to upgrade the classification of the unclassified file to secret, which is known as establishing a "high water" mark [Weissman]. The second way is to deny a subject simultaneous write access to an unclassified file and read access to a secret file and to prevent similar situations from occurring. This second solution to the problem of preventing potential security compromises is defined as preserving the *-property.[3] The *-property requires that all objects to which a subject has write access have the same security level and that all objects to which it has read access have a security level less than or equal to the write security level. Since a subject will always have write access to some object if it is to perform a computation, we define the current security level to be that level at which the subject wishes to have write access. In determining whether or not to grant access to an object, need-to-know is checked by consulting M, and then the security level of the object is compared to the subject's current security level to check that both the security and *-properties are preserved.

It appears that in useful computer systems some subjects will not be able to perform as required if the *-property is applied to them. For this reason the model introduces the concept of trusted subject - trusted subjects do not have the *-property applied to them. Our trust in these subjects derives from the fact that all programs that they execute are certified to behave in a manner consistent with security requirements. Thus, if a trusted subject needs to invoke a compiler, then the entire compiler must be certified to be free of any Trojan horses. The certification requirement highlights the advantages of the *-property. Since most subjects will be untrusted there is no need to certify the programs

[3] Pronounced "star-property". The term is from [LaPadula & Bell].

they execute. This situation is indeed fortunate because certification of a complete operating system and user support programs is far beyond current software technology.

Transition Rules

The model provides rules that do things other than change the triples in b. Rules are provided for changing M and for creating and deleting objects. Since these rules give subjects a form of control over objects, the model must have a mechanism to deal with control. The mechanism used is to hierarchically organize the set of objects into a directory tree. The use of this mechanism is not arbitrary - it was chosen to allow the application of security controls to every object in a real computer system, rather than just the segments or files. Every object except for the root object has an object that is directly superior to it in the hierarchy; this object can be called the parent. The number of objects that are directly inferior to any given object is arbitrary. The model's set H describes the hierarchy of objects at any given instant. If a subject has write access to some object O, then it can create new objects inferior to O and it can change the access privileges for all objects inferior to O or delete them.

Now that the form of control has been identified, the security requirements for exercising control will be developed. Every object has security attributes. These security attributes include the security level (classification and category set), the column in M giving the access permissions, and the attribute that indicates whether or not the object exists.[4] If a subject has control over an object then it can set the value of any of its attributes. The model rules that add triples to b check these security attributes to determine if the state change requested is permitted. Since subjects can, in general, determine if the state change occurred, they have (interpretive) read access to the attributes. If one subject has write access to an object's attributes and another subject has read access, then information can be passed between subjects via these attributes. The model must insure that this passage of information does not violate security requirements.

The approach taken to this problem employs the structure of the hierarchy. The attributes of an object are objects themselves, and access is controlled to these "attribute" objects in the same manner that access to "ordinary" data objects is controlled. More

[4] The value of the security level attribute can only be set at object creation time, whereas the access permission attributes can be modified any time after the object has been created.

specifically, the attributes of an object are kept in the object's parent directory, and the security level of these attributes is the same as the security level of the directory. Thus if we have a secret object immediately inferior to a confidential object, then the confidential object is the secret object's parent directory and the security level of the secret object's attributes is confidential. The security requirements should be clear. If a rule changes (writes) the attributes of some object then the invoking subject must currently have write access to the object's parent; and if a rule reads the attributes of an object (as the rules that add triples to b do) then the subject must have read access to the object's parent. Since the *-property is applied when access to directories is requested, the reading and writing of attributes cannot be used to pass information from a high security level to a low security level. Thus the hierarchy allows the same mechanisms to protect both the information content of objects and the attributes of objects.

At the level of the model the only attributes that an object has are security attributes. In the process of designing a security kernel an interpretation of the object abstraction must be made. This interpretation will create additional attributes, and access to these new attributes cannot necessarily be controlled in the same way as access to the security attributes is. We will deal with this problem of controlling access to non-security attributes in later sections.

A few last points will be made to complete the discussion of the hierarchy. If a confidential object is inferior to a secret object then a subject cleared to confidential can never access the confidential object since the subject can never have read access to the secret object. To avoid situations like this the concept of "compatibility" is introduced. A hierarchy is compatible if security levels are non-decreasing as one moves down the hierarchy from the root, i.e., the security level of an object in the hierarchy must always be greater than or equal to the security level of its parent. Since the root has no parent its security attributes are implied and triples giving all subjects at least read access to the root must be placed in b at initialization. Whether or not a subject is given write access to the root at initialization depends on the particular system's requirements, the security level of the subject, and whether or not it is trusted.

Relation to Reference Monitor

Now that an overview of the model has been given, the relationships among the reference monitor concept, the model, the security kernel, and hardware access controls will be discussed. The reference monitor is an abstraction of the (hardware and software) mechanisms that mediate all attempts by subjects to access objects. The decision to permit or deny access is a function of the reference monitor's internal data base. If the system is to be at all dynamic, then this data base must be modifiable. The model is an interpretation of the reference monitor abstraction, and describes the behavior of a secure system in terms of a four component data base (b, M, f, and H) and rules of operation. These rules specify how the data base may be changed; they represent an "authorize" operation. It is assumed that the access relationships specified by b can and will occur - the "access" operation is implicit. Clearly the relationship between the reference monitor and the model is that: 1) the reference monitor can only allow subjects to access objects as permitted by its representation of the model's set b; and 2) the data base of the reference monitor must correspond to the model's data base and can only change as permitted by the model's rules.

The reference monitor of a physical computer system is realized by a combination of hardware and software; the portion required in software depends on the capabilities and limitations of the hardware. For example, one might expect that the hardware architecture would permit direct access to objects in all desired modes and that the hardware access controls would constrain accesses to those allowed by b. The hardware access controls' data structure (descriptors) would be a representation of b, and the kernel would simply be a realization of the model rules. In general, however, the situation is not this simple. There may be objects to which the hardware cannot properly control access, and there may be alternative representations of the same security state. Either one of these situations calls for a kernel function that does not change the security state. In the former case there would be one or more functions to permit interpretive access to an object; in the latter there would be functions for changing the representation of the security state without changing the actual state.

An example based on Multics should clarify this point. If several objects were located in the same segment, then the Multics hardware access controls, which operate on a per segment basis, could not be used to control access to these objects, for they could not permit a subject to access one object in a segment without accessing all of the objects in the segment. This situation requires functions to allow subjects interpretive access to this type of object. In addition, functions that move segments between main memory and

secondary storage do not allow objects to be accessed or change the security state, but these functions must be in the kernel because they change the representation of the security state and this representation must always be consistent with the current security state. Thus the functions of the security kernel fall into three classes that correspond to the fundamental operations of "authorize", "access", and "null": 1) functions that correspond to the rules of the model, thus changing the security state; 2) functions that implement a part of the reference monitor by allowing interpretive access to objects as permitted by the current security state, thus complementing the hardware access controls; and 3) functions that change the representation of the current security state.

HARDWARE REQUIREMENTS

In this subsection we briefly discuss hardware architecture requirements for secure systems.[5] While every computer scientist knows that any computation that can be performed can be done on a Turing machine, Turing machines are used only for pedagogical purposes. The primitiveness of Turing machines makes the accomplishment of even the simplest computations a very complex task. Thus, more advanced architectures have been developed to facilitate the use of computers for doing productive work.

In light of the above argument it should be clear that there are no absolute hardware requirements for secure computer systems - any hardware is theoretically acceptable. Given the current state of technology, however, certain hardware features are essential if we are to build secure systems. These essential features simplify the software portion of the reference monitor. Simplification of software at the expense of additional hardware is necessary because producing provably correct software is a major technical problem in computer security. There are two basic hardware features that are essential.

The first of these features is support for a segmented memory where access to segments is through unforgeable segment descriptors that include an access control field. The arguments supporting this feature as essential to security are similar to the arguments for a segmented memory given in the subsection on Multics. Segmentation allows all information in the system to be stored in one type of object - the segment. Having to support only a single object type simplifies the kernel. The descriptor driven addressing that is part

---

[5] A more complete treatment of hardware considerations is given in [Smith].

of segmentation allows the basic reference monitor function of mediating all accesses to objects to be performed by hardware, thus helping to minimize the impact of security controls on efficiency.

The other essential hardware feature is multiple execution domains (or states or modes). This feature is used in most contemporary systems to protect operating systems from applications programs. In a secure system it will be used to protect the kernel from the rest of the software in the machine. Strictly speaking only two execution domains are necessary - one for the kernel and the other for everything else - but in practice it will still be desirable to continue to protect the operating system from applications software so three domains (or more) will be useful.[6]

The kernel design presented in this report is for the PDP-11/45. The 11/45 has an optional memory management unit (MMU) that checks all references to memory and recognizes three access modes - write/read/execute access, read/execute access, and no access. The MMU is an adequate hardware base for building a segmented memory system. The MMU in the 11/45 also implements three domains of execution - kernel, supervisor, and user. Thus the PDP-11/45 can be used as the hardware base of a secure system.

Before leaving this subsection on hardware requirements it is worth mentioning that the I/O architecture can be important. I/O is the subject of the next subsection.

INPUT/OUTPUT

Input/output operations are a critical aspect of secure computer systems because they are the interface between two distinct security enforcement systems. On one side are the internal logical security controls of a computer system that associate security attributes with the information in the system, and on the other side is the external "people/paper" system that employs physical separation and document markings. Clearly, a primary requirement for I/O in a secure computer system is that the security attributes of information are correctly transferred as information moves between the internal and external environments. This subsection will briefly review some of the issues involved in secure I/O. A more complete treatment of these issues and solutions to various I/O related problems is given in [Burke] and [Mogilensky].

---

[6] Multiple domains can be implemented in software on a machine with two hardware domains. An example is the original implementation of Multics on the GE 645 [Organick].

A basic requirement for secure computer systems is that a general security marking policy for classified I/O material be established. Security markings are indications that are placed directly on, attached to, or included with classified material. The purposes of a marking policy are to satisfy the security regulations that require that all classified information have an indication of its actual classification, and to insure that the security attributes of classified data are accurately maintained for all I/O transfers. One aspect of a marking policy is a labeling policy - labels are security markings that are generated by the computer system itself (as opposed to markings that are pre-printed on forms used by the computer system). In developing a marking policy it is important to consider the difference between unilevel and multilevel I/O, whether I/O material can or cannot be removed from the computer system, and the extent to which the I/O data is human-legible.

Although a computer system may be operating in a multilevel security mode, some or all of the I/O devices may be operating in a unilevel mode. An I/O device is unilevel if it only processes information at a single security level. The level at which the device operates can be changed by a security reconfiguration. This reconfiguration can be as simple as changing the forms with pre-printed security markings that a line printer uses.

If a device can handle data at more than one security level without human intervention, then the device is operating in a multilevel mode. In this mode it will be necessary for the computer system to generate security labels.

I/O material that can be removed from the computer system includes such things as printed output and magnetic tapes; CRT (cathode ray tube display) output and data traveling between nodes in a network cannot be removed from the system. In addition, the human-legibility of I/O material can vary. Printed and CRT output is directly human legible, magnetic tape and network messages are not. In [Mogilensky] a general security marking policy is developed.

It is not sufficient simply to have a marking policy; the policy must also be effectively applied to the actual computer system. In order to satisfy security requirements for I/O, [Burke] considers three major types of I/O function: 1) authentication; 2) controlled attachment; and 3) controlled operation. Authentication establishes the identity of the user or I/O medium at the I/O device. Once authentication has been performed, the internal security controls know the security attributes of the I/O device. Attachment is the (usually software) connection of the device to some process in the computer system. Finally, controlled operation is the mechanism that

enforces the allowed attachments and insures that security labels, if they are being generated, are valid.

From a hardware point of view, we can see that most current architectures, which either give a process access to all I/O devices or to no I/O devices, make the implementation of controlled attachment and controlled operation difficult. In this environment only certified interpretive software can perform physical I/O operations.

A desirable I/O architecture is one where the hardware controls access to I/O devices on a per device basis. With this architecture controlled attachment involves changing the hardware access controls' data base. If a device is operating unilevel (meaning trusted security labels are not required), controlled operation is enforced by the hardware and uncertified software can perform the physical I/O. This mode of operation is desirable because the I/O subsystems of modern operating systems are often large and complex.

The PDP-11/45 has a desirable I/O architecture for unilevel devices that do not have direct access to memory (non-DMA devices). I/O, for these non-DMA devices, is performed by reading and writing specific main memory locations that act as device control and data registers. To the extent that these device registers can be isolated in individual segments and a set of registers controls a single device, the MMU controls access to I/O devices on a per device basis. Unfortunately, DMA (direct memory access) devices bypass the MMU when they access main memory. Thus, certified software must check the validity of any memory references a DMA device will perform before each I/O operation is initiated. On the 11/45, this checking can be relatively straightforward, because each I/O operation is individually initiated by the CPU.

SECTION III

DESIGN CONCEPT

INTRODUCTION

This section presents an overview of the design of the security kernel. The first subsection introduces the concept of levels of abstraction [Dijkstra (1)] which has heavily influenced the design. The remaining subsections present the major levels of the design.

LEVELS OF ABSTRACTION

Abstraction is a way of avoiding complexity and a mental tool by means of which a finite piece of reasoning can cover a myriad of cases [Dijkstra (2)]. The purpose of abstracting is not to be vague, but to create a semantic level in which one can be absolutely precise. Dijkstra's levels of abstraction have been demonstrated to be a powerful design methodology for complex systems, most notably Dijkstra's "THE" system and the Venus Operating System [Liskov]. In general, the use of levels of abstraction leads to a better design with greater clarity and fewer errors. A level is defined not only by the abstraction that it supports (for example, a segmented virtual memory) but also by the resources employed to realize that abstraction. Lower levels (closer to the machine) are not aware of the abstractions or resources of higher levels; higher levels may apply the resources of lower levels only by appealing to the functions of the lower levels. This pair of restrictions reduces the number of interactions among parts of a system and makes them more explicit.

Each level of abstraction creates a virtual machine environment. Programs above some level do not need to know how the virtual machine of that level is implemented. For example, if a level of abstraction creates sequential processes and multiplexes one or more hardware processors among them, then at higher levels the number of physical processors in the system is not important.

By the rules of levels of abstraction, calls to a procedure at a different level must always be made in the downward direction, and the corresponding return in the upward direction. For maximum clarity, downward calls should be to the next lower level, but there will always be cases where calls that skip over one or more levels can be justified. Returns are always to the calling program, except in the event of a severe error where several of the calling procedures may be skipped over by the return. Figure 2a shows the

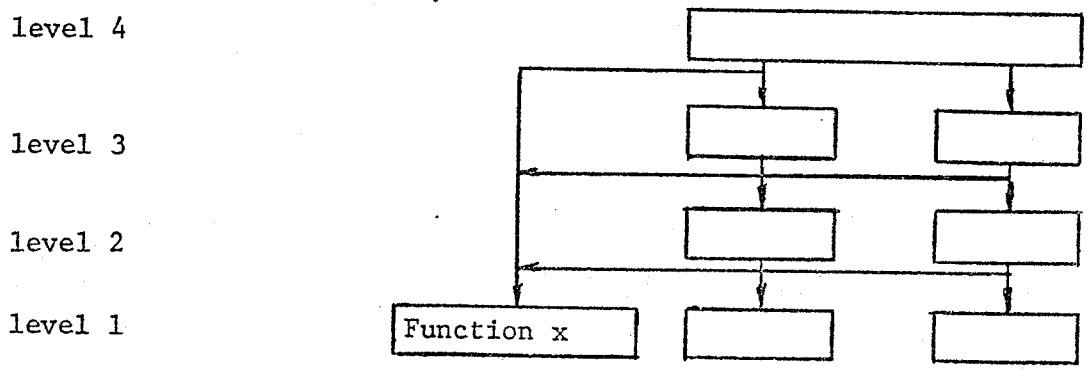


Figure 2a. Levels of Abstraction

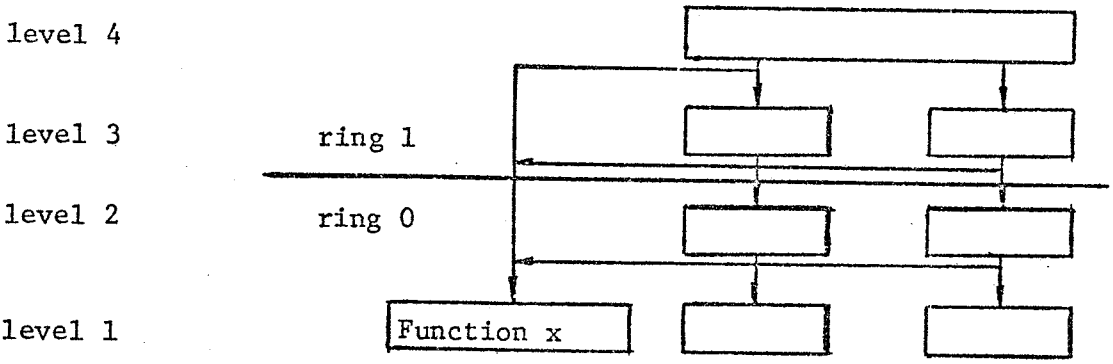


Figure 2b. Levels of Abstraction and Rings

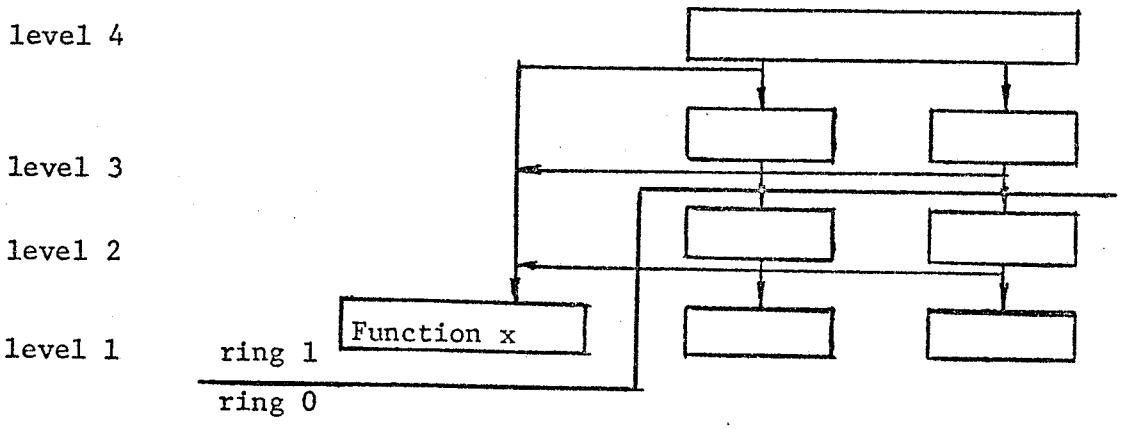


Figure 2c. Outward Ring Crossing

structure of a system where most calls are to functions of the next lower level, but the level 1 function x is called from levels 2, 3, and 4.

When a ring (hierarchical domain of execution) structure is added to the system, ~~simplicity is enhanced by having~~ each ring consist of contiguous levels. Thus the kernel, which must be the innermost ring (ring 0), should consist of the level of abstraction that implements the reference monitor concept and the supporting levels beneath that level. In our example system, the boundary between ring 0 and ring 1 may come between level 2 and level 3 as shown in Figure 2b. Following the policy of making a ring consist of contiguous levels, all cross-ring calls are automatically to an inner ring and this is the type of ring crossing call that is supported by the PDP-11/45 hardware.

It is possible, however, that function x has no security implications, so it can be removed from the kernel as shown in Figure 2c. Now, however, calls by level 2 functions to function x (level 1) are an outward ring crossing. Unfortunately, this type of a procedure call is not supported by the 11/45 hardware (or other computers with a hardware ring mechanism), so if it is to be used at all, it must be implemented with certified software. A case where this calling structure might occur is with the scheduler of a multiprogramming system. The scheduler may appear at a low level of abstraction, but if we make a distinction between the scheduler - code that implements the policy that selects the next process to run - and the process multiplexor - code that implements the mechanism that binds a process to the hardware, - then it can probably be proved that the correctness of the scheduler is not necessary for security. Thus, we would want to remove it from the kernel, in spite of the fact that it may be called from the kernel.

This example illustrates an apparent conflict between the goals of overall system clarity and a small and simple kernel. One could argue that one of these goals, or the use of levels of abstraction with its requirement of strict hierarchical layering, or the use of protection rings causes the conflict. A machine that provided the more general form of non-hierarchical protection domains would solve this problem by allowing an internal partitioning of the kernel. Domain machines, however, are not currently available. Since we are forced to use a ring machine and we believe that the levels of abstraction design methodology will facilitate certification of the kernel, our only choice is to compromise one or both of the design goals of overall system clarity and a small simple kernel. This issue will be discussed further as design details are presented.

LEVELS OF THE KERNEL

In designing the security kernel, levels of abstraction have been used in the translation of the abstract elements of the mathematical model to tangible elements of a secure computer system. The first steps taken were to make an interpretation of the model elements (i.e., objects are virtual memory segments and subjects are sequential processes) and to provide at some level of abstraction a set of functions that controls access to these elements. Thus the abstraction created by this level is that of a secure computer system. It must be emphasized that what this secure system level of abstraction does is to effect the implementation of the reference monitor, thus insuring that the system is always in a secure state.

The specific design structure chosen for the actual implementation of the interpreted elements is done by lower levels of abstraction, as shown by Figure 3. While the software at these levels is not cognizant of specific security requirements, it is part of the kernel because the correct operation of the secure system level functions depends upon the correctness of lower levels. We choose to place the segmented virtual memory level above the process level because segments can be shared by processes, and because we want to be able to start a new process running when the current process must wait for a segment to be swapped into main memory. The PDP-11/45 hardware provides a form of main memory segmentation that is used in the implementation of the process abstraction.

It should be clear that the boundary of the kernel belongs immediately above the secure system level of abstraction. Software outside of this perimeter can execute the unprivileged hardware instructions and invoke the functions provided by the secure system level with arbitrary arguments. Since the unprivileged machine instructions cannot put the system into an unsecure state and the secure system functions make no assumptions about the legality of arguments passed to them, the security of the system is independent of what the software above the secure system level of abstraction does or does not do. Thus the implementation of the security level of abstraction and the implementation of the lower, supporting levels, gives us a complete security kernel.

While the initial presentation of the kernel's levels of abstraction will be made from the bottom up, it should not be inferred that the kernel was designed this way. Rather, the design was constrained at the top by the mathematical model and at the bottom by the characteristics of the PDP-11/45. The bottom hardware constraint was somewhat more rigid than the model constraint because the abstract elements can be interpreted in a variety of ways. Since all design decisions must be in harmony with both constraints, the

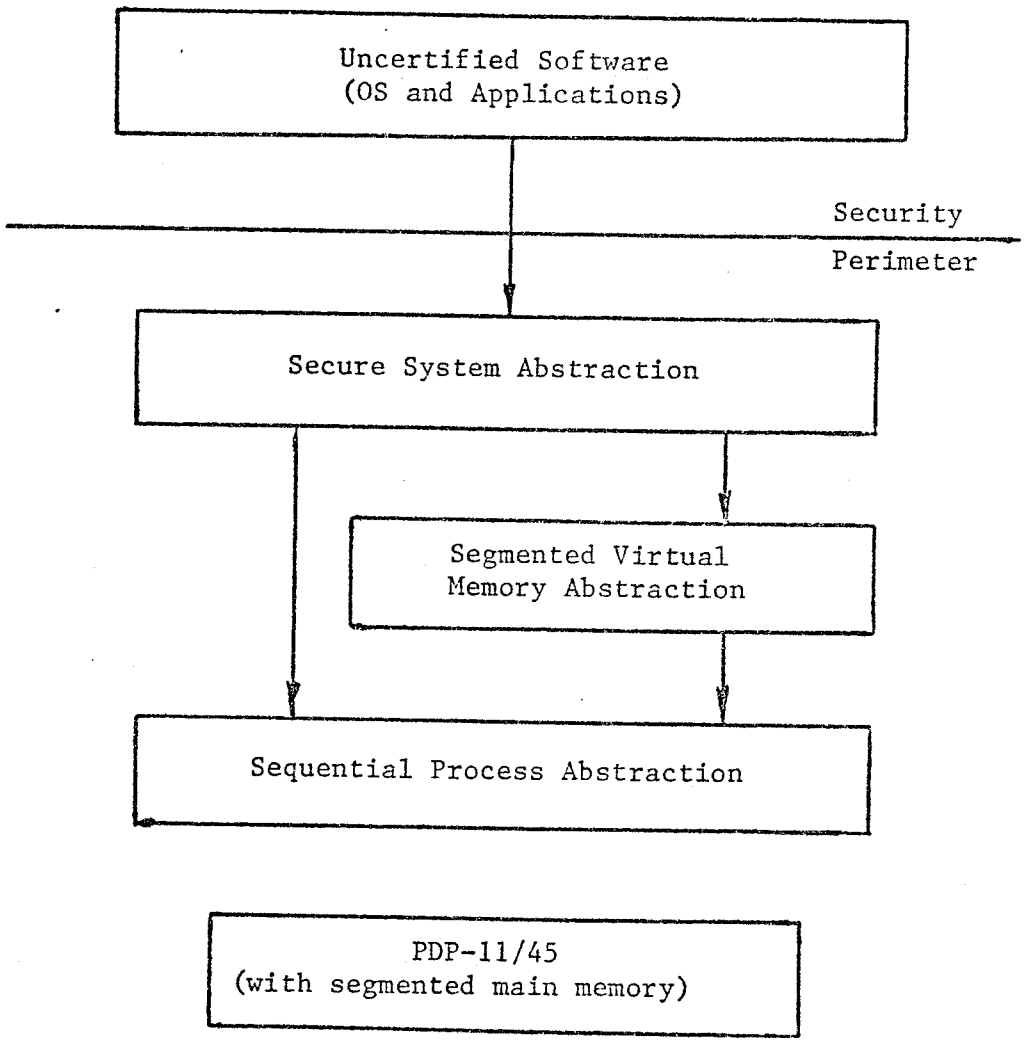


Figure 3. Kernel Structure

design technique was "middle out" - up towards the model and down towards the 11/45. The choice of a bottom up presentation is somewhat arbitrary and was made because we feel it is easier to build on concepts that are well understood.

LEVEL 0 - THE HARDWARE

The PDP-11 is an advanced family of 16 bit mini to medium sized computers with a powerful instruction set, hardware-managed stacks that facilitate procedure nesting and the coding of reentrant procedures, and a set of general purpose registers that can be used as accumulators and/or index registers. An optional feature of the PDP-11/45 that makes it a suitable base for a secure system is the memory management unit (MMU) and its associated three domains of execution - kernel, supervisor, and user. Although the memory management unit is described as a general purpose memory management device [Digital] and one might hope to implement a Multics-like two-dimensional virtual memory with demand paging, it appears that the MMU is most reasonably used to divide main memory into logical address spaces with associated access controls.

The key to understanding the MMU is the dynamic address translation process it performs (illustrated in Figure 4). Every time an effective address is generated during instruction execution, it is treated as a 16 bit virtual address and translated to an 18 bit physical address before the reference to main memory is made. The translation is controlled by the contents of a set of eight segmentation registers. Each segmentation register specifies the base and limit addresses for an area of main memory, and access control information. Recognized modes of access we will initially use include null access, read access, and write access. A 16 bit effective address is treated as a two dimensional virtual address by having the high order 3 bits select one of the eight segmentation registers and the remaining 13 bits be a displacement into the area of main memory addressed by the selected segmentation register. The MMU acts as a hardware reference monitor and generates a fault when the displacement is too large or access is attempted in a mode that is not permitted.

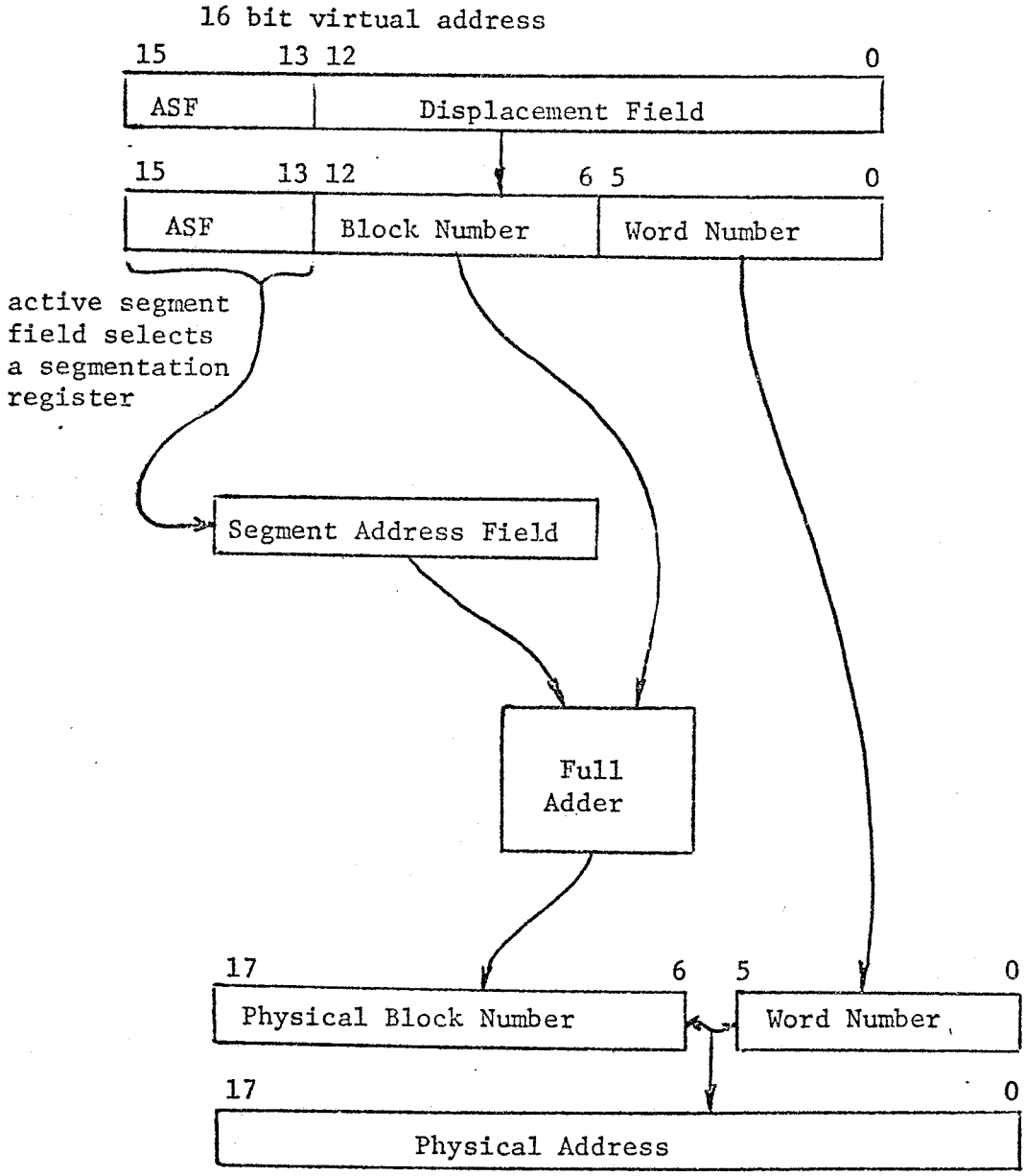


Figure 4. Dynamic Address Translation

In the PDP-11/45, there are three sets of segmentation registers, one for each domain of execution.[7] The current domain of execution has associated with it one of the three sets of segmentation registers which is selected before the remaining part of the dynamic address translation process occurs. Provision for multiple register sets is part of the hardware implementation of multiple execution domains and allows a process to be given a different address space and/or the same address space with different access rights depending on the current domain of execution.

Several characteristics of the MMU have a limiting effect on the kernel's functionality. Since there is only one level of address translation (the segmentation registers are addressed directly and then one add operation is performed), the MMU can be used to provide a segmented memory or a paged memory but not both. Because segmentation is vital to our design of a secure system we must have non-paged segments.

In Multics, the occurrence of a fault while referencing a virtual memory segment can signal one of at least two different cases - an attempt was made to access a segment (or that part of a segment) that was not in main memory or an attempt was made to access a segment in a prohibited manner. In the former case the hardware provides the Multics supervisor with sufficient information to allow corrective action and successful re-execution of the faulting instruction. The latter case indicates an access violation and is handled appropriately. The PDP-11/45's MMU provides only limited information when a fault occurs. This lack of information makes it difficult to distinguish between the missing segment/page fault and access violation cases, and also to resolve missing segment/page faults with "small and simple" software. The impact of this MMU characteristic on the design will be dealt with in the section that discusses the segmented virtual memory level of abstraction.

---

[7] Actually, the MMU has two sets of segmentation registers for each domain of execution. The hardware uses the Instruction (I) Space registers for all memory references that involve instruction fetches, index words, absolute addresses, and immediate operands. The Data (D) Space registers are used for all other references. Language Processors must be aware of the difference between I and D Space and generate code appropriately - program constants that are not immediate operands cannot be in the same segment with program code. Since the language processors used in the initial implementation of the 11/45 kernel are not aware of I and D Space, the D Space segmentation registers are disabled and all address translations use I Space registers.

One other aspect of the MMU is worth mentioning at this time. Since the segmentation registers are directly addressed and not accessed indirectly via a base register, the processor state consists of a large number of registers. Saving and restoring the processor state is quite time consuming because all of the registers must be saved/restored one at a time - the PDP-11/45 has no block move instruction. While performance characteristics are an important but secondary consideration in this prototype development, the cost of context switching could have a severe impact on a secure production system built on the 11/45.

The abstraction at level 0 is a processor with a segmented main memory. The basic resource used in creating this abstraction is the MMU hardware, but there is also some software in the implementation. There is a table that indicates how main memory is segmented and a function that uses information in this table to construct and load segment descriptors.

LEVEL 1 - SEQUENTIAL PROCESSES

Level 1 creates the process abstraction. We use the "standard" (and somewhat vague) definition of process - a process is a procedure in execution. The design supports a fixed number of processes; each process runs on a virtual machine and consists of an address space and control information about the process. At level 1 it is sufficient to know that the address space is defined by the control information, part of which is the contents of the segmentation registers. Level 1 software has the responsibility for allocating the processor to one of the processes whose dynamic progress is permissible.

At a given time, a process is in one of several possible execution states [Saltzer (1)]. Figure 5 shows the relationships among the various execution states and the actions that move a process from one state to another. In the inactive state a process does not have an address space and cannot run. A process can only be moved out of (and also into) the inactive state by a special executive process that is never in the inactive state. At the time that it moves a process out of the inactive state, the executive must establish the initial address space of the process. The purpose of the inactive state is to create a mechanism for minimizing the resources required to support a process that is not currently needed (perhaps because less than the maximum number of users are currently signed onto the system), but is one of a fixed number of processes defined by the implementation.

An active process is either blocked or unblocked. In the blocked state, a process is waiting for the occurrence of some event.

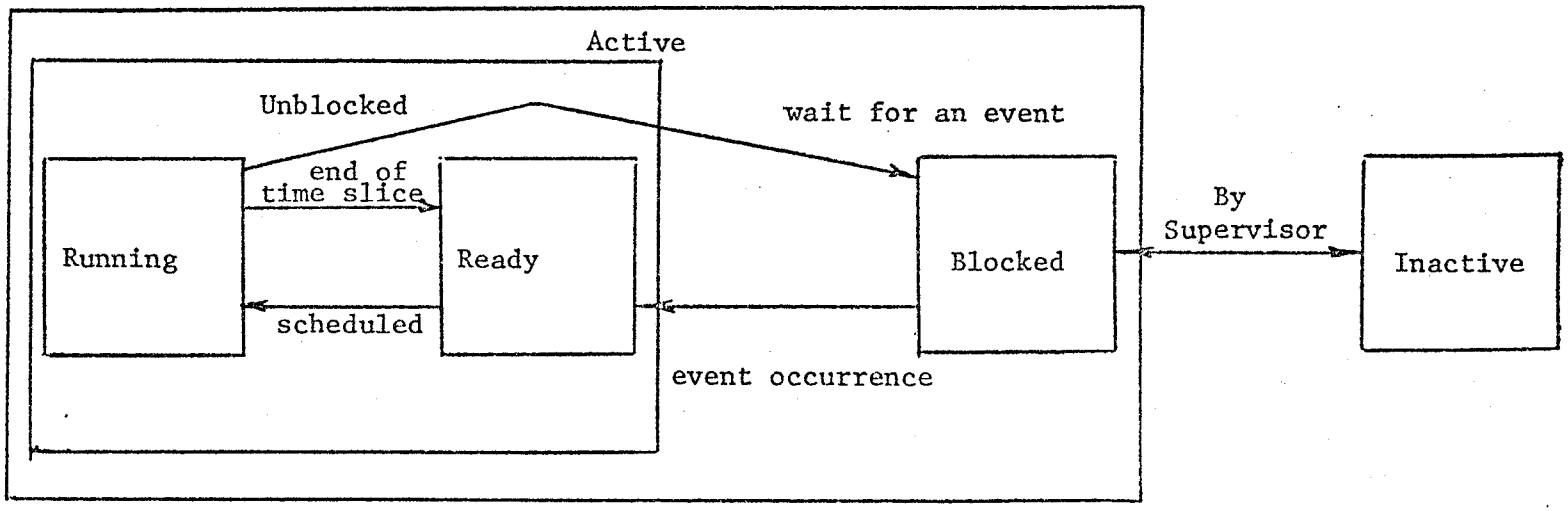


Figure 5. Process Execution States

An unblocked process is either in the running state or ready state. The running state simply signifies that the process has the CPU allocated to it. Above level 1 the running and ready states are logically equivalent. In the ready state a process is ready to run (its dynamic progress is permissible) but must wait for the CPU to be allocated to it. Processes enter the ready state from the blocked state when the event for which they were waiting occurs. Transition of processes between the ready and running states is controlled by a simple scheduling policy internal to level 1.

The hardware resources of this level are the CPU and a real-time clock. A data base is employed to contain state information about the processes and to help manage them. This state information includes a definition of each process's address space, an indication of its execution state, and a specification of the user associated with the process and his security attributes. No interpretation of these security attributes is made at this level (the operation of level 1 is independent of their value), rather, space is set aside in level 1's data base for security attributes as a convenience for higher levels.

Several different types of functions are provided by level 1. Two sets of functions are provided for the synchronization of processes - Dijkstra's P and V functions (explained in Appendix I) and message send and receive for interprocess communication. In the level 2 subsection we will explain how P and V are used to handle I/O interrupts. More detail on these synchronization functions and the rationale for providing two sets of functions where one might suffice is given in the next section.

Level 1 implements a simple scheduling policy - the highest priority process that is ready to run has the CPU allocated to it. To allow a more sophisticated scheduling policy to be implemented outside of the kernel a function can be provided to dynamically change process priorities. A discussion of the issues involved is provided in the next section.

Finally, as online users log on and off the system (and as batch jobs are initialed and terminated) it is necessary to provide them with processes and then to terminate these processes. Two functions - activate process and deactivate process - are provided for this purpose.

Level 1 creates a multiprogramming environment which effectively implements the co-existing subjects that are a major element of the model.

LEVEL 2 - SEGMENTED VIRTUAL MEMORY

The second level of abstraction creates a segmented virtual memory, building on the segmented main memory provided by level 0 (the hardware). Segments are the primary storage entities of the system and will be the basic object to which access is controlled by the security level of abstraction. As mentioned in the subsection on level 0, the characteristics of the MMU limit the flexibility of the segment abstraction created at this level. In particular, one would like (as Multics does) to implement variable sized segments consisting of fixed sized pages. The use of paging facilitates the dynamic growth of segments, permits only part of a segment to be swapped into main memory, and vastly simplifies the allocation of both primary and secondary memory. Unfortunately, the characteristics of the MMU force us to implement unpaged segments. To simplify the design we have implemented fixed sized segments - that is, when a segment is created a permanent size for it is specified and resources are allocated appropriately. If only a single size were provided the resulting system would be difficult to use, especially when the small number of segmentation registers (8 per domain) is considered. A small segment size would severely limit the amount of virtual memory that a process could directly address at any given instant; a large segment size would tend to waste space. Thus the design provides for fixed sized segments available in a variety of sizes. The actual number of sizes will be an implementation decision subject to hardware constraints - sizes must be multiples of 64 bytes up to 8K bytes maximum. Since even the largest segment may be small for some applications, we anticipate the creation of a file abstraction above the kernel. This abstraction will allow several segments to be treated as a single entity and permit subsections of a file (segments) to be individually swapped in and out of main memory.

The lack of adequate hardware to support a demand paging/segmentation environment further affects the design at this level. The virtual memory provided by this level can best be described as a "non-random access" virtual memory. Users of this level cannot arbitarily access segments in their address space without first indicating an intention to perform this access. This intention is indicated by asking level 2 to load a segmentation register with a descriptor for a segment before any instructions referencing that segment are executed. It is sufficient for level 2 to "lock" all segments for which descriptors exist into main memory to guarantee that missing segment faults do not occur. Level 2 considers any fault generated by the MMU to be an access violation and simply passes the fault on upward; it does not attempt to take any corrective action. The implementation of level 2 corresponds to swapping or overlaying, rather than demand paging. It should be

noted that level 2 still implements a one level virtual memory because: 1) segments are the only type of storage entity, and 2) as different segment descriptors are loaded into the segmentation registers the address space of a process can be greater than the size of main memory.

Segments have attributes - information that describes the characteristics of a segment. From the mathematical model we know that at level 3 segments will have security attributes; at level 2 they have implementation attributes. Implementation attributes include a segment's size and disk address. The attributes of a segment are contained in an entry in the segment's parent directory. At level 2 space is provided in directory entries for security attributes but the operation of this level is independent of the values of security attributes. Since directories are themselves segments with attributes residing in other directories, the total structure is a directory hierarchy in the form of a tree. The attributes of the root segment of this tree are fixed by the design and implementation.

All segments in the hierarchy are ~~either directory segments or~~ data segments. (Segments containing executable code are considered data segments by level 2.) Although level 2 does not enforce access control to segments in general, it cannot permit software above it to write directly into directory segments, because the correct operation of level 2 requires the integrity of the (implementation) attributes of segments. Functions at this level provide users with an interpretive directory write capability. The security requirements enforced at the security level will further restrict access to directories because of the nature of some of the segment attributes. This point is discussed in the Data Structures subsection of the next section.

As previously mentioned, some locations in the PDP-11/45's main memory are I/O device control registers. The main memory segments that "cover" these locations are permanently bound to data segments in the virtual memory. Thus, the ability to use an I/O device is controlled by the ability to access the associated segment.

To facilitate segment sharing, level 2 associates a semaphore with each segment and requires write access to the segment in order to P and V on the semaphore. The I/O segment semaphores have a special use - the kernel translates I/O interrupts into V's on the appropriate semaphores. Thus, when a process wishes to wait for an interrupt from an I/O device, it P's on the I/O segment semaphore, (presumably) blocking itself. When the interrupt occurs, a V is performed and the process becomes unblocked. The kernel is only concerned with controlling access to the I/O segments and semaphores,

not with the correct use necessary to assure proper synchronization.

Level 2 implements a segmented virtual memory by building upon level 0's segmented main memory, using secondary storage devices for segment swapping, and employing a data base to indicate the state of the virtual memory. The data base consists of the directory segments, tables for managing the allocation of secondary storage, and the Active Segment Table (AST). The AST is a mechanism that facilitates the sharing of segments in main memory - if two different processes wish to access a segment they both access the same physical segment and not two different copies. Any segment that is in the address space of one or more processes or is "wired down" (permanently swapped into main memory) is active - it has an entry in the AST. An active segment table entry (ASTE) contains the segment's permanent attributes - copied from the directory - as well as additional attributes associated with the fact that the segment is active. These additional attributes include a list of the processes that have the segment in their address space and the main memory address of the segment if it is currently swapped in.

Segments in the hierarchy can be uniquely identified in a variety of ways. If a segment is active, identifying its entry in the AST (aste#) specifies the segment. If a segment is not active but its parent is, then the aste# of the parent directory and the identification of the entry within the directory that contains the segment's attributes (aste#, entry#) specifies the segment. A generalization of the (aste#, entry#) identification method is the complete pathname - a specification of all directory entries, beginning with the root, that identify the segment. Finally, each segment has a unique identifer - its disk address. Within the security kernel the primary segment identification techniques are the aste# and the (aste#, entry#).

Level 2 provides functions for creating and deleting segments, adding and removing segments from a process's address space, and creating and destroying segment descriptors. The segments created at this level are the basic interpretation of the objects of the mathematical model. Although segment descriptors permit access control to segments, the only access control policy enforced at this level is the requirement for interpretive directory writes.

LEVEL 3 - SECURITY

The software above level 2 sees a virtual machine with a segmented virtual memory (that provides for access control if desired) and the multiprogramming of sequential processes. Thus, the major elements of the mathematical model of secure computer systems (subjects, objects and access control) have been realized. Given the environment assumed by the model, the implementation of security by following the rules of the model is straightforward.

Unlike the lower levels, level 3 has no hardware resources or data bases of its own. Level 3 makes a correspondence between the subjects and objects of the model and the abstractions implemented by levels 1 and 2, associates security attributes with these lower level abstractions, and controls access to the lower level functions that operate on these abstractions based on the rules of the model. Each model rule has two parts - the first part consists of security checks to determine if the requested state change can be permitted; if it can, the second part of the rule indicates how the state change is to be made. In the kernel level 3 functions perform security checking and then direct levels 1 and 2 to perform state changes if security requirements are satisfied.

As previously mentioned, the kernel uses processes as the basic interpretation of subjects, and segments as the basic interpretation of objects. In addition, semaphores and interprocess communication messages are also objects. Rather than using its own data structures for representing the model's data base (b, M, f, H), level 3 uses the data structures of levels 1 and 2 for associating security attributes with processes and segments. In addition to holding a segment's implementation attributes, directory entries contain the segment's security level - half of f - and access control lists - M. The (aste#, entry#) method of identifying segments is a representation of H. The data structures used by level 1 to support processes include a specification of each process's current address space (the segments that a process can currently access and the permitted modes of access) - the model's b - and an identification of the user associated with the process together with the user's security attributes - the other half of f.

Now that the model has been implemented all remaining software in the system can be uncertified - contain bugs or malicious penetration attempts - without a threat of security compromise if two conditions are satisfied. The kernel must be protected and access to its functions controlled. These conditions are met by preventing uncertified software from gaining write access to the kernel segments and by having only the kernel execute in kernel domain.

In the next section we will repeat the process of describing the kernel, but this time more details and motivation will be provided.

SECTION IV

DESIGN DETAILS

INTRODUCTION

In this section the design is again presented, but more detail is given. The first subsection discusses the uncertified software environment, the second describes the data structures used by the kernel, and the remaining subsections give a formal specification of the kernel functions.

UNCERTIFIED SOFTWARE ENVIRONMENT

The kernel software and PDP-11/45 (with Memory Management Unit) create a virtual machine environment for processes consisting of uncertified programs. The virtual machine is similar to a real PDP-11 (not 11/45) in that it has the general purpose registers and instruction set of the PDP-11. The virtual machine has, however, a much different memory structure - a non-random access segmented virtual memory that is shared with other virtual machines. The kernel provides the virtual machine with functions for operating on the virtual memory, and for communicating and synchronizing with other virtual machines. Programs executing in the virtual machine can execute any unprivileged PDP-11/45 machine instruction or invoke any kernel function, although in either case the desired operation can be aborted (by the MMU or the kernel) to prevent a security compromise from occurring.

The segmented virtual memory is organized into a tree-like directory hierarchy. We can think of the set of all segments in the hierarchy as the system space (SS). By virtue of security attributes and a security policy, most users of the system will not be able to access all of the segments in SS. The subset that a user may access will be called a virtual space (VS). When a user logs onto the system he has a virtual machine (identically, a process) execute on his behalf. This process will have an address space of segments constrained in size by certain design and implementation parameters. We call the process's address space the working space (WS), and it is always a subset of the user's VS. (WS corresponds to the model's b. Ideally, we would like to permit a process to directly access all segments in its WS, but because of the small number of descriptors provided by the 11/45 (eight per domain), this approach would severely constrain the size of WS. It would probably be necessary for a process frequently to move a segment out of its WS to make room for a new segment, and then shortly thereafter move the old segment

back into the WS. Because of the security checking involved - moving a VS segment into WS changes b - this approach could add considerable overhead to the computation being performed. To avoid this problem it was decided not to limit the size of WS by the number of descriptors available and to add another space - access space (AS) - that represents the segments that a process can directly address, because it has descriptors for them. AS is, of course, constrained by the number of hardware descriptors available, and it is a subset of WS. Now a process will remove a segment from AS to make room for another, rather than removing a segment from WS. The justification for this approach is that the cost of moving a segment into AS is less than the cost of moving a segment into a combined WS/AS, because changing AS does not change the security state. Figure 6 shows the relationships among SS, VS, WS, and AS.

The burden of managing WS and AS, and dealing with constraints, both security and implementation, imposed by the kernel, falls clearly on uncertified software. This does not mean that a user writing applications software must be familiar with all of the kernel's idiosyncrosies, for one of the functions of an operating system might be to make the environment created by the kernel more palatable to the user. Before presenting the specification of the kernel functions, the data structures employed by the kernel will be described.

KERNEL DATA STRUCTURES

In discussing the data structures of the kernel we have a chicken and the egg problem - understanding the design of the data structures requires understanding the functions that use them and vice versa. We choose to deal with the data structures first because their description is more compact than the functions' description. We will start with the structures used to implement the segmented virtual memory, then discuss the process structures, and conclude with the main memory structures.

Directories

A directory is a segment that consists of entries. Each directory entry is either unused or contains the attributes of some other segment. A directory entry (see Table I - the numbers in parenthesis after each field name are the size of the field in bits for the initial kernel implementation) has a fixed part and a variable part - field names for the fixed part begin with "DIR_", for the variable part with "ACL_".

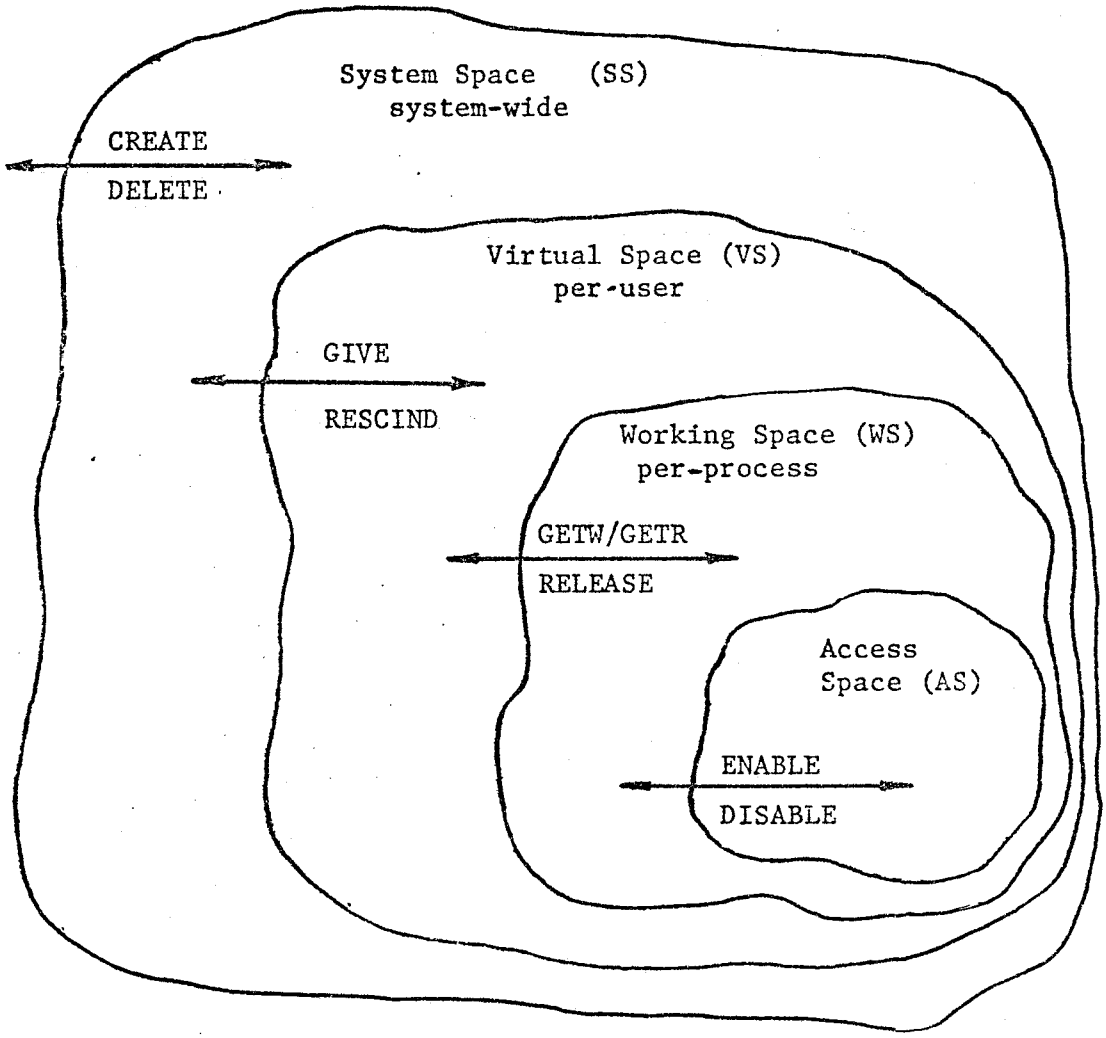


Figure 6. Spaces

Table I

Format of a Directory entry (fixed part)

and an Access Control List (ACL) element (variable part)

a Directory entry is accessed by (aste#, entry#):
DIR_XXX(aste#, entry#)

DIR_TYPE(1)	DIRECTORY or DATA
DIR_STATUS(1)	UNINITIALIZED or INITIALIZED
DIR_CLASS(4)	classification
DIR_CAT(16)	category set
DIR_SIZE(8)	size in blocks
DIR_DISK(24)	disk address of the segment
DIR_ACL_HEAD(8)	head of the ACL (or 0 if list is empty)

an ACL element is accessed by (aste#, acle#): ACL_XXX(aste#, acle#)

ACL_USER(14)	user-id or ALL_USERS
ACL_PROJECT(8)	project-id or ALL_PROJECTS
ACL_MODE(2)	mode of access - WRITE, READ, or NO access
ACL_CHAIN(8)	acle# of next ACL in the chain or 0

the head of the free ACL element chain is accessed
by ACL_CHAIN(aste#, 0)

The field DIR_TYPE specifies the type attribute of the segment. Its value is either DIRECTORY or DATA. All segments are initialized before they are first accessed. Data segments are initialized to all zeros, and the initialization of directory segments will be explained later. The attribute DIR_STATUS indicates whether or not a segment has been initialized. Its value is either UNINITIALIZED or INITIALIZED. DIR_CLASS is part of the security level attribute - the classification. DIR_CAT is the rest of the security level attribute - the category set. DIR_SIZE is the size of the segment. The MMU requires all segments to be a multiple of 64 bytes in size, but in the initial implementation the size of a segment is a multiple of 256 bytes. If the value of DIR_SIZE is zero, the directory entry is not being used and the values of all other fields are undefined. DIR_DISK is the disk address of the segment. DIR_ACL_HEAD is the head of the chain of ACL (access control list) elements for the segment - the ACL is the variable part of a directory entry. If there are no ACL elements then DIR_ACL_HEAD is zero.

The access control list is an open-ended list of names of users permitted to access a segment - it corresponds to a column of the matrix M and implements need-to-know protection.[8] Users are identified with a two part name. The first part (user-id) uniquely identifies each user. The second part (project-id) partitions users into groups called projects. The use of a two part name facilitates granting access to groups of users when all of the members are not known or the membership is dynamically varying.

Whenever a user is on the system the state information of his process includes his user-id and the project-id of the project he is currently working under. (A user may be allowed to log onto the system under one of several different projects.) Similarly, an ACL element includes a two part name but either part may be replaced by a special flag that indicates "don't care". The "don't care" flag is represented by the id ALL_USERS or ALL_PROJECTS. Thus the ACL elements (SMITH, DMS), (SMITH, ALL_PROJECTS), (ALL_USERS, DMS), and (ALL_USERS, ALL_PROJECTS), respectively, permit the user named SMITH to access the segment when he is working under the DMS project, permit SMITH access independent of the project he is working under, permit access to all members of the DMS project, and finally permit access to all users of the system.

In addition to a name, each ACL element has a permitted mode of access - no access, read access, or write access. Associating the access mode with the ACL element rather than the segment itself

---

[8] The 11/45 ACL mechanism is quite similar to the Multics ACL mechanism as described in [Saltzer (2)].

allows different users to have different access rights. The use of the "don't care" flag makes it possible for more than one element in an ACL chain to apply to a user - in this case the first element in the chain that applies determines the permitted access mode. ACL elements are always ordered from most specific to least specific, thus elements with a specific user-id and project-id come first, an (ALL_USERS, ALL_PROJECTS) element can only be last, and elements with a specific user and ALL_PROJECTS come before an ALL_USERS, specific project element. Thus the following chain is possible: (SMITH, DMS, NO), (JONES, DMS, WRITE), (ALL_USERS, DMS, READ). It indicates that all members of the DMS project have read/execute access to the segment except for SMITH who has no access and JONES who has write/read/execute access.

We can now define the ACL element fields. ACL_MODE is the mode of access associated with the element, ACL_USER is the user-id or ALL_USERS, ACL_PROJECT is the project-id or ALL_PROJECTS, and ACL_CHAIN is the link to the next element in the chain or zero if this is the end of the chain. In the initial implementation a directory segment has 63 usable entries (numbered 1 to 63) plus a header entry (entry# 0) and 127 ACL elements that are shared among all entries. The sharing mechanism employs a chain of free ACL elements - the head of this free chain is ACL_CHAIN(0). Thus a directory is initialized by marking all its entries as free and placing all the ACL elements on the free chain.

All segment attributes except for DIR_STATUS and DIR_DISK are specified by users with write access to the directory and therefore have the security level of the parent directory, but the values of DIR_STATUS and DIR_DISK are a function of system-wide activity. In the case of DIR_STATUS it can be changed to INITIALIZED by any process that has access to the segment, so this attribute must have a security level of "system high".[9] A complete explanation of the nature of the DIR_DISK attribute is postponed until we discuss the functions that create and delete segments, but the point is that our view of directory segments must be modified. Directories will be considered to be "composite" objects. Most of the data in a directory will be at the security level of the directory but some will be at a higher level. The format of the directory is defined within the security perimeter so there is no problem in determining the security level of a particular data item. Since the segment is

---

[9] "System high" is a security level that consists of the highest classification in the system and the union of all special access categories. Thus, with respect to the security condition and the *-property constraint, a system high subject may gain read access to all information in a system (subject only to the access matrix M).

the smallest object to which access is controlled by the MMU, we cannot permit uncertified software direct read access to directory segments. Thus if uncertified software is to have read access to a directory it must be via kernel functions that do the reading interpretively and are cognizant of the nature of directories.

Active Segment Table

The Active Segment Table (AST, see Table II) is a system-wide table that facilitates the main memory sharing of segments among processes. Every segment that is in the working space (WS) of one or more processes or is wired down has an entry in the AST - the segment can be identified by its aste# (AST entry#). Like a directory entry, an ASTE is composed of a number of fields. AST_TYPE, AST_STATUS, AST_CLASS, AST_CAT, AST_SIZE, and AST_DISK correspond to the similarly named fields in a directory entry. At the time that a segment is activated these fields in the ASTE are set by copying from the directory entry. Since an active segment may be in the WS or more than one process, we may want to know which processes have it in their WS. AST_CPL (connected process list) tells us this (read access is implied) and AST_WAL (write access list) indicates which processes have write access as well. In the initial implementation AST_CPL and AST_WAL are bit maps - if bit n of AST_CPL is 1 then process# n is on the CPL. Bit 0 of AST_CPL indicates whether or not the segment is wired down.

When a process removes a segment from its WS, AST_CPL may become zero. This event means that the segment can be deactivated, making the ASTE free. Rather than deactivate as soon as possible, we choose to deactivate as late as possible - when we need the ASTE to activate another segment. Segments that can be deactivated (as indicated by a zero AST_CPL) are kept on a chain running through AST_AGE_CHAIN. AST_AGE indicates whether or not a segment is on the age chain. The rationale for this delayed deactivation is discussed when the functions that move segments into and out of a process's WS are described.

A process can ask for a descriptor for a segment in its WS - thus moving the segment into its access space (AS) and locking the segment into main memory. AST_ADR is the main memory address of the segment if it is swapped in, AST_ADR will be zero if the segment is swapped out.[10] Since the (beginning) main memory address of a segment will always be on a 256 byte boundary, AST_ADR need not

[10]The main memory beginning at address zero is used for the internal kernel data base, and thus zero is never a legal address for a user segment.

Table II

Format of an Active Segment Table (AST) entry

an AST entry is accessed by aste#: AST_XXX(aste#)

AST_TYPE(1)	DIRECTORY or DATA
AST_STATUS(1)	UNINITIALIZED or INITIALIZED
AST_CLASS(4)	classification
AST_CAT(16)	category set
AST_SIZE(8)	size in blocks
AST_DISK(24)	disk address
AST_CPL(16)	connected process list
AST_WAL(16)	write access list
AST_AGE_CHAIN(16)	chain of segments eligible for deactivation
AST_AGE(1)	UNAGED - segment is not on the age chain
	AGED - segment is on the age chain
AST_ADR(16)	main memory address of segment
AST_DES_COUNT(16)	number of descriptors for segment
AST_SWAP_CHAIN(16)	chain of segments eligible to be swapped out
AST_LOCK(1)	LOCKED - segment is not on the swap chain
	UNLOCKED - segment is on the swap chain
AST_CHAIN(16)	used by HASH function and for free ASTE chain

the head of chains are accessed by AST_XXX(0)

Table III

Format of the Process Table (PT)

the PT is accessed by process#: PT_XXX(process#)

PT_FLAGS(2)	READY, BLOCKED, or INACTIVE
PT_LINK(6)	chain of processes blocked on a semaphore
PT_PS_ADR(16)	main memory address (block#) of the PS
PT_IPC_QUEUE_HEAD(8)	head of the IPC queue

include the low order (all zero) 8 bits of the address. It is not always sufficient to know that a segment is in main memory, as there are times when the number of descriptors that exist for a segment must be known - AST_DES_COUNT (descriptor count) tells us this information.

When a process removes a segment from its AS, AST_DES_COUNT may go to zero. This event means that the segment has become unlocked and can be removed from main memory. As with the deactivation case, we choose to postpone this removal as long as possible. Active segments that are eligible to be swapped out are kept on a chain running through the AST_SWAP_CHAIN field. AST_LOCK indicates whether or not a segment is on the swap chain.

The last field in the AST is AST_CHAIN. There is a function whose input is the disk address of a segment and whose output is the aste# of the segment if it is active or zero otherwise. This function (HASH) uses the AST_CHAIN field. This field is also used to chain together ASTE's that are free.

The initial implementation provides 256 ASTE's numbered 0 to 255. aste# 0 is a header - the AST_AGE_CHAIN, AST_SWAP_CHAIN, and AST_CHAIN (for free ASTE's) chains begin in aste# 0.

Process Table

The Process Table (PT) is one of the two basic data structures used by level 1 in creating the process abstraction. The PT has an entry for each process, and each entry consists of several fields (see Table III).

PT_FLAGS indicates the execution state of a process - its value is READY, BLOCKED, or INACTIVE. When several processes are blocked on the same semaphore, the processes are chained together through the PT_LINK field. PT_PS_ADR is the address of a main memory segment (the process segment) that contains additional information about the process. It will be described shortly. PT_IPC_QUEUE_HEAD is the beginning of a chain of interprocess communication messages sent to the process. Its value can indicate one of three possible states: 1) there are messages that have been sent and not yet read by the process, 2) there are no messages that have been sent to the process and not yet read, and 3) the process has become blocked because it wants to read another message and none is available.

In the initial implementation the PT will also have an area for saving hardware registers relevant to execution in the kernel domain

when the process does not have the processor allocated to it. Since this area is not relevant to subsequent design details it is not shown in Table III.

Process Segments

The second basic data structure used by level 1 is the Process Segment (PS) - there is a process segment (main, not virtual, memory segment) for each process. Table IV shows the fields of a process segment.

PS_CURRENT_PROCESS is the number of the process associated with the PS. PS_PROCESS_MASK and PS_PROCESS_NOTMASK are used in accessing AST_CPL and AST_WAL. MASK is all zero except for bit n (where n is the process number), NOTMASK is all ones except for a zero at bit n. PS_USER_ID and PS_PROJECT_ID identify the user (subject) associated with the process. PS_CUR_CLASS (current classification) and PS_CUR_CAT (current category set) define the current security level of the process. PS_TYPE indicates whether or not the process is a trusted subject.

PS_MEM_QUOTA is the amount of main memory allocated to the process for its AS but not currently being used. PS_IPC_QUOTA is the number of interprocess communication objects currently available to the user for receiving messages from other processes. PS_DISK_QUOTA is the disk space allocated to the user of the process but not yet used.

The remainder of the PS is dedicated to arrays used for defining the process's address space. PS_SDR (segmentation descriptor register) and PS_SAR (segmentation address register) are two arrays that hold the 16 descriptors (8 in supervisor domain, 8 in user domain) that are available. The third array, PS_SEG, is used for mapping segment numbers (seg#'s - process local segment names) into aste#'s (system wide segment names). When a process has the kernel move a segment into its WS, the kernel returns a seg# which the process subsequently uses to identify the segment. The segment has an aste# because it must be active, but the aste# cannot be returned to the user because its value is a function of system wide activity, making it necessary to classify aste#'s at system high. Thus, PS_SEG is just a mechanism for mapping seg#'s into aste#'s. PS_SEG is provided by level 1 to level 2 as a convenience, as active segments are not meaningful at level 1 - its operation is independent of the contents of PS_SEG. Each element of PS_SEG_INUSE, the fourth and final array, indicates whether or not the corresponding element in PS_SEG is currently in use.

Memory Block Table

The Memory Block Table (MBT) is a level 0 structure used to indicate the state of main memory. A block is the smallest size segment - the MMU supports 64 byte blocks but the initial implementation uses 256 byte blocks. Contiguous blocks can be concatenated to form main memory segments of any multiple block size. A main memory segment is either free or allocated depending on whether or not a virtual memory segment is bound to it. There is an entry in the MBT for each block (see Table V) consisting of several fields. If a block is the first block in a segment MBT_FLAGS is either FREE or ALLOCATED, otherwise it is CONCATENATED. The rest of the fields are not meaningful for CONCATENATED blocks. MBT_SIZE is the number of blocks in the segment. If a block is FREE, MBT_CHAIN is the block# of the next segment in the free chain or zero if this is the end of the chain. (A block# is the address of the first byte in a block with the 8 low order 0 bits removed.) If a block is ALLOCATED, MBT_ASTE is the aste# of the virtual memory segment bound to it. MBT_CHANGE indicates if the segment has been modified. This information can be obtained from hardware conditions - a bit in each segmentation register indicates if the segment "described" by the register has been stored into (via an access through that segmentation register).

SPECIFICATION OF THE KERNEL

The goal of certification is to prove that the behavior of a system corresponds to the behavior of a model. The model in turn must be proved to exhibit a certain desired behavior - in our case that the abstract system remains in a secure state. The final representation of the system's security kernel will be as binary ones and zeros in the computer's memory. Intermediate representations of the security kernel will be used to bridge the vast gap between the abstract model and the binary ones and zeros of executable code, thus aiding the task of proving correspondence.[11] One form of intermediate representation is the higher level language program listings of the kernel functions. This representation, however, will contain many details that are specific to particular implementation decisions and to the language used. What is needed is another representation that describes the design of the kernel in a manner that is independent of implementation and language considerations. We call this representation the design specification and its purpose is to bridge the gap between the model and implementation representations. The form of the design specification used here is

[11] Our certification methodology is discussed in [Bell & Burke].

Table IV

Format of a Process Segment (PS)

Process Segments are accessed by process#: PS_XXX(process#)

PS_CURRENT_PROCESS(8)	process#
PS_PROCESS_MASK(16)	bit mask
PS_PROCESS_NOTMASK(16)	bit mask
PS_USER_ID(14)	user identification
PS_PROJECT_ID(8)	project identification
PS_TYPE(1)	TRUSTED or UNTRUSTED
PS_CUR_CLASS(4)	current classification
PS_CUR_CAT(16)	current category set
PS_MEM_QUOTA(8)	unused main memory quota
PS_DISK_QUOTA(16)	unused disk space quota
PS_IPC_QUOTA(8)	unused ipc element quota
PS_SDR(16 x 16 array)	save area for segmentation registers
PS_SAR(16 x 16 array)	save area for segmentation registers
PS_SEG(32 x 15 array)	definition of process's address space (WS)
PS_SEG_INUSE(32 x 1 array)	TRUE or FALSE

Table V

Format of the Memory Block Table (MBT)

the MBT is accessed by block#: MBT_XXX(block#)

MBT_FLAGS(2)	FREE, ALLOCATED, or CONCATENATED
MBT_SIZE(8)	size of the area in blocks
MBT_CHAIN(14)	chain of FREE blocks
MBT_ASTE#(13)	aste# of the virtual memory segment in the block
MBT_CHANGE(1)	CHANGED or UNCHANGED

derived from a form suggested in [Parnas] and used in [Price]. Figure 7 shows the validation chain between the various representations.

A "Parnas" specification consists of two distinct types of functions: O-functions and V-functions. O-functions (operate) are functions that cause the state of the system to change. V-functions (value) return the values of state variables. Their only effect is the passage of time. The specification of each function includes: 1) the name of the function; 2) a range for possible values of the function, if it is a V-function; 3) an indication of the initial value, possibly undefined, for V-functions; 4) a list of parameters and their domain; and 5) an indication of the effect of the function on the values of other functions, for O-functions.

The effect section of each function consists of specification statements. These statements denote that upon completion of the function certain predicates will be true. The ordering of the specification statements is not significant and some of the predicates are conditional. References to V-functions enclosed in single quotes (') refer to the value of the V-function at the time of call of the O-function; references not enclosed in quotes refer to the value of the V-function immediately after completion of the defined O-function.

Parnas' intention for specification is to give an external view of functions. All of the information needed correctly to use functions and to implement them must be given, and nothing more. Also, specifications must be sufficiently formal so that their completeness, consistency and other desirable properties (in our case correctness) can be determined. This latter requirement seems to rule out the use of natural language specifications. Nevertheless, without prose descriptions of the intended interpretation, specifications can be hopelessly confusing.

Although it consists of O and V functions, the kernel specification that follows is not a Parnas specification since much more than the minimum information needed to use or implement the kernel is given. Also, the ordering of the effect of O-functions is significant. The mechanisms that support the design are included in the specification because their correctness must be proven. To make this additional information more comprehensible to the reader, the specification is structured in much the same way that software is

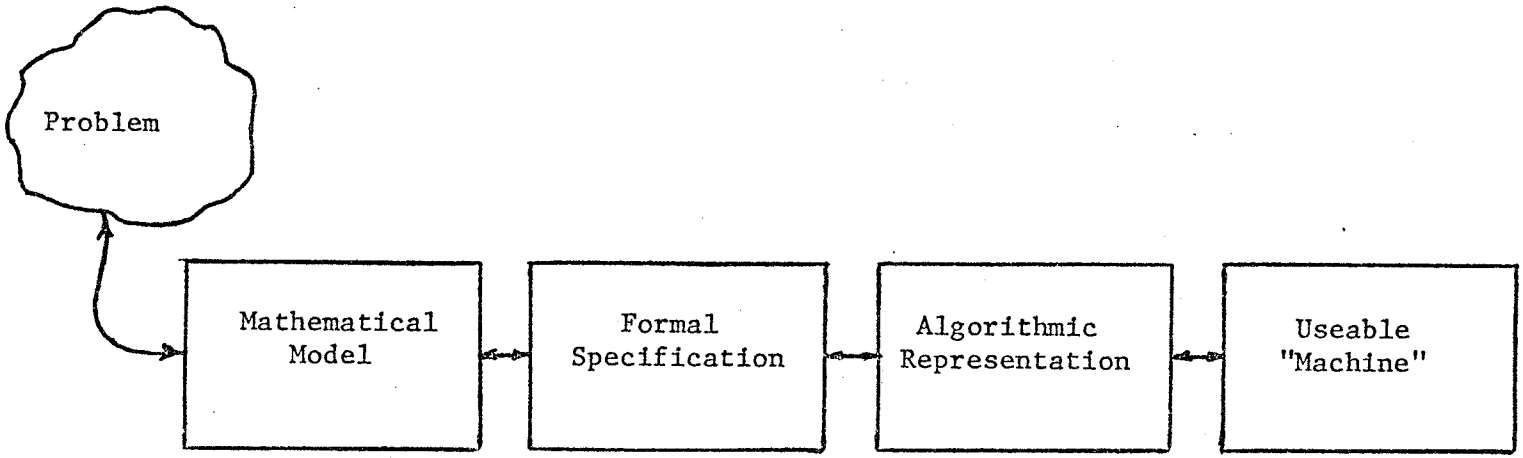


Figure 7. The Validation Chain

structured by nested procedures - a function is defined in terms of other functions, which in turn are defined with more functions, etc.[12]

Specification Conventions

A number of conventions are used in the kernel specification to enhance its clarity and reduce its bulk. One convention is to use the data structures defined in the previous subsection. The term "DIR_DISK(aste#, entry#)", for example, appears in a number of predicates in the specifications of various functions. A specification of a V-function DIR_DISK might be:

Function: DIR_DISK
possible values: a disk address
Parameters: DIR_DISK(aste#, entry#)

Since the value of DIR_DISK is set in some functions and used in others, the body of the DIR_DISK specification is empty. Given the definition of the data structures, we do not feel that the specifications of the V-functions that correspond to the data structure elements are necessary for the purposes of this paper.

---

[12]The initial work in certifying the kernel indicates that the nesting of O-functions hinders the proof process. This problem is being corrected by replacing each "call" of an O-function with the body of the called O-function. The copying works because the specification contains no recursive O-function calls. Recursion in the V-function definitions is being eliminated by the use of quantifiers. A revised specification for the kernel will be published with the proofs.

The use of a structured specification [Neumann, et al] allows the description of hierarchical design mechanisms while avoiding nested O-functions. A complete specification is written for each level, and then the V-functions at each level (except the lowest) are expressed in terms of lower level V-functions. This V-function mapping suggests certain implementation mechanisms, but these mechanisms are not made explicit until O-functions are (abstractly) implemented with lower level O-functions. Thus, when a high level O-function changes the value of a V-function, a call to the lower level O-functions that maintain the corresponding lower level V-functions is implied. The specification prover can ignore these implicit calls because, as a specification, each level is logically complete by itself. The important point of this discussion is that for a given design based on the model there are many possible, correct specifications.

These V-function specifications can be easily generated if the proof requires them.

Another convention is the use of mnemonic names for function parameters and internal variables. Table VI shows the intended interpretation of these names. TCP (The Current Process) is an internal kernel variable that indicates which process is currently bound to the CPU. It is part of the mechanism for implementing a distributed kernel and prevents users of the kernel from forging their identity.

Critical Sections

The specification assumes that the effect of O-functions is instantaneous. In the initial implementation this assumption is realized by making the entire kernel a single critical section. On entry to the kernel a P is performed on a special semaphore (the kernel semaphore), and the corresponding V is not executed until the kernel function is complete, unless the function itself is a P on a segment semaphore that causes the process to become blocked. In this case the kernel semaphore must be released (by a V) before the processor is deallocated from the blocked process and reallocated to another process, or deadlock could result.

This approach to providing determinancy is used because its correctness is obvious, and for single processor systems with one I/O device supporting the virtual memory, it is reasonably efficient. The only time that a blocked process has the kernel semaphore locked is when the kernel is waiting for internal (segment) I/O that was initiated on the process's behalf to complete. This situation will cause system inefficiency only if there are other processes blocked waiting to get into the kernel and there are no ready processes. If the device is fast (drum or fixed head disk), the inefficiency should be minimal.

If the system has a slow virtual memory device (a moving head disk, for example) and/or more than one virtual memory device, the single critical section approach may cause serious inefficiencies. In the first case, the time that a blocked process has the kernel semaphore locked will increase substantially; and in the second, it will not be possible to run more than one device at a time. To avoid these inefficiencies multiple critical sections that depend upon the data observed and modified by the various kernel functions must be introduced and represented in the specification.

Table VI

Intended Interpretations

external kernel function parameters

seg#	segment number of a segment in a process's address space (WS)
entry#	identification of an entry within a directory
class	a classification
cat	a category set
type	DATA or DIRECTORY
size	size of a segment in blocks
mode	WRITE, READ, or NO
user_id	user identification
project_id	project identification
reg#	identification of a segmentation register
process#	identification of a process
block#	main memory address of a segment

internal kernel "variables"

TCP	the current process
aste#	pointer to an AST entry
daste#	aste# for a segment known to be a directory
acle#	pointer to an ACL element
smfr#	pointer to a semaphore
ipce#	pointer to an IPC element
uid	unique identifier - a disk address

The Kernel Gate and Argument Passing

Figure 8 shows the specification of the function KERNEL. This function is the sole user entry point (or gate) into the kernel and the functions that it directly invokes are the "user callable" kernel functions. KERNEL uses PCHECK (Figure 9) to verify that the parameters given by the user are within the acceptable ranges. KERNEL and PCHECK also check that the seg# parameter (if required) specifies a segment that is currently in the process's WS, and translates the seg# into an aste#. The functions of the form "XXX_PARM" used by PCHECK indicate the parameters required by each of the user callable kernel functions.

Many of the kernel functions set the value of a per-process RC (return code) object. The security attributes of the RC object are equal to those of the process. In general, kernel functions set RC to indicate whether or not they were called correctly. A few functions use RC to return additional information to the user. Each process can always observe its own (and only its) RC object.

In the implementation, reserved locations in the user's stack segment are used for argument passing. Before calling the kernel the user process places the kernel parameters (including the code for the particular kernel function it wishes to invoke) in fixed locations in its stack. On entry, the kernel moves the user's stack segment into its own address space, copies the parameters into its own private (kernel) stack segment, and then performs the validity checking on the parameters. The RC object is also implemented as a reserved location in the user's stack, thus making it available to the user for inspection when the kernel returns.

The rest of the specification is given in the following subsections.

DIRECTORY FUNCTIONS

A set of functions is provided for manipulating the attributes of segments. These functions change the security state of the system by creating and deleting segments and adding and deleting elements to/from a segment's access control list (ACL). The common security requirement for all functions that modify segment attributes is that the modifying process currently have write access to the segment's parent directory. A function is also provided to set the RC object equal to the attributes of a segment.

```
Function: KERNEL
Parameters: KERNEL(function_code, seg#, entry#, class, cat, type,
        size, mode, user_id, project_id, reg#, process#, block#)
Effect:
IF (FUNCTION_CODE_MIN ≤ function_code ≤ FUNCTION_CODE_MAX) &
   PCHECK(function_code, seg#, entry#, class, cat, type, size,
            mode, user_id, project_id, reg#, process#, block#);
   THEN: Let aste# = PS_SEG(TCP, seg#);
      CASE OF function_code:
         1: CREATE(TCP, aste#, entry#, class, cat, type, size);
         2: DELETE(TCP, aste#, entry#);
         3: GIVE(TCP, aste#, entry#, mode, user_id, project_id);
         4: RESCIND(TCP, aste#, entry#, user_id, project_id);
         5: DIRREAD(TCP, aste#, entry#);
         6: GETW(TCP, aste#, entry#);
         7: GETR(TCP, aste#, entry#);
         8: RELEASE(TCP, aste#, seg#);
         9: ENABLE(TCP, aste#, reg#);
         10: DISABLE(TCP, reg#);
         11: KP(aste#);
         12: KV(aste#);
         13: IPCRCV;
         14: IPCSEND(process#, message, USER_DOMAIN);
         15: CONCAT(block#);
         16: SPLIT(block#, size);
         17: KSWAPOUT(block#);
      END;
   ELSE: RC(TCP) = NO;
END;
```

Figure 8. KERNEL Function

(see 8.48)

Creation and Deletion of Segments

The CREATE function (Figure 10) creates a segment inferior to a specified directory segment. The parameters of CREATE are the seg# of the intended parent, the entry# of a free directory entry in the intended parent, and attributes for the segment to be created. These attributes are the security level (classification and category set), a type (DIRECTORY or DATA), and a size.

There is NO DIRECTORY TYPE

In addition to enforcing the security requirement that the process currently have write access to the intended parent segment, CREATE also enforces the security requirement of compatibility and implementation requirements. The implementation requirements are that segments can only be created inferior to DIRECTORY segments, that the specified entry# identify an available directory entry, that the size is one of the permitted sizes, and finally that the process has sufficient disk quota to allow disk space to be allocated to the segment.

The motivation for most of the implementation requirements is straightforward. The requirement that the user specify an available directory entry is somewhat arbitrary - it would be slightly more complex for the kernel to search for a free entry. This approach allows users to establish certain conventions for the use of directory entries.[13] The decision to provide fixed sized segments has already been discussed.

The use of the disk quota mechanism insures us that the inability of a process to create a segment because of a lack of disk space is strictly a function of that process's behavior, and not the behavior of some other process. If the quota mechanism were not used all processes would have disk space allocated to them from a common pool, and an uncontrolled communication path would exist between processes. One process could use up all disk space by creating segments and then modulate the (imaginary) bit that indicates whether or not the disk is full by deleting and recreating a segment. Another process (at a lower security level) could read this bit by attempting to create a segment and then seeing if the operation was successful. In this design the success or failure of CREATE is indicated by the value of the RC object, but removing RC from the specification is not sufficient to hide the effect of CREATE from the user. The user can determine if the segment was actually created by

---

[13] For example, the segment at entry# 1 might always contain the symbolic names of the other segments inferior to the directory, and the segment at entry# 2 might be an overflow directory. The file system currently being implemented uses conventions similar to these.

```
Function: PCHECK
possible values: TRUE or FALSE
Parameters: PCHECK(function_code, seg#, entry#, class, cat, type,
         size, mode, user_id, project_id, reg#, process#, block#)
Value:
IF (not SEG#_PARM(function_code) ¦
      ((SEG#_MIN ≤ seg# ≤ SEG#_MAX) & PS_SEG_INUSE(TCP, seg#))) &
   (not ENTRY#_PARM(function_code) ¦
      (ENTRY#_MIN ≤ entry# ≤ ENTRY#_MAX)) &
   (not CLASS_PARM(function_code) ¦
      (CLASS_MIN ≤ class ≤ CLASS_MAX)) &
   (not CAT_PARM(function_code) ¦
      (cat ⊆ CATEGORY_SET)) &
   (not TYPE_PARM(function_code) ¦
      ((type = DIRECTORY) ¦ (type = DATA))) &
   (not SIZE_PARM(function_code) ¦
      (SIZE_MIN ≤ size ≤ SIZE_MAX)) &
   (not MODE_PARM(function_code) ¦
      ((mode = WRITE) ¦ (mode = READ) ¦ (mode = NO))) &
   (not USER_ID_PARM(function_code) ¦
      (USER_ID_MIN ≤ user_id ≤ USER_ID_MAX)) &
   (not PROJECT_ID_PARM(function_code) ¦
      (PROJECT_ID_MIN ≤ project_id ≤ PROJECT_ID_MAX)) &
   (not REG#_PARM(function_code) ¦
      (REG#_MIN ≤ reg# ≤ REG#_MAX)) &
   (not PROCESS#_PARM(function_code) ¦
      (PROCESS#_MIN ≤ process# ≤ PROCESS#_MAX)) &
   (not BLOCK#_PARM(function_code) ¦
      (BLOCK#_MIN ≤ block# ≤ BLOCK#_MAX));
   THEN: TRUE;
   ELSE: FALSE;
END;
```

Figure 9. PCHECK Function

```
Function: CREATE
Parameters: CREATE(process#, aste#, entry#, class, cat, type, size)
Effect:
IF not AST_WAL(aste#, process#) ¦
   (class < AST_CLASS(aste#)) ¦
   (cat ⊉ AST_CAT(aste#)) ¦
   (AST_TYPE(aste#) ≠ DIRECTORY) ¦
   ('DIR_SIZE'(aste#, entry#) ≠ 0) ¦
   (size ∉ SIZE_SET) ¦
   ((type = DIRECTORY) & (size ≠ DIRECTORY_SIZE)) ¦
   (size > 'PS_DISK_QUOTA'(process#, size));
   THEN: RC(process#) = NO;
   ELSE: DIR_TYPE(aste#, entry#) = type;
      DIR_STATUS(aste#, entry#) = UNINITIALIZED;
      DIR_CLASS(aste#, entry#) = class;
      DIR_CAT(aste#, entry#) = cat;
      DIR_SIZE(aste#, entry#) = size;
      DISK_ALLOC(size);
      DIR_DISK(aste#, entry#) = NEXT_DISK_ADDRESS;
      DIR_ACL_HEAD(aste#, entry#) = 0;
      PS_DISK_QUOTA(process#, size) =
         'PS_DISK_QUOTA'(process#, size) - size;
      ANCESTOR(NEXT_DISK_ADDRESS, AST_DISK(aste#)) = TRUE;
      (¥uid)(ANCESTOR(AST_DISK(aste#), uid));
         ANCESTOR(NEXT_DISK_ADDRESS, uid) = TRUE;
      END;
      UID_SIZE(NEXT_DISK_ADDRESS) = size;
      RC(process#) = YES;
END;

Function: DISK_ALLOC
Parameters: DISK_ALLOC(size)
Effect:
(∃k)(('BIT_MAP'(size, k) = 0) &
   (BIT_MAP(size, k) = 1) &
   (NEXT_DISK_ADDRESS = BASE(size) + k*size));

Function: ANCESTOR
possible values: TRUE or FALSE
initial value: FALSE
Parameters: ANCESTOR(uid1, uid2)

Function: UID_SIZE
possible values: size
Parameters: UID_SIZE(uid)
```

Figure 10. CREATE, DISK_ALLOC, ANCESTOR and UID_SIZE Functions

trying to read and write it.

The communication path just described is based on what we call a system-wide variable. A system-wide variable can occur any time physical resources must be shared among processes. In this case the quota mechanism eliminates the communication path - its effect is to partition the physical disk into a virtual disk for each process.[14] It is necessary, of course, for the sum of the virtual disks to be less than or equal to the physical disk. It is interesting to note that while the quota mechanism is necessary for security reasons, one would want something like it even if security were not required. The ability of one user to monopolize disk space at the expense of others is not desirable in any environment.

The effect of CREATE if all requirements are satisfied is to create a segment by putting attributes into the directory entry. Most of the attributes are directly specified by the user. The status of the segment is set to UNINITIALIZED, the ACL is set to empty, and space on the disk is allocated.

The effect statements in CREATE that set the value of the V-function's ANCESTOR and UID_SIZE require careful explanation. Briefly, these V-functions are a specification mechanism that "remember" the shape of the tree and the size of each segment. Although this information is embedded in the directory structure, we will see that having it in this form simplifies the specification of the DELETE function.

The two arguments to ANCESTOR are the unique identifiers (disk addresses) of two segments. ANCESTOR is true if the second segment is an ancestor of the first. The ancestors of a segment are its parent directory, its parent's parent, and so forth. Thus, the root is an ancestor of every segment in the tree (except itself). UID_SIZE remembers the size of a segment by uid. For completeness, Figure 10 gives specifications of ANCESTOR and UID_SIZE. The bodies of their specifications are empty because their values are set directly by O-functions.

The effect of DISK_ALLOC is to allocate space on the disk as segments are created and to set the value of NEXT_DISK_ADDRESS to the address of the space allocated. The disk is partitioned into a region for each segment size, and each region is represented by a bit string. There is one bit for each disk area that can be assigned to a segment. The bit indicates if the area is free or assigned.

---

[14] Actually, a disk quota per security level is sufficient, and this result generalizes for all system wide variables.

DISK_ALLOC scans the appropriate bit string looking for a bit indicating a free area, sets the bit to indicate an assigned area, and translates the bit address to a disk address. The quota mechanism guarantees that DISK_ALLOC will succeed in finding a free bit. If each bit string is partitioned into sections for each process based on the quotas, then the values returned by DISK_ALLOC are a function of each process's behavior, but if the bit strings are not divided into sections then each value returned by DISK_ALLOC is a function of the behavior of all processes. If the latter case holds then the security level of DIR_DISK in each directory entry is "system high".

Now that the explanation of CREATE is complete we pause to make an observation - the notion of levels of abstraction is missing from the create specification. Levels of abstraction have not been abandoned, rather, the specification has collapsed the levels of abstraction to make the specification more compact. Conceptually, there is still a separation among the abstractions that create processes, create segments, and enforce a security policy. Figure 11 shows an alternative specification of create. The function CREATE enforces the security policy, and CREATE2 enforces implementation constraints and creates the segments. Although it is passed the security attributes of the segment to be created, the effect of CREATE2 is independent of their value. For the remaining kernel functions the levels of abstraction will not be made explicit; we hope that the distinction between those parts of the specification that enforce the security policy and those that do not will be obvious to the reader.

Figures 12 and 13 give the specifications of the DELETE function and its immediate support functions. The user identifies the segment to be deleted by giving the seg# of its parent directory and the entry# of the segment. The only requirements are that the user currently have write access to the parent directory (security) and the entry specified is not a free entry (implementation).

In deleting a segment several operations must be performed: 1) the entry must be cleaned up so it can be reused; 2) if the segment is active it must be removed from the address space of all processes that currently have access to it and be deactivated; 3) the disk space allocated to the segment must be released; and 4) if the segment is a directory all of the segments inferior to it must be deleted. While the kernel would be simpler if only empty directory (or data) segments were accepted by DELETE, this approach cannot be employed because a user may be permitted to delete a directory but not know if it is empty or not. Consider a secret directory inferior to a confidential directory. If a confidential user has write access to the confidential directory, he can delete the secret directory,

```
Function: CREATE
Parameters: CREATE(process#, aste#, entry#, class, cat, type, size)
Effect:
IF not AST_WAL(aste#, process#) ¦
   (class < AST_CLASS(aste#)) ¦
   (cat ⊉ AST_CAT(aste#));
   THEN: CREATE2(process#, aste#, entry#, class, cat, type, size);
   ELSE: RC(TCP) = NO;
END;

Function: CREATE2
Parameters: CREATE2(process#, aste#, class, cat, type, size)
Effect:
IF (AST_TYPE(aste#) ≠ DIRECTORY) ¦
   ('DIR_SIZE'(aste#, entry#) ≠ 0) ¦
   (size ∉ SIZE_SET) ¦
   ((type = DIRECTORY) & (size ≠ DIRECTORY_SIZE)) ¦
   (size > 'PS_DISK_QUOTA'(process#, size));
   THEN: RC(process#) = NO;
   ELSE: DIR_TYPE(aste#, entry#) = type;
      DIR_STATUS(aste#, entry#) = UNINITIALIZED;
      DIR_CLASS(aste#, entry#) = class;
      DIR_CAT(aste#, entry#) = cat;
      DIR_SIZE(aste#, entry#) = size;
      DISK_ALLOC(size);
      DIR_DISK(aste#, entry#) = NEXT_DISK_ADDRESS;
      DIR_ACL_HEAD(aste#, entry#) = 0;
      PS_DISK_QUOTA(process#, size) =
         'PS_DISK_QUOTA'(process#, size) - size;
      ANCESTOR(NEXT_DISK_ADDRESS, AST_DISK(aste#)) = TRUE;
      (∀uid)(ANCESTOR(AST_DISK(aste#), uid));
         ANCESTOR(NEXT_DISK_ADDRESS, uid) = TRUE;
      END;
      UID_SIZE(NEXT_DISK_ADDRESS) = size;
      RC(process#) = YES;
END;
```

Figure 11. CREATE and CREATE2 Functions

```
Function: DELETE
Parameters: DELETE(process#, aste#, entry#)
Effect:
IF not AST_WAL(aste#, process#) ¦
   (AST_TYPE(aste#) ≠ DIRECTORY) ¦
   ('DIR_SIZE'(aste#, entry#) = 0);
   THEN: RC(process#) = NO;
   ELSE: Let uid = DIR_DISK(aste#, entry#);
      IF 'DIR_ACL_HEAD'(aste#, entry#) ≠ 0;
         THEN: Let acle# =
                  FINDEND(aste#, 'DIR_ACL_HEAD'(aste#, entry#);
            ACL_CHAIN(aste#, acle#) = 'ACL_CHAIN'(aste#, 0);
            ACL_CHAIN(aste#, 0) = 'DIR_ACL_HEAD'(aste#, entry#);
      END;
      DIR_SIZE(aste#, entry#) = 0;
      DELETESEG(uid);
      IF DIR_TYPE(aste#, entry#) = DIRECTORY;
         THEN: (¥duid)('ANCESTOR'(duid, uid));
            DELETESEG(duid);
      END;
      RC(process#) = YES;
END;
```

Figure 12. DELETE Function

```
Function: DELETESEG
Parameters: DELETESEG(uid)
Effect:
Let aste# = 'HASH'(uid);
IF aste# ≠ 0;
   THEN:
      (¥process#)(PROCESS#_MIN ≤ process# ≤ PROCESS#_MAX);
         IF (PT_FLAGS(process#) ≠ INACTIVE) &
            'AST_CPL'(aste#, process#);
            THEN:
               (¥seg#)(SEG#_MIN ≤ seg# ≤ SEG#_MAX);
                  IF ('PS_SEG'(process#, seg#) = aste#);
                     THEN: RELEASE(process#, aste#, seg#);
                  END;
               END;
      END;
      END;
      DEACTIVATE(uid);
END;
DISK_FREE(uid, UID_SIZE(uid));
(¥puid)('ANCESTOR'(uid, puid));
   ANCESTOR(uid, puid) = FALSE;
END;

Function: DISK_FREE
Parameters: DISK_FREE(disk_address, size)
Effect:
Let k = ((disk_address - BASE(size))/size);
BIT_MAP(size, k) = 0;
```

Figure 13. DELETESEG and DISK FREE Functions

but by virtue of the relative security levels he may not know what is in the secret directory, and the success or failure of a delete conditioned on directory empty would tell him.

Operation 1) consists of removing all of the ACL elements from the entry and putting them on the parent directory's free ACL element chain, and marking the entry free by setting DIR_SIZE to 0. This operation is performed in DELETE; 2) and 3) are done in DELETESEG. DELETE also determines if the segment being deleted is a directory, and if so, determines all of its inferiors with the ANCESTOR function and invokes DELETESEG for each one. In deleting an inferior it is not necessary to clean up its entry (operation 1), because its parent is always being deleted.

The ANCESTOR function is a mechanism that allows the specification to easily identify all of the segments in a sub-tree. The implementation does not need the ANCESTOR function (or the UID_SIZE function) because it can find all of the segments in a sub-tree by performing a tree-walk.

DISK_FREE, the inverse of DISK_ALLOC, is passed the disk address of a segment and the size of the segment. It translates the disk address into a bit address and sets the bit in the appropriate bit string to indicate that the disk area previously allocated to the segment is now free. Note that the user's disk quota is not credited in the delete function. When a user deletes a sub-hierarchy he cannot be credited with all of the disk space freed because he may not be entitled to know the size of the sub-hierarchy. At least two implementation schemes are possible: 1) as a segment is deleted the quota of the user that created the segment can be credited; or 2) periodically, the entire hierarchy can be inspected and the quotas of users can be adjusted to reflect any deletions that have occurred during the previous period. In either case, we would want a segment attribute to identify the user who created the segment.

Giving and Rescinding Access

Functions are provided for giving and rescinding access permissions (modifying M). Actually, these functions' names are deceptive. The GIVE function adds an ACL element to a segment's ACL chain and RESCIND removes an ACL element. Since an ACL element can contain the NO access mode, the "GIVE" function can remove access rights from a user.

The GIVE function (Figure 14) adds an ACL element (mode, user_id, project_id) to the directory entry (seg#, entry#) of some segment. It requires that the user currently have write access to the directory, that the entry is not free, that an ACL element with

```
Function: GIVE
Parameters: GIVE(process#, aste#, entry#, mode, user_id, project_id)
Effect:
IF not AST_WAL(aste#, process#) ¦
   (AST_TYPE(aste#) ≠ DIRECTORY) ¦
   (DIR_SIZE(aste#, entry#) = 0) ¦
   DUPACL(aste#, 'DIR_ACL_HEAD'(aste#, entry#), user_id, project_id) ¦
   ('ACL_CHAIN'(aste#, 0) = 0);
   THEN: RC(process#) = NO;
   ELSE: Let acle# = 'ACL_CHAIN'(aste#, 0);
      ACL_CHAIN(aste#, 0) = 'ACL_CHAIN'(aste#, acle#);
      Let position = FACLPOS(aste#, 'DIR_ACL_HEAD'(aste#, entry#),
         user_id, project_id);
      IF position = 0;
         THEN: ACL_CHAIN(aste#, acle#) =
                  'DIR_ACL_HEAD'(aste#, entry#);
            DIR_ACL_HEAD(aste#, entry#) = acle#;
         ELSE: ACL_CHAIN(aste#, acle#) = 'ACL_CHAIN'(aste#, position);
            ACL_CHAIN(aste#, position) = acle#;
      END;
      ACL_USER(aste#, acle#) = user_id;
      ACL_PROJECT(aste#, acle#) = project_id;
      ACL_MODE(aste#, acle#) = mode;
      SOADD(aste#, entry#);
      RC(process#) = YES;
END;
```

Figure 14. GIVE Function

the same (user_id, project_id) is not already on the ACL (this check is performed by DUPACL), and that there is a free ACL element to use for this request. If all constraints are satisfied then the effect is to allocate a free ACL element, find the correct position for it in the ACL chain, put it there, fill it in as specified by the user, and invoke SOADD. The function of SOADD will be explained shortly.

The specifications of DUPACL and FACLPOS are given later in this subsection. FACLPOS finds the correct position for a new ACL element by using the rules discussed in the subsection on data structures - ACL elements with a more specific (user_id, project_id) go before ACL elements with a more general (user_id, project_id).

The RESCIND function (Figure 15) is the inverse of give - it removes an ACL element from the ACL of a directory entry. Rescind requires that the user currently have write access to the directory, that the specified entry is in use, and that the specified ACL element is currently on the ACL. The function's effect is to remove the ACL element from the entry's ACL, add it to the directory's free ACL element chain, and invoke SOADD. FINDACLE returns the acle# of an ACL element, and FINDPACLE returns the acle# of the previous ACL element. These functions will be specified shortly.

Directory Support Functions

There are a number of functions to support the manipulation of ACL chains. Figure 16 gives the specifications of DUPACL and FACLPOS. DUPACL indicates whether or not a given ACL element (independent of the mode) is on an ACL chain. FACLPOS finds the correct place in an ACL chain to place a new element based on rules previously discussed. It employs FINDEND to find the last ACL element in a chain, and FINDUSER to find the last ACL element in a chain that does not have a user_id of ALL_USERS. Figure 17 gives the specifications of FINDEND and FINDUSER, as well as FINDACLE and FINDPACLE. FINDACLE finds the acle# of a specified ACL element, and FINDPACLE the previous acle# in the chain.

The specification of SOADD (search out and destroy descriptors) is given in Figure 18. Whenever the ACL of a segment changes it is necessary to insure that any process that has the segment in its WS still has access rights. If in fact a process has lost its access rights because of the changed ACL, the segment must be removed from its WS. SOADD performs this function.

If the segment is not active it cannot be in the WS of any process. Otherwise, for each process on the segment's connected process list SOADD determines the process's mode of access, re-searches the ACL using DSEARCH, and if the search fails removes

```
Function: RESCIND
Parameters: RESCIND(process#, aste#, entry#, user_id, project_id)
Effect:
IF not AST_WAL(aste#, process#) |
   (AST_TYPE(aste#) ≠ DIRECTORY) |
   (DIR_SIZE(aste#, entry#) = 0) |
   not DUPACL(aste#, 'DIR_ACL_HEAD'(aste#, entry#), user_id,
      project_id);
   THEN: RC(process#) = NO;
   ELSE: Let acle# = FINDACLE(aste#, 'DIR_ACL_HEAD'(aste#, entry#),
            user_id, project_id);
      IF acle# = 'DIR_ACL_HEAD'(aste#, entry#);
         THEN: DIR_ACL_HEAD(aste#, entry#) =
               'ACL_CHAIN'(aste#, acle#);
         ELSE: Let pacle# = FINDPACLE(aste#,
               'DIR_ACL_HEAD'(aste#, entry#), acle#);
            ACL_CHAIN(aste#, pacle#) = 'ACL_CHAIN'(aste#, acle#);
      END;
      ACL_CHAIN(aste#, acle#) = 'ACL_CHAIN'(aste#, 0);
      ACL_CHAIN(aste#, 0) = acle#;
      SOADD(aste#, entry#);
      RC(process#) = YES;
END;
```

Figure 15. RESCIND Function

```
Function: DUPACL
possible values: TRUE or FALSE
Parameters: DUPACL(aste#, acle#, user_id, project_id);
Value:
IF acle# = 0;
   THEN: FALSE;
   ELSE:
      IF (ACL_USER(aste#, acle#) = user_id) &
         (ACL_PROJECT(aste#, acle#) = project_id);
         THEN: TRUE;
         ELSE: DUPACL(aste#, ACL_CHAIN(aste#, acle#), user_id,
                  project_id);
      END;
END;

Function: FACLPOS
possible values: acle# or 0
Parameters: FACLPOS(aste#, acle#, user_id, project_id)
Value:
IF acle# = 0;
   THEN: 0;
   ELSE:
      IF (user_id = ALL_USERS) &
         (project_id = ALL_PROJECTS);
         THEN: FINDEND(aste#, acle#);
         ELSE:
            IF (user_id = ALL_USERS) |
               (project_id = ALL_PROJECTS);
               THEN:
                  IF ACL_USER(aste#, acle#) = ALL_USERS;
                     THEN: 0;
                     ELSE: FINDUSER(aste#, acle#);
                  END;
               ELSE: 0;
            END;
      END;
END;
```

Figure 16. DUPACL and FACLPOS Functions

```
Function: FINDEND
possible values: acle#
Parameters: FINDEND(aste#, acle#)
Value:
IF ACL_CHAIN(aste#, acle#) ≠ 0;
   THEN: FINDEND(aste#, ACL_CHAIN(aste#, acle#));
   ELSE: acle#;
END;

Function: FINDUSER
possible values: acle#
Parameters: FINDUSER(aste#, acle#);
Value:
IF (ACL_CHAIN(aste#, acle#) = 0) |
   (ACL_USER(aste#, ACL_CHAIN(aste#, acle#)) = ALL_USERS);
   THEN: acle#;
   ELSE: FINDUSER(aste#, ACL_CHAIN(aste#, acle#));
END;

Function: FINDACLE
possible values: acle#
Parameters: FINDACLE(aste#, acle#, user_id, project_id)
Value:
IF (ACL_USER(aste#, acle#) = user_id) &
   (ACL_PROJECT(aste#, acle#) = project_id);
   THEN: acle#;
   ELSE: FINDACLE(aste#, ACL_CHAIN(aste#, acle#), user_id,
            project_id);
END;

Function: FINDPACLE
possible values: acle#
Parameters: FINDPACLE(aste#, vacle#, acle#)
Value:
IF ACL_CHAIN(aste#, vacle#) = acle;
   THEN: vacle#;
   ELSE: FINDPACLE(aste#, ACL_CHAIN(aste#, vacle#,), acle#);
END;
```

Figure 17. FINDEND, FINDUSER, FINDACLE, and FINDPACLE Functions

```
Function: SOADD
Parameters: SOADD(daste#, entry#)
Effect:
Let aste# = 'HASH'(DIR_DISK(daste#, entry#));
IF aste# ≠ 0;
   THEN:
      (¥process#)(PROCESS#_MIN ≤ process# ≤ PROCESS#_MAX);
         IF PT_FLAGS(process#) ≠ INACTIVE) &
            'AST_CPL'(aste#, process#);
            THEN:
               IF 'AST_WAL'(aste#, process#);
                  THEN: Let mode = WRITE;
                  ELSE: Let mode = READ;
               END;
               IF not DSEARCH(process#, daste#,
                     DIR_ACL_HEAD(daste#, entry#), mode);
                  THEN:
                     (¥seg#)(SEG#_MIN ≤ seg# ≤ SEG#_MAX);
                        IF ('PS_SEG'(process#, seg#) = aste#);
                           THEN: RELEASE(process#, aste#, seg#);
                        END;
                     END;
               END;
         END;
      END;
END;
```

Figure 18. SOADD Function

the segment from the process's WS with RELEASE. The specifications of DSEARCH and RELEASE are in the next subsection.

The process is not given any explicit notification when the kernel removes a segment from its address space. The process will receive an error message the next time it tries to access the segment or use its seg# for the segment as an argument to a kernel function. Since the kernel does not "remember" that SOADD performed the removal, the error message will not, in general, be sufficient to determine the underlying cause of the error message. That is, the error message alone will not enable the user to distinguish between the case where some other user removed him from a segment's ACL and the case where there is a bug in his program. It seems likely that human intervention will be necessary when a process has a segment removed from its address space by some other process.

Reading Directories

Since directories are composite objects - they contain data at different security levels including system high - users cannot have direct read access to directories. A function, DIRREAD (see Figure 19) is provided to allow users to read the data in directories that is at the security level of the directory. This function is an example of a function that gives a process interpretive read access to an object that is already in its address space as defined by the current security state. DIRREAD verifies that the user currently has read access to the directory and then stores into the RC object the values of the type, security level, and size fields of the specified directory entry.

ACCESSING SEGMENTS

Functions are provided for moving segments into and out of a process's WS - the design's interpretation of the model's set b, and a process's AS. The functions that change a process's WS change the state of the system with respect to security (and thus correspond to model rules), whereas the functions that change AS are only changing the representation of the current security state. There are also internal kernel functions to support the implementation of WS and AS.

Getting and Releasing Access

External kernel functions are provided for getting and releasing access to segments - these functions move segments into and out of WS. Although a process can directly address (with machine instructions) only those segments in its WS that are also in its AS because of hardware segmentation register constraints, WS is defined

```
Function: DIRREAD
Parameters: DIRREAD(process#, aste#, entry#)
Effect:
IF (AST_TYPE(aste#) = DIRECTORY) &
   (DIR_SIZE(aste#, entry#) ≠ 0);
   THEN: RC(process#) = DIR_TYPE(aste#, entry#),
                        DIR_CLASS(aste#, entry#),
                        DIR_CAT(aste#, entry#),
                        DIR_SIZE(aste#, entry#);
   ELSE: RC(process#) = NO;
END;
```

Figure 19. DIRREAD Function

(for security purposes) to be the address space of a process.

The security requirements that must be satisfied before a process can get access to a segment are: 1) the process must currently have read (or write) access to the segment's parent directory; 2) the process must be on the ACL of the segment in the desired mode; 3) the security condition must be satisfied - the security level of the process must be greater than or equal to the security level of the segment; and 4) the *-property condition must be satisfied - untrusted processes can only have write access to segments at a single security level and read access to segments whose security level is less than or equal to the write access security level. (The write access level is the current security level.) Figure 20 shows the two functions for getting access - GETW (get write) and GETR (get read). Both of these functions require that the user identify the segment he wishes to access by giving the seg# of the parent directory and the entry# into the parent of the entry for the segment. This method of identification is sufficient to enforce security requirement 1). The ACL is searched by DSEARCH. In GETW a destinction is made between trusted and untrusted processes because the *-property is not imposed upon trusted processes. For untrusted processes it is sufficient to enforce the *-property - it is a stronger condition than the security condition. No distinction is made between trusted and untrusted processes by GETR because, for read access, the security condition and *-property condition are equivalent. If all security requirements are satisfied GETW and GETR invoke CONNECT which makes implementation checks and moves the segment into the process's WS.

The RELEASE function (Figure 21) removes a segment from a process's WS. There are no constraints other than the requirement that the seg# parameter be valid. A segment cannot be removed from WS if it is in AS. Given that RELEASE must check to see if the segment is in the process's AS, it is just as easy for RELEASE to remove the segment from AS (with DISABLE) as it is to refuse the WS removal. The removal from WS is performed by disconnecting the process from the segment's ASTE. If after the disconnection there are no other processes connected to the ASTE, then the segment is marked as eligible for deactivation by AGE.

The DSEARCH function searches an ACL chain looking for an ACL element that applies to the invoking process - an ACL element with a user-id equal to the process's user-id or ALL_USERS and a project-id equal to the process's project-id or ALL_PROJECTS. If an ACL element is found the mode field is checked. A mode of WRITE is required by GETW; a mode of WRITE or READ is sufficient for GETR.

```
Function: GETW
Parameters: GETW(process#, aste#, entry#)
Effect:
IF (AST_TYPE(aste#) ≠ DIRECTORY) ¦
   (DIR_SIZE(aste#, entry#) = 0) ¦
   not DSEARCH(process#, aste#, DIR_ACL_HEAD(aste#, entry#), WRITE);
   THEN: RC(process#) = NO;
   ELSE:
      IF PS_TYPE(process#) = TRUSTED;
         THEN:
            IF (PS_CUR_CLASS(process#) < DIR_CLASS(aste#, entry#)) ¦
               (PS_CUR_CAT(process#) ⊉ DIR_CAT(aste#, entry#));
               THEN: RC(process#) = NO;
               ELSE: CONNECT(process#, aste#, entry#, WRITE);
            END;
         ELSE:
            IF (PS_CUR_CLASS(process#) ≠ DIR_CLASS(aste#, entry#)) ¦
               (PS_CUR_CAT(process#) ≠ DIR_CAT(aste#, entry#));
               THEN: RC(process#) = NO;
               ELSE: CONNECT(process#, aste#, entry#, WRITE);
            END;
      END;
END;

Function: GETR
Parameters: GETR(process#, aste# , entry#)
Effect:
IF (AST_TYPE(aste#) ≠ DIRECTORY) ¦
   (DIR_SIZE(aste#, entry#) = 0) ¦
   not DSEARCH(process#, aste#, DIR_ACL_HEAD(aste#, entry#), READ)
   (PS_CUR_CLASS(process#) < DIR_CLASS(aste#, entry#)) ¦
   (PS_CUR_CAT(process#) ⊉ DIR_CAT(aste#, entry#));
   THEN: RC(process#) = NO;
   ELSE: CONNECT(process#, aste#, entry#, READ);
END;
```

Figure 20. GETW and GETR Functions

```
Function: RELEASE
Parameters: RELEASE(process#, aste#, seg#)
Effect:
Let block# = 'AST_ADR'(aste#);
IF (block# ≠ 0);
   THEN:
      (∀reg#)(REG#_MIN ≤ reg# ≤ REG#_MAX) &
         IF ('PS_SAR'(process#, reg#) = block#);
            THEN: DISABLE(process#, reg#);
         END;
      END;
END;
AST_CPL(aste#, process#) = FALSE;
AST_WAL(aste#, process#) = FALSE;
IF not (∃i)((PROCESS#_MIN ≤ i ≤ PROCESS#_MAX) &
      (AST_CPL(aste#, i) = TRUE));
   THEN: AGE(aste#);
END;
PS_SEG(process#, seg#) = 'PS_SEG'(process#, 0);
PS_SEG(process#, 0) = seg#;
PS_SEG_INUSE(process#, seg#) = FALSE;

Function: DSEARCH
possible values: TRUE or FALSE
Parameters: DSEARCH(process#, aste#, acle#, mode)
Value:
IF acle# ≠ 0;
   THEN:
      IF ((ACL_USER(aste#, acle#) = ALL_USERS) ¦
         (ACL_USER(aste#, alce#) = PS_USER_ID(process#)) &
         ((ACL_PROJECT(aste#, acle#) = ALL_PROJECTS) ¦
         (ACL_PROJECT(aste#, acle#) = PS_PROJECT_ID(process#));
         THEN:
            IF ACL_MODE(aste#, acle#) = NO;
               THEN: FALSE;
               ELSE:
                  IF (mode = WRITE) &
                     (ACL_MODE(aste#, acle#) ≠ WRITE);
                     THEN: FALSE;
                     ELSE: TRUE;
                  END;
            END;
         ELSE: DSEARCH(process#, aste#, ACL_CHAIN(aste#, acle#),
               mode);
      END;
   ELSE: FALSE;
END;
```

Figure 21. RELEASE and DSEARCH Functions

WS Support Functions

The get and release functions invoke internal functions that support the concepts of connection, activation, deactivation, and eligible for deactivation. Figure 22 shows the specification of the CONNECT function. All segments that are in the WS of one or more processes are active; each process that has a segment in its WS is connected to the segment. The implementation constraints enforced by CONNECT are that the process must have a free seg# and the process cannot already be connected to the segment.[15] If the segment is not active it must be activated, and if it is active but eligible for deactivation it must be made ineligible. The actual connection is made by adding the process to the CPL (connected process list) in the ASTE and, if the process is gaining write access, the WAL (write access list) also. If the connection is successful, CONNECT sets the RC object to the value of the seg# by which the process can subsequently refer to the segment.

Figure 23 gives the specification of the ACTIVATE and DEACTIVATE functions. The parameters of ACTIVATE identify the segment to be activated - the aste# of the parent directory and the entry# of the segment's entry in the directory. To activate a segment, a free ASTE must be found. A chain of free ASTE's begins at AST_CHAIN(0), but this chain may be empty. If this is the case, then an ASTE to be made free is chosen (by NEXTASTE), and the freeing is effected by deactivating the segment using the ASTE. We must insure that a free ASTE can always be obtained, since otherwise ACTIVATE would fail making CONNECT fail. Ultimately GETW or GETR could fail for reasons that are not necessarily a function of the behavior of the process invoking the external kernel function - another instance of the system wide variable problem. Since the ability of a process to deplete the ASTE resource is constrained by the size of its WS, we can guarantee that a free ASTE can always be obtained by making the number of ASTE's at least equal to the sum of all WS's plus the number of ASTE's needed internally by the kernel. If an active segment is not in the WS of any process (or wired down) then it is eligible for deactivation. The actual activation is straightforward - the directory entry except for the ACL is copied into the ASTE and other fields in the ASTE are initialized. The segment is known not to be in main memory, not to have any segment descriptors pointing to it, and not to be in the WS of any process.

---

[15] This latter restriction prevents a process from having two different seg#'s for a segment. It is enforced to simplify the RELEASE function.

```
Function: CONNECT
Parameters: CONNECT(process#, daste#, entry#, mode)
Effect:
IF 'PS_SEG'(process#, 0) = 0;
   THEN: RC(process#) = NO;
   ELSE: Let flag = 'HASH'(DIR_DISK(daste#, entry#));
      IF (flag ≠ 0) &
         'AST_CPL'(flag, process#);
         THEN: RC(process#) = NO;
         ELSE:
            IF flag ≠ 0;
               THEN: Let aste# = flag;
                  IF 'AST_AGE'(aste#) = AGED;
                     THEN: UNAGE(aste#);
                  END;
               ELSE: ACTIVATE(daste#, entry#);
                  Let aste# = HASH(DIR_DISK(daste#, entry#));
                  UNAGE(aste#);
            END;
            AST_CPL(aste#, process#) = TRUE;
            IF mode = WRITE;
               THEN: AST_WAL(aste#, process#) = TRUE;
            END;
            Let seg# = 'PS_SEG'(process#, 0);
            PS_SEG(process#, 0) = 'PS_SEG'(process#, seg#);
            PS_SEG(process#, seg#) = aste#;
            PS_SEG_INUSE(process#, seg#) = TRUE;
            RC(process#) = YES, seg#;
      END;
END;
```

Figure 22. CONNECT Function

```
Function: ACTIVATE
Parameters: ACTIVATE(daste#, entry#)
Effect:
IF 'AST_CHAIN'(0) = 0;
   THEN: Let aste# = NEXTASTE('AST_AGE_CHAIN'(0));
      DEACTIVATE(aste#);
   ELSE: Let aste# = 'AST_CHAIN'(0);
      AST_CHAIN(0) = 'AST_CHAIN'(aste#);
END;
HASH(DIR_DISK(daste#, entry#)) = aste#;
AST_ADR(aste#) = 0;
AST_LOCK(aste#) = UNLOCKED;
AST_DES_COUNT(aste#) = 0;
(¥process#)(PROCESS#_MIN ≤ process# ≤ PROCESS#_MAX);
   AST_CPL(aste#, process#) = FALSE;
   AST_WAL(aste#, process#) = FALSE;
END;
AST_TYPE(aste#) = DIR_TYPE(daste#, entry#);
AST_STATUS(aste#) = 'DIR_STATUS'(daste#, entry#);
AST_CLASS(aste#) = DIR_CLASS(daste#, entry#);
AST_CAT(aste#) = DIR_CAT(daste#, entry#);
AST_DISK(aste#) = DIR_DISK(daste#, entry#);
AST_SIZE(aste#) = DIR_SIZE(daste#, entry#);
IF 'DIR_STATUS'(daste#, entry#) = UNINITIALIZED;
   THEN: DIR_STATUS(daste#, entry#) = INITIALIZED;
END;
AGE(aste#);

Function: DEACTIVATE
Parameters: DEACTIVATE(aste#)
Effect:
UNAGE(aste#);
IF 'AST_STATUS'(aste#) = UNINITIALIZED;
   THEN: SWAPIN(aste#);
      SWAPOUT(aste#);
   ELSE:
      IF 'AST_ADR'(aste#) ≠ 0;
         THEN: SWAPOUT(aste#);
      END;
END;
HASH(AST_DISK(aste#)) = 0;
AST_CHAIN(aste#) = 'AST_CHAIN'(0);
AST_CHAIN(0) = aste#;
```

Figure 23. ACTIVATE and DEACTIVE Functions

The design is structured so that once a segment is activated, the ASTE contains all of the information necessary to swap the segment into and out of main memory and deactivate it - no more references to the parent directory are required. To preserve this structure it is necessary to set DIR_STATUS to INITIALIZED if it is UNINITIALIZED, even though the segment is not initialized at activate time. We must, of course, insure that the segment is initialized before it is deactivated.

ACTIVATE invokes AGE to mark the segment eligible for deactivation. It does this because at activate time no processes are put on the CPL and it is not known if any will be - ACTIVATE is invoked by DELETEDIR as well as CONNECT. Finally, there is a function HASH whose input is the disk address (a unique identifier) of a segment and whose output is the aste# of the segment, if it is active, otherwise 0. HASH uses a hashing function and a hash table, and resolves hashing collisions by running chains through the AST. ACTIVATE must update HASH's data base.

The DEACTIVATE function is much simpler than ACTIVATE. It removes the segment from the list of segments eligible for deactivation, causes the segment to be initialized if it is UNINITIALIZED, swaps it out of main memory if it is in, updates HASH's data base, and adds the ASTE to the list of free ASTE's.

The kernel maintains a list of active segments eligible for deactivation by running a chain through the AST_AGE_CHAIN field of the AST. The head of the chain is the segment that most recently became eligible for deactivation, the tail is the segment that has been eligible the longest. In addition to HASH, Figure 24 gives specifications for the four functions that deal with this chain, AGE, UNAGE, FINDUNAGE, and NEXTASTE. (The body of HASH's specification is empty because its value is always set by ACTIVATE and DEACTIVATE.) AGE adds a segment to the head, UNAGE removes a segment from the chain, FINDUNAGE finds the segment's position in the chain for UNAGE, and NEXTASTE returns the aste# of the segment at the tail. NEXTASTE implements a policy that does not have to be in the kernel - when a segment has to be deactivated to make an ASTE available, the segment that has been eligible for deactivation longest is chosen. The design is done this way because the policy seems reasonable, the distinction between segments eligible for deactivation and those that are not must be maintained within the kernel, and to have a mechanism that permitted uncertified software to implement an alternative policy would add more complexity and overhead than it saved.

We conclude the treatment of WS functions with a few remarks on why we choose to postpone deactivation until the last possible moment, rather then doing it as soon as possible. Given the

```
Function: HASH
possible values: aste# or 0
initial value: 0
Parameters: HASH(disk_address)

Function: AGE
Paramteters: AGE(aste#)
Effect:
AST_AGE_CHAIN(aste#) = 'AST_AGE_CHAIN'(0);
AST_AGE_CHAIN(0)= aste#;
AST_AGE(aste#) = AGED;

Function: UNAGE
Parameters: UNAGE(aste#)
Effect:
Let vaste# = 'FINDUNAGE'(0, aste#);
AST_AGE_CHAIN(vaste#) = 'AST_AGE_CHAIN'(aste#);
AST_AGE(aste#) = UNAGED;

Function: FINDUNAGE
possible values: aste#
Parameters: FINDUNAGE(vaste#, aste#)
Value:
IF AST_AGE_CHAIN(vaste#) = aste#;
   THEN: aste#;
   ELSE: FINDUNAGE(AST_AGE_CHAIN(vaste#), aste#);
END;

Function: NEXTASTE
posssible values: aste#
Parameters: NEXTASTE(aste#)
Value:
IF AST_AGE_CHAIN(aste#) = 0;
   THEN: aste#;
   ELSE: NEXTASTE(AST_AGE_CHAIN(aste#));
END;
```

Figure 24. HASH, AGE, UNAGE, FINDUNAGE and NEXTASTE Functions

requirement on the number of ASTE's, it should be clear that as soon as processes start to share segments - the CPL of an ASTE contains more than one process - there will be more ASTE's than are needed. (The requirement for ASTE's assumes the worst case - absolutely no segment sharing.) The strategy of AGEing rather than DEACTIVATing is intended to take advantage of these surplus ASTE's. It is much more efficient to AGE and then UNAGE than it is to DEACTIVATE and ACTIVATE. The assumption is that there will be segments eligible for deactivation that are moved back into a process's WS before they are actually deactivated.

If the hardware had adequate support for segment and page faults, then the requirement on the number of ASTE's would go away. In the extreme case of little or no segment sharing, it would be possible to deactivate a segment out from under a process by setting a segment fault. Even in this environment, however, it would still be desirable to postpone deactivation and to have sufficient ASTE's to make the postponement worthwhile.

Enabling and Disabling Access

Since the number of hardware descriptors available on the PDP-11/45 prevents a process from having a descriptor for each segment in its WS (and thus directly accessing it), external kernel functions are provided for managing the allocation of descriptors. The ENABLE function (Figure 25) moves a segment in a process's WS into its AS, allocating a descriptor to the segment. The parameters of the ENABLE function are the seg# of the segment and the reg# of the segmentation register to use. Since moving a segment into AS only changes the representation of the current security state and not the state itself, all of the constraints imposed by ENABLE are implementation and consistency constraints. ENABLE requires that the seg# parameter be valid, that the specified segmentation register be free, and that the process have sufficient main memory quota. The main memory quota supports a mechanism similar to that used for controlling disk space allocation. Main memory is effectively partitioned into areas for each process. This mechanism is required because if a descriptor exists for a segment, that segment is locked into main memory - missing segment/page faults are not supported. The inability of a process to enable access to a segment must be due strictly to its own behavior and not the behavior of some other process.[16]

[16] As with the disk quota, a per-security-level main memory quota is sufficient for security.

```
Function: ENABLE
Parameters: ENABLE(process#, aste#, reg#)
Effect:
Let size = AST_SIZE(aste#);
IF ('PS_SAR'(process#, reg#) ≠ 0) |
   ((AST_WIRED_DOWN(aste#) = OFF) & (size > 'PS_MEM_QUOTA'(process#));
   THEN: RC(process#) = NO;
   ELSE:
      IF 'AST_ADR'(aste#) = 0;
         THEN: SWAPIN(aste#);
      END;
      IF 'AST_LOCK'(aste#) = UNLOCKED;
         THEN: LOCK(aste#);
      END;
      IF AST_TYPE(aste#) = DIRECTORY;
         THEN: Let mode = NO;
         ELSE:
            IF AST_WAL(aste#, process#) = TRUE;
               THEN: Let mode = WRITE;
               ELSE: Let mode = READ;
            END;
      END;
      LSD(process#, AST_ADR(aste#), reg#, mode);
      IF AST_WIRED_DOWN(aste#) = OFF;
         THEN: PS_MEM_QUOTA(process#) = 'PS_MEM_QUOTA'(process#) -
                  AST_SIZE(aste#);
      END;
      AST_DES_COUNT(aste#) = 'AST_DES_COUNT'(aste#) + 1;
      RC(process#) = YES
END;

Function: LSD
Parameters: LSD(process#, block#; reg#; mode)
Effect:
PS_SAR(process#, reg#) = block#;
PS_SDR(process#, reg#) = MBT_SIZE(block#), mode;
```

Figure 25. ENABLE and LSD Functions

Before a descriptor can actually be constructed, the segment must be swapped into main memory, if it is not in already, and the mode of access that the user has requested must be determined. Enabling access to directories is allowed, but the resulting descriptor will not actually allow access - the effect is simply to lock the directory into main memory. The actual construction of segment descriptors and the storing of them into descriptor registers is performed by LSD (Load Segment Descriptor). ENABLE concludes by debiting the main memory quota of the process and incrementing the descriptor count for the segment.

The DISABLE function (Figure 26) is the inverse of the ENABLE function - it removes a segment from AS and makes a segmentation register free by destroying the descriptor in it. DISABLE's only parameter is the reg# of the segmentation register and its only requirement is that the register actually contain a descriptor. The aste# of the segment is determined, the change bit in the descriptor register is "remembered", the descriptor is destroyed, the segment's descriptor count is decremented, and the process's memory quota is credited. If the segment's descriptor count goes to zero, the segment is marked as eligible for being swapped out of main memory.

AS Support Functions

There are several internal functions that support the implementation of AS. Figure 27 shows the specification of the SWAPIN and SWAPOUT functions. SWAPIN swaps the segment specified by the aste# parameter into main memory. First, SWAPIN finds an area of memory of the correct size and removes it from the free chain. Then, depending on whether or not the segment must be initialized, it either initializes the segment or reads it in off the disk and waits for the disk I/O to complete. Finally, SWAPIN updates the AST and MBT and invokes UNLOCK to put the segment on the SWAP_CHAIN.

SWAPOUT removes a segment from main memory. The segment need not be written back to the disk unless it has changed since it was swapped in - MBT_CHANGE indicates whether or not this is the case. The segment is removed from the SWAP_CHAIN, and the memory it formerly occupied is put on the free memory chain.

Figure 28 gives the specifications of INITSEG, DISKIO, and the two functions that manipulate the SWAP_CHAIN. Directories are initialized by marking all entries as free and putting all of the ACL elements on the free ACL chain. Data segments are initialized to all zero. DISKIO simply initiates a disk I/O operation.

The SWAP_CHAIN contains all segments that are in main memory but are eligible to be swapped out because there are no segment

```
Function: DISABLE
Parameters: DISABLE(process#, reg#)
Effect:
IF 'PS_SAR'(process#, reg#) ≠ 0;
   THEN: Let block# = 'PS_SAR'(process#, reg#);
      Let aste# = MBT_ASTE(block#);
      MBT_CHANGE(block#) = 'MBT_CHANGE'(block#) |
         'PS_SDR_CHANGE(process#, reg#);
      PS_SAR(process#, reg#) = 0;
      PS_SDR(process#, reg#) = 0;
      AST_DES_COUNT(aste#) = 'AST_DES_COUNT'(aste#) - 1;
      IF (AST_DES_COUNT(aste#) = 0) &
         (AST_WIRED_DOWN(aste#) = OFF);
         THEN: UNLOCK(aste#);
      END;
      IF AST_WIRED_DOWN(aste#) = OFF;
         THEN: PS_MEM_QUOTA(process#) = 'PS_MEM_QUOTA'(process#) +
                  AST_SIZE(aste#);
      END;
END;
```

Figure 26. DISABLE Function

```
Function: SWAPIN
Parameters: SWAPIN(aste#)
Effect:
Let size = AST_SIZE(aste#);
Let block# = FINDFREE('MBT_CHAIN'(0), size);
ALLOCMEM('MBT_CHAIN'(0), block#);
IF 'AST_STATUS'(aste#) = UNINITIALIZED;
   THEN: INITSEG(aste#, block#);
      AST_STATUS(aste#) = INITIALIZED:
      MBT_CHANGE(block#) = CHANGED;
   ELSE: DISKIO(aste#, block#, DISK_READ);
      MBT_CHANGE(block#) = UNCHANGED;
      P(DISK_SEMAPHORE);
END;
AST_ADR(aste#) = block#;
MBT_ASTE(block#) = aste#;
UNLOCK(aste#);

Function: SWAPOUT
Parameters: SWAPOUT(aste#)
Effect:
Let block# = 'AST_ADR'(aste#);
LOCK(aste#);
AST_ADR(aste#) = 0;
IF MBT_CHANGE(block#) = CHANGED;
   THEN: DISKIO(aste#, block#, DISK_WRITE);
   P(DISK_SEMAPHORE);
END;
FREEMEM('MBT_CHAIN'(0), block#);
```

Figure 27. SWAPIN and SWAPOUT Functions

```
Function: INITSEG
Parameters: INITSEG(aste#, block#)
Effect:
IF AST_TYPE(aste#) = DIRECTORY;
   THEN:
      (¥i)(ENTRY#_MIN ≤ i ≤ ENTRY#_MAX);
         DIR_SIZE(aste#, i) = 0;
      END;
      (¥j)(ACLE#_MIN ≤ j ≤ ACLE#_MAX);
         ACL_CHAIN(aste#, j) = (j+1) MODULO (ACLE#_MAX+1);
      END;
   ELSE: segment_contents = 0;
END;

Function: DISKIO
Parameters: DISKIO(aste#, block#, command)
Effect:
DISK_ADR = AST_DISK(aste#);
DISK_COUNT = AST_SIZE(aste#);
MEM_ADR = block#;
DISK_COMMAND = command, ENABLE_INTERRUPTS;

Function: LOCK
Parameters: LOCK(aste#)
Effect:
Let vaste# = 'FINDLOCK'(0, aste#);
AST_SWAP_CHAIN(vaste#) = 'AST_SWAP_CHAIN'(aste#);
AST_LOCK(aste#) = LOCKED;

Function: FINDLOCK
possible values: aste#
Parameters: FINDLOCK(vaste#, aste#)
Value:
IF AST_SWAP_CHAIN(vaste#) = aste#;
   THEN: vaste#;
   ELSE: FINDLOCK(AST_SWAP_CHAIN(vaste#), aste#);
END;

Function: UNLOCK
Parameters: UNLOCK(aste#)
Effect:
AST_SWAP_CHAIN(aste#) = 'AST_SWAP_CHAIN'(0);
AST_SWAP_CHAIN(0) = aste#;
AST_LOCK(aste#) = UNLOCKED;
```

Figure 28. INITSEG, DISKIO, LOCK, FINDLOCK and UNLOCK Functions

descriptors for them. The design postpones the removal of a segment from main memory for as long as possible, for reasons that are similar to the deactivation postponement. A process may DISABLE access to a segment, thus making it eligible to be swapped out, simply because it has run out of descriptor registers and wants to ENABLE access to some other segment. If all processes are not using their full memory quota or there is some in-core sharing of segments, then it may be possible to swap the second segment into main memory without removing the first. We choose not to swap out the first segment because the process may choose to reENABLE access to it, thus requiring its presence in main memory. The three functions that deal with the SWAP_CHAIN are LOCK (into main memory), FINDLOCK and UNLOCK. LOCK removes a segment from the chain, using FINDLOCK to find the segment's position in the chain, and UNLOCK adds a segment to it.

The final set of AS functions deals with the allocation and deallocation of main memory segments to virtual memory segments. Figure 29 gives the specifications of FINDFREE, ALLOCMEM, and FREEMEM. Free memory areas are on a chain ordered by block#. A free memory area is characterized by its block# - the address of its first byte - and its size. ALLOCMEM removes a free memory area from this chain and FREEMEM adds an area to it. FINDFREE searches the free chain looking for an area of a given size. If one is found, its block# is returned, otherwise, we have a problem. Assuming that the main memory quota mechanism is correct, then a free memory area of the desired size can be constructed by some combination of the following: 1) concatenating adjacent free areas; 2) splitting a free area into two free areas; 3) removing segments on the SWAP_CHAIN from main memory; and 4) compacting fragmented free areas. Determining the appropriate course of action requires a policy that does not belong in the kernel. Rather, the kernel designs assumes the existence of a process whose function is to keep the free memory chain in "good shape" - sufficient free areas of the right sizes. To perform this task the kernel provides it with functions that perform the three operations just described. These functions are discussed in a later subsection. While we expect that this process will be correctly implemented and be able to keep ahead of the kernel, the kernel must be prepared to deal with FINDFREE's failure to find a free area.

When FINDFREE fails the kernel can do one of two things: 1) it can explicitly cause the memory management process to run and only permit its three memory structuring kernel functions (that do not affect the current security state) to be invoked; or 2) it can deallocate the CPU from the current process, allocate the CPU to any other process ready to run (but somehow indicate that the memory management process should run), and allow all kernel functions to be invoked. If course 1) is chosen, then the original process can be

```
Function: FINDFREE
possible values: block#
Parameters: FINDFREE(block#, size)
Value:
IF block# = END_BLOCK#;
   THEN: RESTART;
   ELSE:
      IF MBT_SIZE(block#) = size;
         THEN: block#;
         ELSE: FINDFREE(MBT_CHAIN(block#, size));
      END;
END;

Function: ALLOCMEM
Parameters: ALLOCMEM(vblock#, block#)
Effect:
IF 'MBT_CHAIN'(vblock#) = block#;
   THEN: MBT_CHAIN(vblock#) = MBT_CHAIN(block#);
      MBT_FLAGS(block#) = ALLOCATED;
   ELSE: ALLOCMEM('MBT_CHAIN'(vblock#), block#);
END;

Function: FREEMEM
Parameters: FREEMEM(vblock#, block#)
Effect:
IF 'MBT_CHAIN'(vblock#) > block#;
   THEN: MBT_CHAIN(block#) = 'MBT_CHAIN'(vblock#);
      MBT_CHAIN(vblock#) = block#;
      MBT_FLAGS(block#) = FREE;
      MBT_ASTE(block#) = 0;
   ELSE: FREEMEM('MBT_CHAIN'(vblock#), block#);
END;
```

Figure 29. FINDFREE, ALLOCMEM and FREEMEM Functions

restarted at the point where FINDFREE was invoked, because the security state of the system has not changed. Course 2) is more flexible but it requires that we prove that the system is in a secure state at the point where FINDFREE fails. Also, the process must be restarted at the point where the external kernel function was invoked, because the security state of the system may have changed, thus invalidating security checks made before the original FINDFREE failure.

PROCESS COOPERATION

Mechanisms are provided to allow the sequential processes that coexist in the physical computer system to cooperate. These mechanisms are used within the kernel to insure its correct operation, and the kernel provides external functions that allow these mechanisms to be used in an arbitrary manner, subject only to security constraints. Two mechanisms are provided - a synchronization mechanism that employs semaphores and the P and V operations, and an interprocess communication (IPC) mechanism. The functions for these mechanisms do not change the security state of the system. They provide interpretive access to objects as permitted by the current state and, since they can cause the execution state of a process to change, they modify the representation of the current state.

P and V

P and V are synchronization primitives that operate on semaphores. They are the basic synchronization primitives used within the kernel, and an explanation of them is given in Appendix I. In the specification, synchronization with the disk during segment swapping is achieved by performing a P on the disk semaphore. When the disk operation completes, the interrupt handler does a V on the disk semaphore.

To allow users to synchronize with each other the kernel associates a semaphore with each segment in the virtual memory. Processes that have write access to a segment may P and V on the associated semaphore. Write access is required because both P and V modify the semaphore, and the execution state of a process may change as a result of the P or V. It is assumed that users will use P's and V's to coordinate the modification of shared segments and to synchronize with their terminal I/O. The user may P on the I/O segment associated with his terminal. An interrupt from the terminal will cause the V. Whether or not the appropriate conventions are followed to insure the cooperation desired by users, is, of course, no concern of the kernel.

The specification of the P function is given in Figure 30. P decrements the semaphore counter and, if the result is negative, it blocks the process and adds it to the queue of processes blocked on the semaphore. PSWAP is invoked when a process becomes blocked in order to allocate the CPU to some other process.

The function of PSWAP is to deallocate the CPU from the current process and reallocate it to some other process. This other process must be in the READY state. It is possible, that when PSWAP first looks, it cannot find a READY process. In this case, it waits for an I/O interrupt (which always results in a V on a I/O segment semaphore), and then looks again. At the level of the specification, it is sufficient to change the value of TCP to any process that is READY. At the implementation level, more work may be required depending upon specific hardware characteristics. On the PDP-11/45 it is necessary to unload registers associated with the execution of the current process and reload them for the new process. This save/restore operation must, of course, be done correctly to insure security. The specification assumes that the contents of the hardware descriptor registers are equal to PS_SDR and PS_SAR for the current process.

If more than one process is READY, PSWAP must have some algorithm for selecting a particular process to run. This topic is discussed in the subsection on policy functions. For the time being we assume that PSWAP has some way of selecting a new process to run.

The V function (Figure 31) is the inverse of P. It increments the counter of a semaphore, and if the result is non-positive, makes a process that was blocked on the semaphore ready. If more than one process is blocked on the semaphore, VEND finds the process that has been blocked the longest, and VUNCHAIN removes it from the queue. PSWAP is invoked because a process that was blocked is now ready and PSWAP may want to allocate the CPU to it.

Interprocess Communication

Although the P and V primitives are probably sufficient for implementing any desired form of process synchronization, another mechanism, interprocess communication (IPC), is provided. The utility of IPC is that it allows the transfer of data between processes, and the receiving process is provided with the identification of the sending process.

Figure 32 shows the specification of IPCSEND, the first half of an IPC sequence. The security requirement for IPC is that a process can only send a message to another process at an equal or higher

```
Function: KP
Parameters: KP(aste#)
Effect:
IF AST_WAL(aste#, TCP);
   THEN: P(aste#);
   ELSE: RC(TCP) = NO;
END;

Function: P
Parameters: P(smfr#)
Effect:
SMFR_COUNT(smfr#) = 'SMFR_COUNT'(smfr#) - 1;
IF SMFR_COUNT(smfr#) < 0;
   THEN: PT_FLAGS(TCP) = BLOCKED;
      PT_LINK(TCP) = 'SMFR_POINTER'(smfr#);
      SMFR_POINTER(smfr#) = TCP;
      PSWAP;
END;
RC(TCP) = YES;

Function: PSWAP
Parameters: PSWAP
Effect:
IF (∃process#) (PT_FLAGS(process#) = READY);
   THEN: TCP = process#;
   ELSE: WAIT;
      PSWAP;
END;
```

Figure 30. KP, P and PSWAP Functions

```
Function: KV
Parameters: KV(aste#)
Effect:
IF AST_WAL(aste#, TCP);
   THEN: V(aste#);
   ELSE: RC(TCP) = NO;
END;

Function: V
Parameters: V(smfr#)
Effect:
SMFR_COUNT(smfr#) = 'SMFR_COUNT'(smfr#) + 1;
IF SMFR_COUNT(smfr#) <= 0;
   THEN:
      IF SMFR_COUNT(smfr#) = 0;
         THEN: Let process# = 'SMFR_POINTER'(smfr#);
            SMFR_POINTER(smfr#) = 0;
         ELSE: Let process# = VEND;
            VUNCHAIN('SMFR_POINTER'(smfr#));
      END;
      PT_FLAGS(process#) = READY;
      PSWAP;
END;
RC(TCP) = YES;

Function: VEND
possible values: process#
Parameters: VEND(process#)
Value:
IF 'PT_LINK'(process#) = 0;
   THEN: process#;
   ELSE: VEND('PT_LINK'(process#));
END;

Function: VUNCHAIN
Parameters: VUNCHAIN(process#)
Effect:
IF 'PT_LINK'('PT_LINK'(process#)) = 0;
   THEN: PT_LINK(process#) = 0;
   ELSE: VUNCHAIN('PT_LINK'(process#));
END;
```

Figure 31. KV, V, VEND and VUNCHAIN Functions

```
Function: IPCSEND
Parameters: IPCSEND(process#, message, domain)
Effect:
IF (PT_FLAGS(process#) ≠ INACTIVE) &
   (((PS_CUR_CLASS(process#) ≥ PS_CUR_CLASS(TCP)) &
   (PS_CUR_CAT(process#) ⊇ PS_CUR_CAT(TCP))) |
   (PT_TYPE(TCP) = TRUSTED)) &
   ('PT_IPC_QUOTA'(process#) ≠ 0);
   THEN: Let ipce# = 'IPC_LINK'(0);
      IPC_LINK(0) = 'IPC_LINK'(ipce#);
      IPC_LINK(ipce#) = 0;
      IPC_PROCESS(ipce#) = TCP;
      IPC_DOMAIN(ipce#) = domain;
      IPC_DATA(ipce#) = message;
      IF 'PT_IPC_QUEUE_HEAD'(process#) = 0;
         THEN: PT_IPC_QUEUE_HEAD(process#) = ipce#;
         ELSE: Let eipce# =
                  FINDIPCEND('PT_IPC_QUEUE_HEAD'(process#));
            IPC_LINK(eipce#) = ipce#;
      END;
      PT_IPC_QUOTA(process#) = 'PT_IPC_QUOTA'(process#) - 1;
      IF 'PT_IPC_WAIT'(process#) = ON;
         THEN: PT_IPC_WAIT(process#) = OFF;
            PT_FLAGS(process#) = READY;
            PSWAP;
      END;
END;

Function: FINDIPCEND
possible values: ipce#
Parameters: FINDIPCEND(ipce#)
Value:
IF IPC_LINK(ipce#) = 0;
   THEN: ipce#;
   ELSE: FINDIPCEND(IPC_LINK(ipce#));
END;
```

Figure 32. IPCSEND and FINDIPCEND Functions

security level, unless the sending process is a trusted subject. In this case there are no security requirements. The object in an IPC sequence is an IPC element. The basic functions of IPCSEND is to allocate an IPC element from the free pool, fill it in with the data being transferred and sending process identification, and add it (using FINDIPCEND) to the queue of elements waiting to be received by the receiving process. The process identification includes a domain indicator to allow the receiving process to distinguish between messages originating in the kernel domain of some other process and messages originating in the user domains.

On the receiving side there are two cases: 1) the receiving process is blocked because it is waiting for a message and until this IPCSEND there were none available; and 2) the receiving process is not waiting for a message. For case 1) the receiving process becomes ready and PSWAP is invoked to allow the CPU to be reallocated to it, if that action is dictated by PSWAP's CPU allocation policy.

Since IPC elements are a finite, shared resource, it seems reasonable to control allocation of them with a quota mechanism. Intuitively, one might think that the quota would be imposed on the sending process in an IPC sequence. When a process sent a message, its quota would be debited; when a message was received and the IPC element was returned to the free pool, the sending process's quota would be credited. The problem with this approach is that an action by the receiving process, which may be at a higher security level than the sending process, modifies the quota of the sending process. The sending process could determine if its quota had gone to zero by trying to send another message to a second process at the same security level and observing a segment shared with this second process to see if the message is actually sent. Without giving all of the details, we hope that the reader can see that this quota implementation would not be secure, because a higher level process could "signal" a lower level process.

An alternative quota implementation is to debit the quota of the receiving process when some other process sends it a message, and credit the receiving process when it actually receives the message. As the specification shows, this is the method used in the 11/45 design. If the IPC quota of some process has gone to zero then no other process can send it any messages. This is not a security problem, because a process can only determine if its IPCSEND was successful or not if it is sending to a process at the same security level.

The second half of an IPC sequence occurs when a process invokes IPCRCV (IPC receive, see Figure 33). Again there are two cases: 1) there are one or more messages waiting for the process; and 2) there

```
Function: IPCRCV
Parameters: IPCRCV
Effect:
IF 'PT_IPC_QUEUE_HEAD'(TCP) = 0;
   THEN: PT_IPC_WAIT(TCP) = ON;
      PT_FLAGS(TCP) = BLOCKED;
      PSWAP;
      IPCRCV2;
   ELSE: IPCUNQUEUE;
END;

Function: IPCUNQUEUE
Parameters: IPCUNQUEUE
Effect:
Let ipce# = 'PT_IPC_QUEUE_HEAD'(TCP);
PT_IPC_QUEUE_HEAD(TCP) = 'IPC_LINK'(ipce#);
RC(TCP) = IPC_PROCESS(ipce#), IPC_DOMAIN(ipce#), IPC_DATA(ipce#);
IPC_LINK(ipce#) = 'IPC_LINK'(0);
IPC_LINK(0) = ipce#;
PT_IPC_QUOTA(TCP) = 'PT_IPC_QUOTA'(TCP) + 1;

Function: IPCRCV2
Parameters: IPCRCV2
Effect:
IF 'PT_IPC_QUEUE_HEAD'(TCP) ≠ 0;
   THEN: IPCUNQUEUE;
END;
```

Figure 33. IPCRCV, IPCUNQUEUE and IPCRCV2 Functions

are no messages waiting for the process. In case 1) IPCUNQUEUE unqueues the IPC element that has been queued the longest, moves its contents to the process's RC object, returns the IPC element to the free pool, and credits the process's quota. In case 2) the process becomes blocked until a message is received. At that time IPCRCV2 is invoked to receive the message.

POLICY FUNCTIONS

In any system it is desirable to separate policy and mechanism. This is particularly true in secure systems, where the size and complexity of the kernel must be minimized. The kernel must contain the mechanisms for implementing the elements of the system and the security policy for controlling access to these elements. Any policy that influences the allocation of physical resources need not, and should not, be in the kernel. The actual allocation of resources must be performed by the kernel in a secure and correct manner. For these reasons it is necessary to have external kernel functions that communicate policy decisions made outside the kernel to the implementation mechanisms within the kernel. The effect of all of these functions is to simply change the representation of the current security state.

Memory Control

The 11/45 kernel design views main memory as a series of fixed sized blocks. The size of a block must be a multiple of the minimum segment size supported by the 11/45's MMU (64 bytes). The initial implementation uses 256 byte blocks. Adjacent blocks can be combined into main memory segments. Characteristics of a main memory segment include: 1) the address of its first block; 2) its size; and 3) the virtual memory segment bound to it, if any. The first block of a main memory segment is either FREE or ALLOCATED, all other blocks are CONCATENATED.

The kernel design assumes the existence of a memory control function with the task of keeping the free memory chain in "good shape" - sufficient free main memory segments of different sizes so that the FINDFREE subfunction of SWAPIN always succeeds. In order for the memory control function to operate properly it must be able to observe the state of main memory, decide how it should be changed, if at all, and communicate its decision to the kernel. To make the necessary observations it must have read access to the MBT (Memory Block Table) and AST. These tables have a security level of system high, because they contain system-wide information on the mapping of virtual resources into physical resources. Therefore, the memory control function cannot be distributed among all process - it must be

isolated in a process at a system-high security level. There is no need for this process to be trusted, because the kernel does not depend on its correct operation, and this process can only communicate with other system-high processes. If the process does not operate properly the system may fail, but there will be no security compromise.

Figure 34 shows the functions that the memory control process can use to communicate its policy decisions to the kernel. One of these functions, KSWAPOUT, directs the kernel to swap a segment out of main memory. Before invoking SWAPOUT the kernel insures that the specified segment is eligible to be swapped out. Note that the kernel does not check the identity of the process that invokes KSWAPOUT. The only security requirement is that the data base that must be observed in order to make intelligent use of the memory control functions has a security level of system high. Since the kernel makes no assumptions about the correct use of the memory control functions, there is no need for it to check the invoking process's identity. For practical reasons the operating system may choose to prevent user processes from using these functions.

It may not be necessary to swap a segment out of main memory to make room for another - changing the size of free main memory segments may be sufficient. CONCAT and SPLIT are two functions for performing this operation. The parameter of CONCAT is the block# of a main memory segment. The kernel requires that this segment be free and that the next segment in the free chain be adjacent to it. The two segments are then concatenated into a single free segment. The parameters of SPLIT are a block# and size. The kernel requires that the block# identify a free main memory segment whose size is greater than the size parameter. It then splits the segment into two segments; the size of the first is equal to the size parameter.

In any system where memory is dynamically allocated to and deallocated from different sized elements, fragmentation can be a problem [Knuth]. During kernel operation, it is possible that there may be enough free memory for a segment that must be swapped in but there is no combination of KSWAPOUTs, CONCATs, and SPLITs that can form a free main memory segment of the proper size. The problem can only be solved by reallocating virtual memory segments to main memory segments. Virtual memory segments, locked in or not, can be moved from one area of main memory to another because the memory management unit prevents the use of physical addresses. In fact, the only places that physical addresses need occur are in the AST and segment descriptors. Thus a function could be provided to physically move a segment from one area of main memory to another and make the necessary corrections to stored addresses. A specification for this function is not given because it may not be necessary for all systems

```
Function: KSWAPOUT
Parameters: KSWAPOUT(block#)
Effect:
IF (MBT_FLAGS(block#) = ALLOCATED) &
   (AST_LOCK(MBT_ASTE(block#)) = UNLOCKED);
   THEN: SWAPOUT(MBT_ASTE(block#));
END;

Function: CONCAT
Parameters: CONCAT(block#)
Effect:
Let next_block# = MBT_CHAIN(block#);
IF (MBT_FLAGS(block#) = FREE) &
   ('MBT_SIZE'(block#) + block# = next_block#);
   THEN: MBT_SIZE(block) = 'MBT_SIZE'(block#) +
         MBT_SIZE(next_block#);
      MBT_CHAIN(block#) = MBT_CHAIN(next_block#);
      MBT_FLAGS(next_block#) = CONCATENATED;
END;

Function: SPLIT
Parameters: SPLIT(block#, size)
Effect:
IF MBT_FLAGS(block#) = FREE) &
   (MBT_SIZE(block#) > size);
   THEN: Let new_block# = block# + size;
      MBT_FLAGS(new_block#) = FREE;
      MBT_SIZE(new_block#) = 'MBT_SIZE'(block#) - size;
      MBT_SIZE(block#) = size;
      MBT_CHAIN(new_block#) = 'MBT_CHAIN'(block#);
      MBT_CHAIN(block) = new_block#;
END;
```

Figure 34. KSWAPOUT, CONCAT and SPLIT Functions

to be built on the PDP-11/45 kernel.

Before going on to process control it should be noted that the need for the memory control functions and software to use them is a result of implementing multiple sizes of unpaged segments. If all segments were a single size or composed of fixed sized pages (assuming hardware support for segmentation and paging), then memory allocation would be much simpler.

Process Control

Since our definition of a security compromise does not include denial of service, it is not necessary for the process scheduling policy to be implemented within the kernel. Scheduling decisions can be made outside of the kernel and the results communicated to PSWAP, the process multiplexor, by suitable external kernel functions. The actual mechanisms used are somewhat arbitrary and should depend upon specific system requirements.

As an example, let us postulate that PSWAP implements a policy of allocating the CPU to the ready process with the highest priority, and within groups of processes with equal priorities the CPU is multiplexed in a round-robin fashion. This policy seems simple enough to justify implementation within the kernel - a dozen higher level language statements should be sufficient. To meet system requirements it may be necessary to dynamically adjust process priorities. This requirement can be met by having the kernel assume the existence of a scheduler and providing it with a function to change process priorities.

There are at least two ways to implement a scheduler: 1) the scheduler can be distributed among all processes; or 2) it can be isolated in a process of its own. If it is distributed, then the scheduler can make decisions about a process's priority based only upon that process's behavior. If the process is in a highly interactive phase, the scheduler may choose to give it a high priority, and if it is compute bound (doing a large compilation, for example) it may have a low priority. If scheduling decisions are to be based on the relative behavior of all processes, then the scheduler must be isolated in a single process, because the information it must observe is at system high. This scheduler would insure that it is scheduled with a certain frequency, and each time it ran it would observe recent system behavior and adjust priorities appropriately. It is also possible to include time-slicing in this approach. The scheduler indicates to the kernel what processes are to be eligible for time slicing and the appropriate time quantums. The actual management of the interval timer would be by PSWAP.

The point of this discussion is to show that since our definition of security does not have any implications on scheduling policy, scheduling can be done in a variety of ways. By selecting appropriate kernel functions it is possible to separate the process multiplexing mechanism from scheduling policy.

INPUT/OUTPUT

Up to this point the design details include no explicit provision for I/O. The reason for this apparent omission is that I/O can be entirely transparent to the security kernel.[17] If we postulate a system where all devices operate in a unilevel mode and the attachment of I/O devices to processes is performed at system initialization, then there is nothing more to do. Since the MMU will enforce the controlled access to the I/O devices, and the unilevel operation does not require computer generated security labels, all of the I/O can be performed entirely by uncertified software.

If requirements demand a more flexible system environment, then it may be necessary to introduce additional kernel functions. For example, if we wish to multiplex the line printer among different security levels but retain unilevel printer operation, it will be necessary to perform a security reconfiguration on the printer each time a level change is desired. Externally, this reconfiguration may just be a change of the printer forms. Internally, it will be necessary to change the security level of the segment that contains the printer's control registers to reflect the security reconfiguration. The kernel can provide a function to change the security level of a segment, and it can insure that changes are only made that keep the system in a secure state (in terms of triples in b, the *-property, and the compatibility requirement), but the kernel cannot determine if the change of security level is "appropriate". Thus the kernel must restrict the use of this change function to trusted processes, and trusted processes that use it must be certified to use it correctly - for example, to assure that the change of security level of the printer control segment is coordinated with the change of forms.

If we wish to avoid reconfigurations and operate the printer in a multilevel mode, then it will be necessary to run the printer with

---

[17] By I/O we really mean external I/O - the transfer of information into and out of the computer system. Internal I/O is used in implementing the virtual memory and is completely controlled by the security kernel. Also, we are only allowing non-DMA devices to be used for external I/O.

a certified process. Since only the printer process knows the security attributes of the information that it is printing, this process must produce security labels on the documents it prints. The process must be certified to produce correct labels that cannot be altered or "spoofed" by uncertified software.

The requirements for terminals are similar. If a terminal is to be used at different levels, then, when a user specifies the security level at which he wishes to operate, there must be a mechanism to guarantee that he is talking to certified software, and not to uncertified software spoofing certified software. One way to implement such a mechanism is to use terminals that generate a unique interrupt when they are powered on, and to vector this interrupt into the kernel. Thus, if the user turns the terminal on before logging in, we can guarantee that he is talking to a certified logger.

SUMMARY

In this section we have presented the kernel primitives that will support a static system. In a static system all software configuration decisions, including the security level of shared I/O devices (printers, card readers, etc.), and non-shared devices (terminals), and the binding of users to processes, are made at system initialization. The kernel, of course, depends upon the initial state of the system being secure. Although initialization is beyond the scope of this paper, secure initialization simply requires that all of the triples in the initial b are correct with respect to the security condition and the *-property, and that the initial hierarchy is compatible. User requirements will determine what the actual initial state of a particular system is.

In most systems a static software configuration will not be acceptable - at a minimum it will be necessary to permit users to log on and off the system. This feature must be supported by kernel primitives that bind/unbind users to/from processes. This action includes the initialization of processes - an operation whose security requirements are similiar to system initialization. Thus, a complete kernel design specification for a dynamic system will include functions for initializing and terminating user processes. The actual initial state of a user process will probably depend upon the requirements of the specific system.

SECTION V

SUMMARY

In this paper we have presented the design of a secure system kernel for the PDP-11/45. The kernel design is based on a general-purpose mathematical model of secure computer systems. This section summarizes the accomplishments and limitations of the design to date.

DESIGN LIMITATIONS

Although the design is based on a proven model of security, we have not yet proved that the design corresponds to the model. Thus, there may be errors in the design. A methodology for proving that the design and implementation representations of a kernel correspond to the model has been developed [Bell and Burke]. This methodology has been applied to part of this kernel design, and the results demonstrate the validity of the proof approach and the correctness of the relevant parts of the design. We are confident that any errors in the design are not fundamental problems and can be easily corrected.

There is one aspect of the model, however, that in extreme cases could be viewed as a fundamental problem. The model is based on an asynchronous view of computation. Thus it is possible for programs executing outside of the security kernel to influence the response time that other programs see, and to use this ability to modulate response time to send "Morse code" [Lampson]. We feel that the presence of this uncontrolled communication channel is an intrinsic problem, but not a serious one because: 1) the kernel can reduce the bandwidth of the channel to any desired value by adding noise; 2) the use of this channel to pass information at one security level to a lower level requires cooperating processes at both levels that are able to synchronize with each other; and 3) if we have solved all other problems we have made a great deal of progress in computer security.

The treatment of hardware in this paper has been limited to a discussion of characteristics that the kernel depends on. Two other aspects of hardware are important - its correctness and the possibility of component failure. By correctness we mean the correspondence of the actual hardware to a formal specification that describes its behavior. The part of the hardware that the kernel depends on, access controls and many instructions, must be correct. An error in the hardware will have the same effect as an error in the

```
Function: CONNECT
Parameters: CONNECT(process#, daste#, entry#, mode)
Effect:
IF 'PS_SEG'(process#, 0) = 0;
   THEN: RC(process#) = NO;
   ELSE: Let flag = 'HASH'(DIR_DISK(daste#, entry#));
      IF (flag ≠ 0) &
         'AST_CPL'(flag, process#);
         THEN: RC(process#) = NO;
         ELSE:
            IF flag ≠ 0;
               THEN: Let aste# = flag;
                  IF 'AST_AGE'(aste#) = AGED;
                     THEN: UNAGE(aste#);
                  END;
               ELSE: ACTIVATE(daste#, entry#);
                  Let aste# = HASH(DIR_DISK(daste#, entry#));
                  UNAGE(aste#);
            END;
            AST_CPL(aste#, process#) = TRUE;
            IF mode = WRITE;
               THEN: AST_WAL(aste#, process#) = TRUE;
            END;
            Let seg# = 'PS_SEG'(process#, 0);
            PS_SEG(process#, 0) = 'PS_SEG'(process#, seg#);
            PS_SEG(process#, seg#) = aste#;
            PS_SEG_INUSE(process#, seg#) = TRUE;
            RC(process#) = YES, seg#;
      END;
END;
```

Figure 22. CONNECT Function

software security controls - it will allow repeated and undetectable penetration of the system. Component failure, on the other hand, is a probabilistic event. The probability that component failure will allow a security compromise can be reduced by adding redundancy, but never eliminated. The impact of component failure on computer security should be addressed by future research.

One final comment on hardware: the design considers only single CPU (central processing unit) systems. Support for multiple CPU's would add complexity to level 1, but could be accomplished with existing mechanisms (P and V primitives).

ACCOMPLISHMENTS

The principal achievement of this work is a feasible design for computer systems that can be proven to implement an abstract model of the Department of Defense Security Policy. The model and design provide a high degree of confinement of the actions of arbitrary (uncertified) programs. Included in the design is a clean handling of user I/O. Although the features provided by the design are in some sense arbitrary (for example, another design might do without the IPC objects), the security controls are in no way ad hoc - they can be proven effective in a rigorous, mathematical manner.

In summary, this paper demonstrates the soundness of the security kernel approach to solving the computer security problem by presenting a prototype kernel design. The work ahead includes designing a kernel for a large scale system with specific requirements in such a way that the impact of the kernel on efficiency is acceptable, and finding new hardware architectures that facilitate secure system development.

W. Lee Schiller
Intelligence and
Information Systems

APPENDIX I

SYNCHRONIZING PRIMITIVES

The systems to be built on the PDP-11/45 will be composed of parallel sequential processes, and consequently, primitives for synchronization are required. The primitives we have chosen are Dijkstra's P and V operations. This appendix provides background for understanding them. (Some of this material has been taken from [Dijkstra (1)] and [Horning & Randell]).

The P and V primitives operate on special purpose integer variables called "semaphores". Semaphores are usually initialized with the value 0 or 1. A P operation decreases the value of a semaphore by 1. If the resulting value of the semaphore is non-negative, the process performing the P can continue; if, however, the resulting value is negative the process becomes blocked and is placed on a queue associated with the semaphore. Until further notice in the form of a V operation on the same semaphore by some other process, the dynamic progress of the first process is not logically permissible and a processor will not be allocated to it.

A V operation increases the value of a semaphore by 1. If the resulting value of the semaphore is positive, the V has no further effect; if, however, the resulting value is non-positive, one of the processes on the semaphore's waiting queue is removed - its dynamic progress is again logically permissible and a processor may be allocated to it.

Several observations can be made from these definitions. If a semaphore value is non-positive its absolute value equals the number of processes on its waiting queue. P and V operations must be "indivisible actions" - they cannot occur "simultaneously" in parallel processes. When a V causes a process to be removed from a semaphore's waiting queue it is undefined - logically immaterial - which process (if more than one is waiting) is actually removed. (In the 11/45 kernel implementation the process waiting the longest will be removed.) Finally, a consequence of the P and V synchronization mechanism is that a process whose dynamic progress is permissible can only lose that status by actually progressing - by performing a P.

Semaphores can be used in two different ways. The first is mutual exclusion - the protection of a critical section of program code or data - and it requires a semaphore, initialized to 1, for each critical section. If all processes precede entry to a critical section with a P on that section's semaphore, and perform the corresponding V on exit, them it can easily be shown that two or more

processes can never be in the critical section simultaneously. The second use of semaphores is to synchronize "producer-consumer" relationships among processes. When a consumer requires a resource it performs a P; the corresponding V is performed by a producer when it makes a resource available. The correct initialization of the semaphore (usually to 0) insures that the consumers do not get ahead of the producers. It should be pointed out that although the use of P's and V's facilitates the demonstration of correctness, their use does not guarantee correctness. The appropriate conventions for using the system's semaphores must be established, and these conventions must be followed by the cooperating sequential processes.

BIBLIOGRAPHY

ANDERSON, J. P., Computer Security Technology Planning Study, ESD-TR-73-51, Volume I, October, 1972.

BELL, D. E., Secure Computer Systems: A Refinement of the Mathematical Model, ESD-TR-73-278, Volume III, MITRE Corporation, June, 1974.

BELL, D. E. & BURKE, E. L., A Software Validation Technique for Certification, Part 1: The Methodology, MTR-2932, MITRE Corporation, November, 1974.

BELL, D. E. & LAPADULA, L. J., Secure Computer Systems: Mathematical Foundations, ESD-TR-73-278, Volume I, MITRE Corporation, November, 1973.

BENSOUSSAN, A., CLINGEN, C. T., & DALEY, R. C., "The Multics Virtual Memory: Concepts and Design," Communications of the ACM, Volume 15, Number 5, May, 1972, 308-318.

BRANSTAD, D., "Privacy and Protection in Operating Systems," Computer, Volume 6, Number 1, January, 1973, 43-47.

BURKE, E. L., Concept of Operation for Handling I/O in a Secure Computer at the Air Force Data Services Center (AFDSC), ESD-TR-74-113, MITRE Corporation, April, 1974.

DIGITAL, PDP-11/45 Processor Handbook, Digital Equipment Corporation, 1973.

DIJKSTRA(1), E. W., "The Structure of the "THE" Multiprogramming System," Communications of the ACM, Volume 11, Number 5, May, 1968, 341-346.

DIJKSTRA(2), E. W., "The Humble Programmer," Communications of the ACM, Volume 15, Number 10, October, 1972, 859-866.

ESD, Computer Security Development Summary, MCI-74-1, Directorate of Information Systems Technology, Electronic Systems Division, December, 1973.

HORNING, J. J., & RANDELL, B., "Process Structuring," Computing Surveys, Volume 5, Number 1, March, 1973, 5-30.

KARGER, 2Lt. P. A., & SCHELL, Major R. R., Multics Security Evaluation: Vulnerability Analysis, ESD-TR-74-193, Volume II, Electronic Systems Division, June, 1974.

KNUTH, D. E., The Art of Computer Programming, Volume I: Fundamental Algorithms, Reading, Mass.: Addison-Wesly, 1969, 435-451.

LAMPSON, B. W., "A Note on the Confinement Problem," Communications of the ACM, Volume 16, Number 10, October, 1973, 613-615.

LAPADULA, L. J., & BELL, D. E., Secure Computer Systems: A Mathematical Model, ESD-TR-73-278, Volume II, MITRE Corporation, November, 1973.

LIPNER, S. B., Computer Security Research & Development Requirements, MTP-142, MITRE Corporation, February, 1973.

LISKOV, B. H., "The Design of the Venus Operating System," Communications of the ACM, Volume 15, Number 3, March, 1972, 144-149.

MOGILENSKY, J., A General Security Marking Policy for Classified Computer Input/Output Material, private communication, April, 1974.

NEUMANN, P. G., FABRY, R. S., LEVITT, K. N., ROBINSON, L., & WENSLEY, J. H., "On the Design of a Provably Secure Operating System," International Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France, August, 1974, 161-175.

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

PARNAS, D. L., "A Technique for Software Module Specification with Examples," Communications of the ACM, Volume 15, Number 5, May, 1972, 330-336.

PRICE, W. R., Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems, Ph.D. Thesis, Carnegie-Mellon University, June, 1973.

SALTZER(1), J. H., Traffic Control in a Multiplexed Computer System, MAC-TR-30 (Thesis), Project MAC, July, 1966.

SALTZER(2), J. H., "Protection and the Control of Information in Multics," Communications of the ACM, Volume 17, Number 7, July, 1974, 388-402.

SCHILLER, W. L., Design of a Security Kernel for the PDP-11/45, ESD-TR-73-294, MITRE Corporation, December, 1973.

SMITH, L., Architectures for Secure Computer Systems, ESD-TR-75-51, MITRE Corporation, April, 1975.

WEISSMAN, C., "Security Controls in the ADEPT-50 Time-Sharing System," AFIPS Proceedings FJCC, 1969, 119-133.

DISTRIBUTION LIST

INTERNAL

D-70
W. S. Attridge
J. J. Croke
J. W. Shay

D-72
L. J. LaPadula

D-73
S. R. Ames, Jr.
N. H. Anschuetz
D. E. Bell
E. H. Bensley
E. J. Bertrand
K. J. Biba
R. H. Bullen, Jr.
E. L. Burke (10)

C. Engelman
M. Gasser
S. M. Goheen
S. R. Harper
J. F. Jacobs
E. L. Lafferty (2)
J. A. Larkins
S. B. Lipner (20)
J. L. Mack
D. L. Martell
J. K. Millen
G. H. Nibaldi
R. D. Rhode
J. M. Schacht
W. L. Schiller (10)
H. S. Stone
D. F. Stork
B. N. Wagner
J. C. C. White (10)
R. C. Yens

D-75
R. S. Gardella

EXTERNAL

Headquarters
Electronic Systems Division
Hanscom Air Force Base
Bedford, Massachusetts 01731

MCIT
Col. F. Emma
Lt. Col. R. Park
Maj. T. Bailey
Maj. L. Noble
Maj. R. Schell
1Lt. P. Karger
1Lt. W. Price
2Lt. W. Austell