74-4721

JONES, Anita Katherine, n.d.
PROTECTION IN PROGRAMMED SYSTEMS.

Carnegie-Mellon University, Ph.D., 1973
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PROTECTION IN PROGRAMMED SYSTEMS

Anita Katherine Jones

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
June, 1973

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) monitored by the Air Force Office of Scientific Research, and in part by the Univac Division of the Sperry Rand Corporation. This document has been approved for public release and sale; its distribution is unlimited.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Carnegie-Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

AND

MELLON INSTITUTE OF SCIENCE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE Protection in Programmed Systems

PRESENTED BY Anita Katherine Jones

ACCEPTED BY THE DEPARTMENT OF Computer Science

MAJOR PROFESSOR — DATE

DEPARTMENT HEAD — 8/8/73 DATE

APPROVED BY THE COLLEGE COUNCIL

DEAN — 14 August 1973 DATE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

This dissertation investigates the control of access to objects within programmed systems. The vehicle for this study is a model of protection that isolates a small set of mechanisms needed to provide access control, leaving the policy for invoking these mechanisms to vary naturally with applications. Emphasis is placed on access control required for parameters that accompany a process crossing between execution environments; and a new concept called amplification is defined.

The model is shown to provide structure and terminology sufficient for describing and comparing diverse protection systems, for expressing and proving boundary conditions that characterize the manipulation of objects within environments independent of the code executed, and for partially ordering protection systems according to the services they provide. In addition, the dissertation introduces the concept of a centralized protection facility capable of providing access control for user defined objects and accesses.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

I wish to express my deep appreciation to my thesis advisor, Nico Habermann, for his unfailing encouragement, his guidance and his perceptive judgments, and to the rest of my committee, Bill Wulf, Allen Newell and John Grason, for their criticisms and support.

I feel extremely fortunate in having had the opportunity to contribute some of the 'thesis ideas' to an actual system design during the progress of the thesis, and to learn from arguing about those ideas with Bill Wulf and the members of the initial HYDRA design group.

Thanks also to the members of the 'brownie-plate' seminar.

It is a pleasure to acknowledge these debts!

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Abstract ii
Acknowledgements iii
Table of Contents iv

Chapter I: A Model of Protection 1
Limits of the Protection Problem--a View from a Larger Context 2
Background 3
A Model 4
A Theory of Protection 5
Organization of the Thesis 12

Chapter II: Enforcement of Protection 14
Environment Representations 17
Execution Site Enforcement 20
Object Site Enforcement 23
Access Site Enforcement 25
Interleaving of Name Interpretation and Protection Checking 26
Time at Which Protection Checks can be Performed 27
Summary 29

Chapter III: Right Transfer Primitives 30
Summary 38

Chapter IV: Environment Binding 40
No Parameterization 42
Passed Parameters 43
Amplified Parameters 44
Protection and Execution 53
Environment Boundary Crossing Policies 56
Execution Control by Nested Procedure Invocation 57
Summary 60

Chapter V: Application of the Protection Model 61
Isolation 61
Conservation of Rights 63
Sharing 68
Additional Examples 72

Chapter VI: The Boundaries of Execution 77
Protection States 79
Access History of File Objects 85
Extending the File Problem 88
Complex File Access History 90
Memoryless Execution 95
Summary 99

Chapter VII: A Protection Facility 101
Implications of Dynamic Type Creation 101

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Types as Objects 103
Maintenance of Type-objects 104
Constituent Rights Revisited 107
Constituent Rights and Amplification 110
Summary 112

Chapter VIII: Suitability Measure 113
Measuring Accuracy 114
Protection System Comparison 118
Summary 122

Chapter IX: Conclusion 124
Summary 124
Future Research 126

Appendix A: Notation 129
Appendix B: Index of Definitions 132
Appendix C: Comparing Protection Features 133
Bibliography 137

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A MODEL OF PROTECTION

Chapter I

A programmed system is a set of algorithms implemented in software and hardware, mapping values input to that system to output values. Protection is the enforcement of rules to maintain order during the mapping activity. To be more precise the mapping activity is described as a set of distinguishable processes each performing a sequence of algorithms defined in the programmed system. The processes share the use of data retaining objects. At any time each process is aware of a set of objects and the specified ways each object can be used. Order is maintained and interactions are made limited and predictable by rules which regulate the acquisition and use of access to objects. The enforcement of these rules is called **protection.**

Though particular protection goals differ between systems, several commonly espoused goals allow

1. concurrent users to cooperate, yet not unnecessarily 'interfere' with one another
2. absent users to store data permanently, so that it is inaccessible to other users without authorization
3. a user to be protected from himself: resources and powers a user employs while performing one task are not available for misuse when he is engaged in performing another.

This thesis performs a systematic study with the intent of furthering a general theory of protection in programmed systems. The thesis

1. isolates and analyzes the notions and structure fundamental to protection

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. provides a parameterized model and descriptive terminology sufficient to describe and compare a large class of protection policies found in both operating systems and programming languages

3. analyzes properties of certain classes of protection regimes and characterizes some common problems that can be solved through use of protection policies

4. provides a yardstick so that incomparable protection implementations can be compared functionally

5. provides some insight into the properties of protection enforcement mechanisms.

Limits of the protection problem--a View from a Larger Context

Protection is a part of a larger concern, called security, that encompasses the regulation of activity both internal and external to the programmed system. A secure computer system depends upon

1. controlled access to objects within the programmed system--the topic addressed within this thesis

2. reliable hardware components

3. prevention of threats perpetrated outside the programmed system, for example, monitoring or tapping communication lines, theft of data volumes, or illegal entry into the programmed system through falsified identification.

To isolate protection from the other components of the security problem, we assume that all physical resources used to implement a computer system are completely reliable and free from external tampering. We assume infallible verification of the identity of users requesting service from the system. Since the protection component of security addresses controlling access to entities within a closed universe assumed to be free from any external influences, it can be studied separately.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Background

In early computers the lifetime of the software system was the duration of the execution of one user program. Though user programs could reference all of memory during execution, users were isolated from one another since memory was reloaded for each user execution.

The later development of operating systems to multiplex physical resources under software control, rather than external human control, required that the operating system also be responsible for protecting those multiplexed resources. For example the operating system which supports multiprogramming through simultaneous existence of multiple processes, must enforce controlled access to memory by those processes. Similarly, protection was required when operating systems were augmented to provide software defined and maintained resources such as files. Access to a file must be controlled not only during its active use, but during the absence of all authorized users. In this thesis we will refer to the protection components of programmed systems as protection systems. Each is composed of a policy and a set of mechanisms built from hardware and software, to enforce and implement the policy.

Protection appears in higher level languages as well as in operating systems. As programming languages moved from machine level toward languages more natural for human expression, 'scope rules' controlling when variables could be referenced were introduced. Their main purpose is to provide structure for optimization of storage usage; but they act as a protection policy for the use of variable objects as well. Thus the compiler, which flags attempts to violate scope rules or automatically creates new duplicate variables to prevent a violation, is enforcing a protection policy.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Languages which separate variables into classes by their types provide another kind of limited protection. Manipulation of a typed variable is possible only through use of the operators applicable to that type. Hardware implementation includes tagged word machines in which each variable is flagged to indicate its class (Iliffe & Jodeit [IJ62]).

Protection has usually been implemented as an integral part of the subsystem that controls and dispenses the type of object protected. Computer science literature contains a large number of reports on protection system designs and implementations, restricted to controlling access to individual types of objects such as language variables, core blocks, virtual memory segments, files, buffers or processes. The environments with respect to each different type of object are represented, maintained and employed differently.

In most cases protection is entwined with resource allocation strategies, synchronization and specific computational requirements of that subsystem. Few people have approached protection as other than an adjunct to the resource allocation problem. Consequently a general theory of protection has been a long time in evolving compared to advancement in other areas of programmed system research.

A Model

In our model of protection a programmed system is a community of processes which share the use of objects. † The objective of providing

† The term 'object' refers to software-defined structured encoding of information to which access is to be controlled as well as physical resources (for example, a block of memory or a device).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

protection is to restrict the objects a process can use to be those of immediate relevance to the task that the process is performing. Thus a process is made to operate in an environment that specifies the subset of objects that the process can reference as well as the variety of ways in which the process can access each object. We refer to the environment as being composed of rights, each naming an object and an access applicable to that object.

From this it follows that if a process attempts to use an object, the process is successful only if the right to do so is in the operating environment of the process. This is the first aspect of a protection system: the enforcement of protection.

The relevance of specific objects to a process changes in time. Consequently the rights of a process to use those objects should also change. For some objects the process no longer needs to have access rights. For others, rights must be acquired. This is the genesis of the second aspect of a protection system: the transfer of rights into and out of environments.

The changing relevance of objects to a process can also result from the need of the process to operate temporarily in an entirely different environment. That brings us to the third aspect of a protection system: the binding of process execution to environments.

A Theory of Protection

It is the model sketched in the previous section that the thesis explores and exploits. The model embodies a theory about the fundamental components

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and structure of protection. This section states these basic assumptions, elaborates on how they shape our model, and defines a set of terms (object, access, type, environment and right) as well as the three rules that must be satisfied to define any individual protection system covered by the model. Our theory is grounded in past work. Therefore this discussion is used as a vehicle to point out ideas from the literature with which we agree or are in conflict.

One basic notion of this theory of protection is that each **object** whose use is to be controlled is a uniquely distinguishable structured encoding of information. An **access** to an object is an algorithm to reference the object in order to transform it or extract information encoded within it.

The access algorithm depends upon the internal representation of those objects to which the algorithm can be applied. Thus the set of all objects is partitioned into equivalence classes, each called a **type**. The type of an object determines the accesses that can potentially be applied to that object. Example: All objects of type file can be manipulated only by the accesses: READ, WRITE, APPEND, SORT, ERASE, each of which manipulates a file object to perform a function suggested by its name.

Knowledge of the internal representation of objects need not be available outside the access algorithms. Any description of an object type required by a user is completely specified by the definitions of applicable accesses.

A protection system does not rely upon the interpretation of objects and accesses, but provides a facility defined in terms of the use of their names. In general, both the structure of objects and the access algorithm definitions are irrelevant to protection. It merely prohibits performance of the algorithms

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

unless a specific right, the (object name-access name) pair, is in the execution environment.

In some early examples of protection systems, restrictions on the use of an object were attributes of that object. For example, if a memory block object could be manipulated by two access algorithms, READ and WRITE, then one aspect of the state of a block indicated whether that block was READ-only. The disadvantage of this kind of protection is that two users cannot have different access to the same object simultaneously.

A second basic notion allows this problem to be avoided by requiring that the access to an object depend not on the object, but on the path leading to it so that two users who follow different paths to reach the same object can be permitted to access it differently.

Access path control is embodied in our model as follows: First we define each process to execute in a single environment at a time. An **environment** is a table of **rights** each expressed as an (object-name, access-name) pair restricted so that an access is named in a right only if it is applicable to the object named in that right. Secondly, the activity of a process is restricted by the **Enforcement Rule** which states that a process can access an object only when the right to do so is in the environment of that process. The path from a process to an object is through the environment of that process. Thus when executing in two different environments, two processes potentially have different rights to the same object.

The environment is a definition of the tools a process has at hand. For any system a specific implementation of the environment must be bound. In many

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cases that implementation chosen for the particular system will restrict the protection policies that can be implemented easily using that environment representation.

An early implementation of environments appeared in a paper by Jack Dennis and Earl Van Horn [DVH65] which introduced (I believe) the concept of having an unforgeable token indicating that its possessor has the right to access an object (usually a segment of information in their case) in a certain well-defined way. Without such a token, called a capability, a user has no authorization to perform any access to that object. Capabilities are a direct implementation of rights: one capability per right so that the environment of a process is expressed as a set of capabilities.

Both Dennis and Van Horn [DVH65] and Lampson [L69b] applied capabilities more generally. Capabilities were used to protect not only objects designed for retention of arbitrary data such as memory blocks or segments, but also for more structured objects such as programs, processes and environments. One implication of this generalization is that the number and complexity of accesses required for the more structured objects exceeds that for the simpler kinds of objects. Controlling the use of active (processes) and passive (data retaining) objects in a homogeneous way lessens the distinction between them and facilitates substitution of one for the other, when appropriate.

CAL-TSS (Lampson [L69b]) implemented protection in terms of capabilities showing that homogeneous protection of many types of objects is not only conceptually attractive, but viable as the basis of implementation for an operating system. One restriction designed into CAL-TSS was that all possible

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

object types were predefined in the design. Our model is not restricted in this way but allows dynamic creation of object types. In fact it permits some control of entire classes of objects (i.e. by types).

Using capabilities to control access to arbitrary object types permits homogeneous representation of environments. Thus the environment of a process is sufficient to provide names for precisely those objects a process can use. To specify a physical or abstract object (one contructed and maintained completely by software), the process names not the object, but a capability for it. Thus a process has no way of specifying any object that it is unable to access in some way.

Hardware implementations of reference validation mechanisms have been built in many machines. The MULTICS hardware interprets segment capabilities (Graham [G68]). The B5000 (Burroughs [B61]) interprets descriptors. In the Rice Machine (Iliffe & Jodeit [IJ62]) access to variable size memory blocks was controlled though codewords. Each codeword represented the right to access the block described by that codeword. Hardware automatically computed indirect references through a sequence of codewords and prevented unauthorized changes to codewords.

Another basic notion of this protection theory is that the objects relevant to the execution of a process change in time. Any change in what a process can do is expressed by a change in its operating environment. Thus our model requires that to define a particular protection system, one must define a **Right Transfer Rule** that specifies the well defined way in which rights enter and leave an environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Some researchers have tried to find a single policy governing right transfer that is sufficient for most if not all applications (Graham & Denning [GD72]). The policy Lampson [L71] suggests is based upon the attribute of 'ownership'. A permanent and unique association is made between an object and one environment. The owner that executes in that environment can permit others to access objects he owns. It is not clear that ownership is the only attribute on which a protection policy should be based; the real value of ownership is in the realm of resource allocation control and is only one of a number of attributes which could form the basis of protection administration.

As an alternative to the capability representation of environments, Lampson [L71] proposed recording environments in an access matrix indexed by environments and objects. Each entry in the matrix contains the accesses applicable to a specific object through execution within a particular environment. Thus each row in the matrix encodes one environment.

The matrix representation is particularly appropriate for the illustration of right transfer operations because simultaneous change of multiple environments can be shown as successive matrix values. There are several natural right transfer policies for which the matrix is useful: e.g. migration of rights from one environment to another (an operation involving two rows) and object based access control, where all environments with respect to a particular object can simultaneously change (a column operation on the matrix.) Though conceptually attractive, the maintenance expense of such a matrix makes it unattractive as an implementation of environments in most cases.

The other way a process can change the rights which it can exercise is to temporarily execute in another environment. The Environment Binding Rule of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the model requires specification of how such crossing between environments can take place.

It was recognized very early (Dennis & Van Horn [DVH65]) that controlled entry to programs is vital. (If a process were able to begin execution of a program at an arbitrary point or with arbitrary rights, the function defined by the program would be unpredictable.) However, it is useful for a process to execute in different environments at different times. The ramifications of dynamic environment crossing, particularly where rights are passed as parameters, has not been studied in other work on protection and is a major topic in this thesis.

One specific Environment Binding policy appears in an article by Evans and LeClerc [ELC67] describing a memory system in which a process can cross environment boundaries without carrying parameter rights to the new execution site. HYDRA (Wulf et al [WU73]) and a Virtual Memory System formally specified in a thesis by Price [P73] both link environment boundary crossing to nested procedure calls. Both references contain clearly defined environment boundary crossing policies which do permit parameter rights to accompany a process crossing between environments.

Support for environment binding policies has been implemented in hardware processors. The Burroughs B6700 (Organick & Cleary [OC71]) includes an 'enter' instruction which adjusts environment control registers and the stack for procedure entry. Schroeder and Saltzer [SS72] propose a revision and extension of the reference validation hardware of MULTICS to provide hardware assisted environment crossing ('gate crossing' in their terminology.)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In contrast to the Enforcement Rule which is fixed, both the Right Transfer and Environment Binding rules are meta-rules which require that a policy be specified to define an individual protection system. We believe that no single protection policy can satisfy the requirements of the diverse applications of programmed systems. Therefore no policy, not even one with alternative rules, is included in the model. Instead, we posit a set of primitive mechanisms for right manipulation and show that these can be used to construct a wide class of protection policies.

The model is sufficient to cover a broader class of protection systems than has been possible in the past. Each system is described by fixing parameters to specify the policy embedded in that particular system. Thus the model provides a set of tools for describing and defining a protection policy, analogous to a language that provides a vehicle for representing (and executing) an algorithm.

Organization of the Thesis

The following chapters have three major goals: exploration of the model, verification of the model and exploitation of the structure which protection provides to programmed systems. The next three chapters explore the model by investigating the ramifications of the three rules. Then Chapter V verifies by example that the model does cover a variety of existing systems.

To address the last goal Chapter VI shows how the separation of environment from code can be the basis for proving properties which limit execution. Then Chapter VII formulates protection as an operating system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

facility in which dynamic type creation is the basis for protecting user defined objects. In conclusion, Chapter VIII uses the structure provided by the model to compare and contrast the facilities and limitations of different classes of protection systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ENFORCEMENT OF PROTECTION

Chapter II

The Enforcement Rule specifies that a process can exercise only those rights which are in its current operating environment. The objective of this chapter is to consider the representation of environments and the definition of mechanisms to implement this rule. We begin by determining that all attempts to violate the Enforcement Rule can be detected by checks made in conjunction with object name interpretation. Then we investigate the ramifications of different environment representations and their associated enforcement mechanisms.

A process sequentially exercises rights as specified by a program. In general it is possible that the program directs an executing process to violate protection, i.e. to exercise a right that is not in the execution environment. Therefore either static compile time protection checks or dynamic run time protection checks must be made to detect attempted protection violations. If a check reveals that a protection violation will occur, some action to avoid the violation must be taken. For example, in the dynamic case, execution by the offending process could be aborted. In the static case, the attempt to use a variable that is not available could cause automatic declaration of a new variable with the same local name.

A process distinguishes an object that is to be referenced by using a name for it. All names that can be uttered by the process comprise its current name

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

space. Because a process can only reference those objects for which its environment contains a right, we can assume that there exists one name space associated with each execution environment. The name interpretation function f_E maps the name space associated with environment E to the set of all objects.

$$f_E : \{\text{names}\} \xrightarrow{\text{interpretation path}} \{\text{objects}\}$$

Because name interpretation necessarily accompanies each exercise of a right, correlating the performance of name interpretation and protection checking will guarantee that all requisite checks are made in support of the Enforcement Rule. We assume that every object is uniquely identifiable, both in our discussion and in actual systems. For the purposes of this thesis each object is identified by its unique global name. (To facilitate readability of the following prose, no strict distinction is made between the use of the terms 'object' and 'global name of an object'. The term 'object' is used for either of the two.)

The name interpretation function can be as simple as the identity function, if the name space of a process is the set of global names. (More complex name interpretation functions will appear in later examples.) The name interpretation function itself can be used as a form of protection. This can be achieved by letting the range of f_E be a proper subset of all objects. The result is that a process in E cannot access any object which is not in the range of f_E, for it cannot name such an object.

EXAMPLE 2.1:

Address calculation using base and bound registers restricts a process

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

name space to addressing a contiguous, but restricted, block of memory cells. Let the global name space for a set of memory cell objects and the name space of environment E both be the set 0,1,2, ..., MEM. Associated with each environment E are two values BASE and BOUND, obeying the predicate:

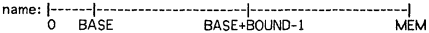
$$0 \leq BASE < BASE + BOUND - 1 \leq MEM$$

The name interpretation function for environment E is

$$f_E(j) = BASE + k$$

where j is in the name space of E, and

k is the minimum non negative number such that $k \equiv j$ (modulo BOUND).



Cells whose global names are in the range [BASE,BASE+BOUND-1] have local names in E and no names exist for cells outside this range. There are cells for which multiple names exist. For example, if c is a positive integer, all names of the form j + (c * BOUND) which are in the name space of E map to the same memory cell.

Assume environment E includes all access rights to cells for which there are names in the name space of E and no rights to any other cell. Since f_E prohibits any attempt to access cells outside the range of f_E, it acts as a mechanism prohibiting protection violation for memory cell objects.

⊠

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Environment Representations.

A mechanism to detect attempted protection violations in the general case depends upon the representation of environments. Because the environment is itself a data structure, we are concerned with not only its definition, but its representation and whether that representation precludes any class of protection policies. In the introduction capability and access matrix representations were cited. Both record environments directly. Associated protection enforcement mechanisms need depend upon only the environment representations and perhaps some constants, but not upon arbitrary system variables. Environments could be represented indirectly as ordered sets of boolean expressions, each depending upon arbitrary subsets of the variables of the entire programmed system. For example, a right to READ the medical record describing patient P is defined to be in the environment of a process executing on behalf of doctor D only

if D treated P in the last 300 days.

To evaluate this 'prose' boolean expression requires reference to P's record and the real time clock.

Thus the most general environment descriptions potentially depend upon all variables in the entire system and require a protection mechanism with the computational power of a Turing machine to evaluate the boolean expression. This thesis is limited to the study of environments which are directly represented, i.e, there exists an encoding † of environments independent of

† Observe that this restriction does not eliminate representations that require decoding according to a formula dependent on constants and known within the mechanism definition. For example, negative encoding is permissible: An environment is represented by the rights not in it, so that a process can exercise any right not specifically found in the 'negative representation of the environment'.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

other types of variables in the system. The motivation for considering only direct representations is so that access to environments can be controlled as for any other object.

This limitation does not restrict the class of protection policies covered by the model. In general, the reason is because any policy can be programmed. In particular, consider the problem of defining right r to be in E if $B(x_1, ..., x_n)$ is true, where $x_1, ..., x_n$ are arbitrary variables. To implement this, we define a procedure † named P. Environment E does not include r but includes the right to call P. Procedure P evaluates $B(x_1, ..., x_n)$ in its private execution environment that does contain r. Only if $B(x_1, ..., x_n)$ is true is r exercised on behalf of the caller. Therefore policies defined using indirectly represented environments can be implemented using directly represented environments.

The benefits of this limitation are that first, environment alterations are explicitly performed so that the content of an environment can be protected as an object. Secondly, an efficient protection mechanism can be constructed. (We are particularly interested in the possibility of defining dynamic protection checking mechanisms in microcode or hardware.)

A further assumption made in the model is that each right can be exercised independently. This restriction does not reduce the class of protection policies that the model covers. Where applications dictate that the exercise of one right is to be dependent upon the exercise of another right, the necessary sequencing of right usage can be programmed in a procedure. The dependent rights are available for use only during execution of that procedure. ††

† A procedure is a body of code together with a specification of the environment in which the code is to be executed.
†† Later discussion will show that such procedures can be constructed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Given these assumptions, we consider only the direct representation of environments and their associated protection checking mechanisms.

Protection checking can be performed:

1. at the beginning of the name interpretation path, i.e. the site at which names are generated
2. at the termination of the interpretation path, i.e. the site of the referenced object, or
3. at an intermediate stage along the interpretation path by referencing data structures not needed for name interpretation.

Environments vary in time, therefore whether the check is performed dynamically at run time or statically at compile time, the mechanism requires a representation of the environment (actual or predicted). The most general picture of an environment is as a set of triples, each of the form: (global name X, access name A, environment name E). (X,A,E) exists at time t if and only if access A can be performed on object X by execution in environment E at time t. (In discussing different representations of protection information, the triples will be used as a standard description of the information to be recorded.)

The triplets can be stored in a matrix with two of the three items as indices and the third recorded as the content of the matrix structure. Though useful for explication (Lampson [L71]), such a monolithic structuring of protection data (even in compacted form) is rarely as advantageous as other forms of encoding. Use characteristics of environments suggest that they can be represented and maintained more efficiently if subdivided into multiple structures. As later chapters uncover, when one environment is of interest (e.g., the current operating environment of a process), in general other environments are not of interest. Here, encoding each environment as an separate structure can be most efficient. For some protection data usage,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

when rights with respect to a particular object are of interest, rights to other objects are not of interest. In this case encoding all rights to an object in one structure can be the most efficient. Distributing protection information among many structures rather than encoding it within a monolithic structure also contributes to isolating errors in referencing the protection data to a subset of data representations.

Protection checking mechanisms depend upon the representation of environments. The name interpretation function f_E will be extended † to include protection checking for three different direct representations of environments. Each representation houses pairs of items from a triple at the site of the third member named in that triple. It will be demonstrated that the **keys, access lists, capability lists** and **categories** defined in the literature arise from using one of the alternative structures.

Execution Site Enforcement

A first extension of the name interpretation function incorporates protection checking at the beginning of the interpretation path--the site of execution, where local names are generated. For each environment there exists a capability list representation (Dennis & Van Horn [DVH65]). Each right in an environment E is encoded as a capability of the form (global object name X, access name A). Because a capability list encodes one environment, we will sometimes refer to a capability list as an 'environment structure'. Environment structures are a natural encoding when interest is focussed (as it usually is in this thesis) on the execution that can take place in a specified environment.

† The extended function will be called F_E to distinguish it from the name interpretation function f_E that does not include protection checking.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The name interpretation function for an environment E, f_E, is extended to include protection checking. Because the Enforcement Rule restricts the objects a process can access to those for which its environment contains a right, and because the capability list representation can be referenced at the execution site, then without loss of generality, we can assume that the name space for execution in an environment E is the set of selectors distinguishing elements in the capability list. If N is a local name, the E[N] selects a right (X,A) from the capability list encoding and $f_E(N) = X$. The name interpretation function is extended to perform a protection check before returning the global name of the object referenced:

$F_E(N,A) \equiv$ **if** E[N] = (X,A) **then** X **else** protection error

Recall that we have adopted the convention that 'protection error' triggers some action which prevents the protection violation from occurring. (For example, a dynamic employment of F_E can result in abortion of execution by the offending process.) The Enforcement Rule is implemented by the extended function: a process can reference an object only by specifying its local name and all local names are interpreted by the extended function F_E.

Encoding protection data in an environment structure at the site of execution has the following advantages:

1. It provides a name space (names are selectors into the capability list) that includes names only for objects accessible from within that environment.
2. It isolates within a single data structure all protection information pertaining to execution in each environment. (This structure need only be available when execution in the environment is in progress.)
3. Protection checking can be performed without leaving the execution site. By definition, F_E prevents all violations of the Enforcement Rule.

Terse encodings of E can be obtained by exploiting regularities in either

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the content or the pattern of use of E. The following commonly used techniques are applicable to capability lists as well as the other alternative encodings of the protection data base to be discussed later:

1. Bit Coding: If fixed items of information consistently appear in many instances of a structure, then each item can be associated with a bit position whose boolean value records the presence or absence of the associated item. All users of a structure in which bit coding is employed must know the mapping between information items and bit positions. All rights with respect to one object in an environment can be represented in one capability of the form: (global object name X, boolean access list A). Let the finite set of accesses applicable to instances of each type of object be ordered. The k-th bit of the access field in a capability records the presence or absence of the k-th (applicable) access to the object named in the capability.

EXAMPLE 2.2:

Virtual memory systems call capability lists for segment objects segment tables and employ ordered bits to encode the presence of READ, WRITE and perhaps EXECUTE accesses to each segment object. (The global identity of the segment which is needed for sharing is known to the segment controller. However, to accommodate resource allocation, the segment table holds a physical memory block address which uniquely identifies the segment among those with which it might be confused, i.e. those in primary memory or in swapping store.)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bit coding can be similarly applied to the object field of a right if a finite set of objects X_1, ..., X_n are always used together with the same access. The rights in E to perform access A on a subset of the objects are represented as a capability of the form (X,A) such that X[i] = 1, $1 \leq i \leq n$ if and only if the protection triple (X_i,A,E) is to be recorded. (The extended name interpretation function must be adjusted to compute global object names using the bit coded fields.)

2. Grouping: If an environment includes precisely the same access rights to a set of objects (necessarily of the same type) at all times, then the object names can be joined together in a set which is referred to as a **category** [G72]. Let $C = \{X_1, ..., X_n\}$ be a category. The rights to perform access A on all objects in the category can be encoded as one capability (C,A). Both bit coding and grouping techniques are adopted to minimize the cost of encoding the protection data and do not alter the encoding site of that data.

Object Site Enforcement

An alternative to performing protection checking at the beginning of the interpretation path is to do it at the end of the path. The translation from the local to the global name of an object precedes protection checking because it is with the object itself that the encoding of the appropriate protection information is located. Each triple (object name X, access name A, environment name E) is encoded as the pair (A,E) in the table named X*, a structure which is associated with the representation of the object X. In the literature X* is referred to as an **access list** [L71].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Let N be a local name in the name space of environment E. Then the extended function is defined

$F_E(N,A) \equiv$ **if** $f_E(N) = X \wedge (A,E) \in X'$ **then** X **else** protection error

This extension differs from the execution site extension of f_E in that here, the definition of the name space for the environment E is necessarily divorced from protection, since name interpretation must precede protection checking. The sequential nature of interpretation and checking precludes implementation of the two in parallel.

The bit coding technique can be used for elements of an access list to encode either the access field (since accesses potentially applicable to X are fixed by the type of X) or the environment field (if the number and identity of environments are relatively constant).

Grouping of environments into disjoint sets is done by defining an environment group $G = \{E_1, ..., E_n\}$ such that all environments in the grouping include the same access to X. The triples (X,A,E_i), $1 \leq i \leq n$, are represented as the one pair (G,A) encoded in the access list X' associated with X.

EXAMPLE 2.3:

Define a set of environment groups $K_1, ..., K_n$, calling each group name a **key** [IBM]. Associate with each object an access list containing a pair specifying one key name and a list of accesses. Access A can be performed upon object X from within environment E if $E \in K_i$ and $X' = (K_i,A)$. This use of grouping tersely encodes the protection data, but is restricted in that it is useful only if environments that are grouped together have common access to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

more than one object, and if member environments do not enter and leave key groups often.

⊠

Access Site Enforcement

The remaining point at which protection data can be recorded is within a structure not otherwise needed for the interpretation of names. As mentioned earlier, one such alternative can be to have a single structure such as the access matrix which encodes the protection triples. Another alternative is to encode the protection data as pairs in a table associated with the access involved. Define each entry in the table A' as (X,E) such that (X,A,E) exists. The name interpretation function is then:

$F_E(N,A) \equiv$ **if** $f_E(N) = X \wedge (X,E) \in A'$ **then** X **else** protection error

F_E must be capable of locating and referencing the table A'. Compact encoding of A' can be constructed by bit coding and grouping as in the previously described representations.

EXAMPLE 2.4:

In ALGOL 68 each operator is defined to apply to operands of specific types. Each time an operator appears in the program text, the compiler checks whether the operands are of compatible types.

⊠

We do not assume a particular implementation of any of the protection checking mechanisms, F_E, specified above. We note that if protection checking

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is performed each time a local object name is interpreted, all necessary protection checking is accomplished. It is the perogative of the designer to choose how such a mechanism is realized. For example, access site enforcement could be achieved by placing the protection check in the prelude of each access algorithm implementation.

Interleaving of Name Interpretation and Protection Checking

Protection information can be represented within a combination of structures. For example, a series of environment structures can be referenced while performing a series of protection checks at intervals along the interpretation path.

EXAMPLE 2.5:

A file system in the form of a rooted directed graph with files as terminal nodes is structured to permit its users to name arbitrarily overlapping sets of file objects. A major goal of this structure is that in a controlled way one user can alter another's right to a file. To adapt to these requirements, protection data is encoded in capability lists or environment structures known as directories. Each directory contains rights to only two types of objects, directories and files.

A local file name is interpreted to select a sequence of nodes beginning with the root directory and ending with a terminal file node. At each non-terminal node, a portion of the file name is used to select a right (encoded as a capability) from the directory of that node. The selected right names the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

successor node, another directory or file. Before further name interpretation takes place, a protection check is performed to insure that the process which generated the local name is permitted to reference the successor node. Thus protection checking is performed at each intermediate node, interleaved with name interpretation.

⊠

Time at Which Protection Checks can be Performed

The previous discussion addressed how the enforcement of protection can be performed. Now we consider how it can be done as inexpensively as possible. Bit coding and grouping have already been introduced as techniques to minimize the amount of storage used for encoding protection data.

The second cost measure is in terms of execution time. If protection checking can be done in parallel (i.e. in the hardware) with other necessary name interpretation activities, then its cost in time is small. If protection checking cannot be done in parallel, then static checking (when possible) is cheapest, for it is done only once. A static check is one made at the time that the request for an access to an object is encoded, for example at compile time. A protection check can be performed as soon as both

1. the mapping from the local name to the object, and
2. the elements of the triple to be checked for existence

are unalterably fixed for a predictable interval of time which extends to include all executions which depend on that check for protection enforcement.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Static checks imply that the interval of time between the protection check and an exercise of a right which depends upon it, extends from compile to run time. A static check is not always possible because of:

1. external influences--a process capable of causing alteration of pertinent protection data executes during that interval between a static check and invocation of an action depending on that check
2. internal influence--execution by the process on whose behalf the protection check is made can cause a change the protection data base in an unpredictable way (for example, by a procedure call whose side effects are unknown at compile time.)
3. interdependence of the name space mapping on preemptive resource allocation which potentially alters that mapping
4. a member of the protection triple is unknown until run time.

Protection checks can be performed at three different times. The first is the static check. A compiler knows the mapping from local names for data structures to memory cells, for it performs name interpretation and generates code which can not compute a request to access an illegal memory cell. Because index values cannot be checked at compile time, static compiler checks fail (point 4) in the case of array structures, so that array bound checking must be done dynamically.

Therefore another kind of protection checking is dynamic: immediately and indivisibly prior to a single performance of the access to the object. This is the time at which protection checking is done for objects which are represented in terms of preemptible resources, as virtual memory objects are.

The third kind of protection check is dynamic but the check applies to more than a single performance of an access. For example, one protection at entry to a loop in which a right is repeatedly exercised may suffice for all uses of the right. To meet cost effectiveness objectives, one protection check should suffice for multiple exercises of the same right, where possibie.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Summary

The Enforcement Rule of our protection model states that a process cannot exercise any right not in its current execution environment. This chapter focusses on the encoding of protection information within different data structures and the associated protection checking mechanisms defined in relation to object name interpretation. We point out that the different implementations of protection checking reported in the literature arise from variations on these protection information encodings.

There is a natural ordering to the times at which protection checking can be performed. But no ordering exists among the alternative encodings of protection information or their associated name interpretation functions. The criteria for choosing among the encodings is to select the encoding for which the cost of updating and referencing during protection checks is minimal. If the update of an environment can occur only as a result of an action taken by a process executing in that environment, then capability or environment site encoding is preferable. If updates by one process affect any or all environment (for example, a column operation on the access matrix), then object coding may be the only way to make encode the protection information without inducing massive search costs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

RIGHT TRANSFER PRIMITIVES

Chapter III

The Right Transfer Rule of the protection model specifies that every protection system must define a policy to regulate the movement of rights into and out of environments. In any particular system both the right transfer policy and its implementation must be bound. (This contrasts with the Enforcement Rule which is fixed for every protection system that the model covers, although implementations of it vary, for example, with the representation of environments.)

The goals that different policies serve can conflict. Compare:

G1: A process that owns an object controls the use that can be made of that object.

G2: Processes are autonomous.

In a system where policies serve goal G1, a process may be permitted to erase an owned object by transferring all rights to access that object out of all environments, including the execution environment of other processes. Such an action violates goal G2.

In recognition of the fact that one policy cannot serve all reasonable goals, our protection model does not contain a policy. Instead, we assume that each individual protection system specifies a right transfer policy as a finite set of predicates controlling environment alteration. If a specific predicate holds for a protection state, described as the value of a set of environments, then a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

specific protection state change is permitted. The variability among right transfer policies in different systems arises from the predicates which specify policy decisions.

This chapter investigates what is common to right transfer policies: the environment alterations. These operations are called primitives for they are the only means to manipulate environments and rights. Chapter V demonstrates that these operations do form a set of building blocks sufficient to construct a variety of protection policies. We are not concerned with sorting out minimal set of primitives, so much as separating policy from mechanisms.

The primitives will be described in program form and therefore require a representation of environments and rights. We choose to represent an environment as a table of rights. If E is an environment then E[x] names the x-th right, selecting it from environment table E. Indexing can be by content addressing, by linear offset or even by indirection through a data structure other than the environment, but the method of indexing need not be specified here. A right is defined to have two fields: †

struct right = obj, acc

Let r be an instance of a right; obj:r selects the name of an object from the obj field of right r and acc:r selects the set of accesses which can be made to obj:r. The acc field contains a set, rather than a single access name for notational convenience and to allow later introduction of accesses which can be used to control manipulation of the right itself.

Each primitive will be defined in terms of reading and writing the two fields of a right, selecting individual rights from environments (using the

† Appendix A contains an explanatory discussion of the few programming notations required for exposition of the primitives.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

bracket notation, E[x]), and adding and removing rights from an environment. The primitives are the only way in which right and environment entities can be manipulated.

The first pair of primitives are those that can be used to arbitrarily redistribute existing rights among a set of environments.

COPY(r,E,E') ≡ E' ← E' ∪ E[r]

E and E' are environments existing before the COPY primitive is performed. The contents of E[r] is copied into an empty slot in environment E'. Whenever possible, environments will be treated as sets to avoid the added notation of naming table slots. However when writing programs in later chapters explicit locations of rights within environments will be specified.

The primitive

DELETE(r,E,a) ≡ acc:E[r] ← acc:E[r] - a

removes from the access field of right E[r] any access names in that field which are also in the set a. We adopt the notation that a slot in the tabular representation of the environment is empty if the acc field of the right in that slot is null. The notation DELETE(r,E,all) means erase all access to obj:E[r], i.e erase the entire right.

To show that COPY and DELETE are sufficient to allow arbitrary redistribution of rights among environments, some terminology used in describing the effect of sequences of right transfers is defined.

Defn: A **protection state** S is a set of environments, each with fixed content at an instant t.

Pairs of states which are time ordered are denoted by S, S' where S' is a later state than S. Environments are similarly marked prime so that E' ∈ S'. A right is said to be in state S if it is contained in at least one environment in S.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Defn: Let T denote a set of transformation functions including the null function. $\forall t \in T$, S **directly derives** S' ($S \Rightarrow S'$) by t if S' is the result of performing the t transformation on state S.

Each transition between states can be thought of as being instantaneous.

Defn: S **derives** S' ($S \stackrel{*}{\Rightarrow} S'$) by the transformation sequence $t_1, \ldots, t_k$ if there exists a sequence of states $S=S_0,S_1, \ldots, S_k=S'$ such that $S = S_0 \Rightarrow S_1$ by t_1, $\ldots, S_{k-1} \Rightarrow S_k = S'$ by t_k.

We can now show that the primitives COPY and DELETE are sufficient to effect any redistribution of existing rights among a set of environments. To show this property, we define two states S and S', where each right in S' is in S. It is sufficient to show that $S \stackrel{*}{\Rightarrow} S'$ by a finite transformation sequence. Define T = {COPY, DELETE}. Let S and S' contain the same environments. There are n rights of the form r, such that $r \in E_i' \wedge \neg(r \in E_i)$ but $r \in E_j$, $i \neq j$. We construct a sequence of states

$S = S_0 \Rightarrow S_1 \Rightarrow \ldots \Rightarrow S_n \Rightarrow S_{n+1} \Rightarrow \ldots \Rightarrow S_{n+m} = S'$. $S \stackrel{*}{\Rightarrow} S_n$ by a sequence of n COPY transformations, and $S_n \stackrel{*}{\Rightarrow} S'$ by a sequence of m DELETE transformations. Each COPY transformation copies one of the n rights described above from an environment in S to the environment in which the right appears in S'. The m deletions remove the excess rights that remain in environments in state S_n after all copying has been performed. (This intuitive sketch of the proof could easily be formalized.) Thus we have shown that the primitives COPY and DELETE are sufficient to effect any redistribution of rights among a set of environments.

The third primitive to be defined is

GRANT(r,E,E') ≡ COPY(r,E,E'); DELETE(r,E,all)

The purpose of the primitive GRANT is to permit a right to be conserved as it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is transported between environments. Performance of an individual primitive causes a state transition from one distinct protection state to another with no intermediate state. Sequences of primitives do have intermediate states. Thus the effect of COPY(r,E,E'); DELETE(r,E,all) differs from that of GRANT(r,E,E') in that there does exist a state in which r exists in both E and E'. This primitive is very useful in proving that certain rights are conserved and can not be replicated under any circumstances.

EXAMPLE 3.1:

This example demonstrates how the Right Transfer Rule is bound to a specific policy in one protection system. The policy is implemented by two procedures, using only the GRANT primitive to do all manipulation of environments. No other primitives are defined in this system and GRANT is not invocable outside the protection system procedures.

Processes execute in one environment at all times and are related by their predefined family tree using the boolean predicate:

child(i,j) ≡ process i created process j

Assume all rights initially exist and define the protection system procedures:

procedure Give$_i$(r,j) = **if** child(i,j) **then** GRANT(r,E_i,E_j)

procedure Take$_i$(r,j) = **if** child(i,j) **then** GRANT(r,E_j,E_i)

Give and Take are assumed to know the identity, i, of the calling process and its execution environment, E_i. In this parent dominated system, a parent can Give to any child a right which the parent possesses (losing it himself) or Take away any right his immediate child has.

Relying on the indivisibility of GRANT, it can be shown that all rights are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

conserved. It can also be shown that if $i \neq j$ and i and j are not direct descendants then there exists a process k, differing from i and j, such that k can prevent the passage of a parameter right from E_i to E_j.

⊠

The next primitive to be defined permits creation of new rights. As specified in the introduction, every object has a unique type which determines the accesses applicable to it. The acc field of a right is restricted to naming only accesses applicable to the object named in the obj field of that right. It is the creation primitive that must initialize new rights observing this restriction.

For every object type t, let there be a set of names, denoted Access_set(t), of all accesses defined to be applicable to an object of type t. (Chaper VII will consider the creation of types and specification of the associated Access_sets.) When a new object of type t is created, a right containing all accesses to that object is constructed using the CREATE primitive. We stipulated that every object must be distinguishable from other objects in order to be protected. † And we have chosen to uniquely distinguish objects by a global name. A variable, GEN, private to the (indivisible) CREATE primitive is incremented each time CREATE is performed to provide unique global names. Define

CREATE(t,E) ≡ E ← E ∪ (GEN ← GEN + 1, Access_set(t))

It is important to realize that this primitive augments E with a new right containing the name of an object differing from the names of all objects created in the past; but the primitive does not create a representation of the object itself.

† Actual implementations can take advantage of the knowledge that an object must only be differentiated from a subset of all objects and provide it with an identity unique only among that subset.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is useful to be able to read the contents of a right without altering it. (Security conscious systems can elect to control the reading as well as alteration of rights.) To aid such applications

READ(r,E,x) ≡ x ← E[r]

is defined to copy the encoding of right E[r] into location x.

The primitives already defined in support of the right transfer rule permit rights to be created, lessened (by reducing the acc field) erased, read and moved arbitrarily between environments (either by copying or by conserving the entire right).

As stated earlier, in any individual protection system a primitive is successfully invoked to alter an environment only under conditions specified by the right transfer policy. For this to be enforced, environment representations themselves must be protected from arbitrary alteration. To facilitate that protection, environment representations can be declared to be objects. Before illustrating this rather elegant use of protection, consider limiting the manipulation of individual rights.

We can control the use of the primitives on individual rights by extending the acc field of a right to specify how the right itself as well as how the object named in the right can be manipulated. † For example, the acc field of every right could potentially includes the access names GRANT, DELETE and COPY. (The names of accesses as well as primitive names will be capitalized in this thesis.) The DELETE primitive could be used to reduce the access field of

† Declaring rights to be objects would lead to infinite regression. If rights are objects, then the creation of a right is also the creation of an object. Therefore it must be accompanied by the creation of a second right used to control access to the first right. But the second right is also an object and requires the creation of a third right to control access to the second . . .

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

right r only if DELETE $\in$ acc:r. Similarly a right r could be copied only if COPY $\in$ acc:r.

To illustrate both protection of environment objects and limiting right manipulation, we reformulate the example of the totalitarian parent.

EXAMPLE 3.2:

The parent process arbitrarily gives and takes rights from immediate children with the exception of rights which are to remain permanently in one environment. Environments are objects with accesses TO, FROM. Each process has TO and FROM access to its execution environment, so these rights are not shown. The acc field of all rights is extended to potentially include GRANT. For each parent i with a direct descendant j, $(E_j$,FROM TO$) \in E_i$, so that the parent can manipulate each child's environment as well as its own.

Again we assume all rights initially exist and that each process executes in one environment at all times. The right transfer policy is defined in the two procedures:

procedure Give$_i$(r,j) = **if** GRANT $\in$ acc:$E_i[r] \wedge (E_j$,TO$) \in E_i$ **then** GRANT(r,E_i,E_j)

procedure Take$_i$(r,j) = **if** GRANT $\in$ acc:$E_j[r] \wedge (E_j$,FROM$) \in E_i$ **then** GRANT(r,E_j,E_i)

Observe that a parent can not arbitrarily remove rights from its own or its child's environment. If GRANT does not appear in a right, then that right remains in one environment forever. Thus a parent cannot lose or give away the right to control its child since the access field of the right to the child's environment does not include the access name GRANT. In addition, a child that has a non-grantable right in its environment is shielded from the parent who cannot remove it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Using rights to control environment manipulation provides the means to adopt policies other than the totalitarian parent policy. In fact, different initializations of rights to environment objects would permit different policies to apply to subparts of the process tree within the same system.

Summary

Investigation of the Right Transfer Rule shows that different policies controlling the transfer of rights into and out of environments serve different goals. Consequently no one policy is sufficient to model all possible protection systems. This chapter postulates a set of primitives: COPY, DELETE, GRANT, CREATE and READ. Using these primitives to cause all possible environment and right alteration, a diverse set of right transfer policies can be defined.

We do not argue that the primitives specified here are the only possible set or that they are universal in that any policy can be formulated using them or that they all must be present in each protection system. We do argue that their specification clearly separates policy and environment alteration mechanisms, and that they are sufficient to construct a large class of right transfer policies as will be demonstrated in Chapter V.

This chapter develops two ideas in examples that are useful in later chapters. First the access field of a right can be extended to include the names of accesses that can be performed on the right itself. Secondly, environments can be defined to be objects so that with rights to environments the protection system itself can be used to control environment manipulation. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

opens the possibilities of varying the way different environments can be altered as well as using environments as right storage areas.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ENVIRONMENT BINDING

Chapter IV

The Environment Binding Rule of our model requires that each protection system specify how a process can cross between environments. Processes temporarily suspend execution in one environment to cross to another environment in order to perform a task for which the second environment contains the appropriate rights. (If absence from the first environment were not temporary, then that environment could be altered to suit the next task the process is to perform.)

This chapter investigates the specification of environment binding policies focussing on the generation of parameter rights that accompany a process crossing to a new environment. It defines one more primitive operation called AMPLIFY for cases in which the right transfer primitives defined in the last chapter are insufficient to describe modification of a parameter right as it enters the (target) environment. A set of parameter generating mechanisms are considered and shown to be partially ordered by containment. All mechanisms can be defined in terms of our model. For use in later chapters, a particular environment binding policy based on procedure invocation and AMPLIFY is defined in detail.

We assume a process executes in one environment at a time and that each environment is the site of execution by no more than one process, though that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

process may suspend execution in an environment to resume it later. One degenerate environment binding policy defines how a process is initially given an environment and does not permit environment crossing at all.

Ideally an execution environment should contain the minimum set of rights needed to perform the task(s) executed within that environment; i.e., an executing process should have access to those objects it 'needs to know' about and no more. Each unnecessary right provides additional opportunity for errors to have a 'worse' consequence. The algorithm to perform a task in an execution environment is given as a program that can be invoked with parameters. The rights required to execute a program are of two flavors: the declared rights needed whenever the operation is executed (for example, access to the encoding of the program itself) and the dynamic rights which are invocation dependent. The dynamically determined rights could depend upon any variable in the system. Needham [N72] discusses a system where the rights to execute a task depend upon the program to be executed, the executing process and the parameter rights.

Several attitudes can be adopted regarding dynamic determination of parameter rights for a specific program invocation. Either the execution environment in which a program is to be executed:

1. includes (or can generate) all necessary rights, or
2. the process invoking that program brings rights with it when crossing to the execution environment, or
3. the needed rights are generated during the act of crossing.

Each of the three alternatives (no parameterization, parameter passing, and amplified parameter passing) reflects a strategy for providing needed rights. We will consider the three strategies in more detail. To do this, we define a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

set of six increasingly complex parameter generating mechanisms that will be proven to be partially ordered in terms of the actual parameter values they compute.

No Parameterization

If no parameter rights can be passed to an environment in which a certain program is to be executed, then that environment must include all rights potentially required to execute that program, no matter what data parameters are used at its invocation. This is the strategy adopted in processor hardware which implements supervisor and user environments. Usually a supervisor environment includes all possible rights. No matter what the supervisor is called upon to do, all rights needed as well as rights required for unrelated operations are available.

For each parameter strategy to be discussed we define a mechanism that implements that strategy for comparison purposes. To do so we assume that all programs are packaged as procedures defining how to perform a single task and having one entry point. A process in environment $E_{\pm}$, performs a call to procedure Q to perform the task that Q implements. Part of the work of performing such a call is to generate **actual parameter(s)** for use by the called procedure. Each of the six mechanisms to accomplish actual parameter generation will be described as a function of two arguments:

Q, a procedure being called, and

r, a **parameter right** from the calling environment, $E_{\pm}$, to be used in the construction of the actual parameter right(s).

Only a single parameter is used for ease of notation, since extension of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mechanisms to multiple parameters is straightforward. The parameter mechanism itself does not perform the calling operation; rather it evaluates to the actual parameter(s) to be made available to the called procedure and thus forms only a portion of a procedure call. Included in the definition of every procedure is a description of the environment in which that procedure must execute. Whether that environment exists or is constructed at the time of a call to the procedure is not relevant in this discussion.

The mechanism for no parameterization is

NONE(Q,r) ≡ ;

An invocation of NONE(Q,r) generates no parameters to accompany a procedure across environment boundaries. Thus communication between the call site, E_*, and the execution site of procedure Q, E_Q, is possible only if E_* and E_Q have appropriate access rights to a shared object.

Passed Parameters

The second strategy permits passing parameter rights from the calling environment to the called environment. The actual parameter is generated by the IDENT mechanism defined as

IDENT(Q,r) ≡ r

The caller operating in E_* can designate any of its current rights as the actual parameter to be passed unchecked and unchanged for use during the execution of procedure Q. Because the parameter right r is an element in the environment of the caller at the time of the call, COPY and GRANT are sufficient to describe the movement of right r into environment E_Q.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the mechanisms yet to be defined, the generated actual parameter(s) are functions of both the actual parameter right designated by the caller and a procedure parameter template (which was defined at the time the procedure was declared.) The first kind of parameter specification template restricts only the type of object that can be named in an actual parameter right.

template(Q) = ty

Thus Q requires a parameter right to an object of type ty.

A third mechanism evaluating to unaltered parameter rights from the calling environment checks that a parameter right names an object which is of the type † expected by the procedure being called.

The type checking parameter mechanism is defined as

TYCHECK(Q,r) ≡ **if** type(obj:r) = ty **then** r **else** protection error

type(obj:r) deterimes the type of the object named in the actual parameter right. Thus a calling process can designate as a parameter only a current right that names an object of the type acceptable to the procedure being called. It will be shown later that mechanisms IDENT and TYCHECK can be used to generate the same class of actual parameters. Thus COPY and GRANT are still sufficient to transfer the parameters to the called environment.

Amplified Parameters

To motivate further parameter defining mechanisms, consider in more detail the use made of parameter rights. A process calls a procedure Q to

† Types partition the set of all objects into equivalence classes. All members of one class can be manipulated by the same accesses. This implies that objects of the same type share the same structural properties and the called procedure expects a parameter right to an object with known structural properties.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

perform a task which references an object named in a parameter right. Procedure Q may require a different access, in particular a more extensive access, to the parameter object in order to accomplish its task. **Access amplification,** defined by mechanism ACCAMP, generates an actual parameter right in which the access field from the parameter right is augmented by a new set of access names. The increased accesses are particular to the procedure being called and possibly unknown at the call site. They are described in the extended parameter template for that procedure:

template(Q) = ty, reqacc, ampacc

where ty names an object type, and

reqacc, ampacc are sets of names of accesses ('req-uired access' and 'amp-lification access' respectively) that are applicable to objects of type ty.

Define

ACCAMP(Q,r) ≡ if type(obj:r) = ty ∧ reqacc ⊂ acc:r
then (obj:r, acc:r ∪ ampacc) **else** protection error

Procedure Q acquires more access to the parameter object than the caller possesses. To control when procedure Q can be called to manipulate an object in a way that the caller cannot, the caller must have the prerequisite accesses named in the set reqacc; otherwise the access amplification fails.

EXAMPLE 4.1:

A process in E_* has APPEND access to file X. The procedure which actually performs the append operation requires both READ and WRITE access to the same file X. It gains this access by having a parameter specification of the form (ty=file, reqacc=APPEND, ampacc=READ,WRITE) so that the execution

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

environment of the append procedure will include the right (X,APPEND READ WRITE).

⊠

EXAMPLE 4.2:

Let storage environments be a type of object with accesses TO, FROM and READ. TO is an access used to transfer rights from an execution environment to a storage environment. FROM is the inverse of TO, and READ permits reading of the content of the storage environment. Assume that a process in E_* has only the right to READ the storage environment named S. The process passes the right (S, READ) as a parameter to a procedure which gains TO,FROM access to S through access amplification. By exercising the access FROM on S, rights may be transferred from S into the execution environment of the called procedure. The storage environment can be used as a carrying container whose contents cannot be removed or incremented except under control of the execution of a certain procedure. Thus, arbitrary sets of rights not available to the caller are made available to the called procedure.

⊠

This use of amplification is restricted in that it provides a called procedure with accesses to a parameter object that the caller did not have so that it can manipulate the object in a way that the caller cannot. Our model places no restrictions on the creation of types of objects. There exist some initially defined objects (for example, devices, memory and processors) having physical realizations. However, in general, objects are software defined and have representations formed from other component objects. To manipulate an object

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

requires rights to these components. To generalize our concept of an object, we let each object name be related to an ordered set of rights called **constituent rights**. The constituent rights can be though of as being contained in a storage environment never used as a site for execution. Each object X is composed of the set of objects named in the constituent rights of X. Because an object has constituent rights rather than merely component objects, access to the components can be controlled. Note also that nothing restricts two objects from having the same object as a component so that two objects may be merely different organizations of the same components.

To actually perform an access to an object requires knowledge of the structure of the object and access to the components which form that object. To construct an execution environment in which to perform an access requires acquisition of the constituent rights of an object named in a parameter right. The calling environment does not contain these rights because otherwise the caller could manipulate the components of the object directly, rather than via invoking the accesses defined for the object. If the environment in which the procedure will execute always contains the needed constituent rights for any possible parameter object, then it violates the 'need-to-know principle'. To meet the requirements of this principle, the constituent rights of a parameter object must be made available only for the duration of a specific performance of a procedure. There are several ways to do this.

The first kind of amplification, called uniform object amplification, computes all constituent rights. Let X be an object names and let constituent(X) be an environment containing $\{r_1,..., r_n\}$. In the following discussion, we assume that each object of a specific type has the same number of constituent rights.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Defn: **Uniform object amplification** of a right r generates all constituent rights of the object named in r.

Uniform object amplification depends upon a procedure parameter template defined as for TYCHECK, i.e. template(Q) = ty, reqacc, ampacc

The uniform object amplification parameter mechanism is

OBJAMP(Q,r) ≡ **if** type(obj:r) = ty ∧ reqacc ⊂ acc:r
then {constituent(obj:r), (obj:r, acc:r ∪ ampacc)}
else protection error

Evaluation of OBJAMP(Q,r) generates as actual parameters the constituent rights of the object named in r in addition to the amplified actual parameter right r.

A digression will now be made to define nonuniform object amplification in which the actual parameters generated are a subset of those computed by OBJAMP. The subset is controlled by the wishes of both the caller and the called.

Defn: **Nonuniform object amplification** of a right r generates a subset of the constituent rights of the object named in r as restricted by both the caller and the called.

Nonuniform object amplification divides the constituent rights associated with the object named in the actual parameter into two disjoint sets: those rights to be generated as actual parameters and the complementary set. The constituent rights which become actual parameters are those requested by the procedure (as recorded in its parameter specification), but limited to those rights that the caller wishes the called procedure to be able to use. Thus

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

either the caller or the called can prevent a constituent right from being made available to a called procedure.

The objective of this digression is to argue that the nonuniform object amplification mechanism can be described in terms of uniform object amplification and the primitives already in the model. Therefore the investigation to establish a parameter primitive for the model can be restricted to consideration of uniform object amplification.

Assume uniform amplification can be described in the model. Then nonuniform object amplification can be defined in terms of it. We extend the parameter template specification so that

template(Q) = ty, reqacc, ampacc, **s**

where **s** is a boolean vector of length n, and

n is the number of constituent rights for an object of type ty.

The vector s records the request by the procedure for constituent rights. The caller records the same information in a dynamically defined boolean **mask** (vector). The nonuniform parameter mechanism is defined as

NOBJAMP(Q,r,**mask**) ≡ **if** type(obj:r) = ty ∧ reqacc ⊂ acc:r
then {**for** i←0 **until** n **do if** (s[i] ∧ **mask**[i]) **then** constituent(obj:r)[i];
(obj:r,ampacc ∪ acc:r)}

where constituent(obj:r)[i] is the i-th constituent right of the object named by obj:r, and

mask is the boolean vector of length n provided by the caller.

The result of NOBJAMP is a set of rights E which contain the actual parameter right r after access amplification and a subset of the constituent rights of the object named in the parameter right r. Nonuniform object amplification can be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

described with the primitives of the model and uniform object amplification, since GRANT and COPY are sufficient to move selected constituent rights to the called environment.

Now all the actual parameter defining mechanisms have been introduced. The right transfer primitives are sufficient to describe the first several parameter mechanisms, as noted at the time of their definition. We will relate all the mechanisms in terms of the rights that they generate with particular emphasis on the two amplification mechanisms (object amplification and access amplification) to determine the definition of an amplification primitive to be added to the model.

In an effort to minimize the primitives required to describe amplification, we define

Defn: **Immediate closure(S,E)** is the union of state S and the set of all states resulting from exercising one right existing in environment E in state S.

The motivation for immediate closure is to be able to define a primitive to perform access amplification, then use it in conjunction with the right transfer primitives to describe object amplification.

Then

Defn: $p_i \leq p_j$ if immediate closure$(S,p_i(Q,r)) \subset$ immediate closure$(S,p_j(Q,r))$ in every case for which
1) both parameter sets are generated without error and
2) common fields in the parameter specifications for both p_i and p_j have the same value.

Defn: $p_i = p_j$ if $(p_i \leq p_j) \wedge (p_j \leq p_i)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The next four properties relate the parameter defining mechanisms in terms of $\leq$ and $=$. Note that in the following text naming a mechanism with arguments denotes the set of parameter rights generated by evaluating the mechanism and naming the mechanism without parameters refers to the mechanism definition, not an invocation of it.

Property 4.1: NONE $\leq$ IDENT .

Proof: The template is null for both NULL and IDENT. By definition NONE(Q,r) generates a null set of rights, thus NONE(Q,r) $\subset$ IDENT(Q,r). Since IDENT(Q,r) cannot be contained in NONE(Q,r) then we conclude that NONE $\leq$ IDENT.

$\boxtimes$

Property 4.2: IDENT = TYCHECK.

Proof: 1. By definition, TYCHECK(Q,r) $\subset$ IDENT(Q,r) if no error occurs. Therefore TYCHECK $\leq$ IDENT.

2. Let template(Q) = type(obj:r) then IDENT(Q,r) and TYCHECK(Q,r) generate the same set of rights thus IDENT $\leq$ TYCHECK.

We conclude that IDENT = TYCHECK.

$\boxtimes$

Property 4.3: TYCHECK $\leq$ ACCAMP .

Proof: 1. If template(Q) = (ty, null, null) then the definition of ACCAMP reduces to

ACCAMP(Q,r) $\equiv$ **if** type(obj:r) = ty **then** r **else** protection error

so that ACCAMP(Q,r) and TYCHECK(Q,r) generate the same set of rights. Thus TYCHECK $\leq$ ACCAMP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. To show that ACCAMP ≤ TYCHECK is FALSE, consider the case in which (ampacc ∪ acc:r) is a larger set of accesses then acc:r alone. TYCHECK is not able to generate new rights as ACCAMP does, but can only pass rights already in the calling environment. Thus ¬ (TYCHECK = ACCAMP).

⊠

Property 4.4: ACCAMP = OBJAMP.

Proof: By definition ACCAMP ≤ OBJAMP. To assist in showing that OBJAMP ≤ ACCAMP, we define OUT to be an access applicable to every type of object.

Procedure Out implements the OUT access, so that Out(X,j) transfers a right indexed by j from the constituent rights of object X into the current execution environment using GRANT or COPY as is appropriate. Recall that the constituent rights are kept in a storage environment related to object X.

To define (uniform) object amplification (OBJAMP) using access amplification (ACCAMP), we add OUT to the ampacc field of the parameter template of the procedure to be called. For every right s ∈ OBJAMP(Q,r) either s ∈ ACCAMP(Q,r) or s ∈ constituent(obj:r). In the latter case (obj:r,OUT) ∈ ACCAMP(Q,r) implying that s can be brought into the called environment by one use of the OUT access, thus s ∈ E in one or more states in immediate closure(S,ACCAMP(Q,r)) where state S is that state resulting from the performance of ACCAMP(Q,r). Therefore s ∈ OBJAMP(Q,r) implies
s ∈ E' ∧ E' ∈ S' ∧ S' ∈ immediate closure(S,ACCAMP(Q,r)).
Thus OBJAMP ≤ ACCAMP and ACCAMP = OBJAMP.

⊠

Summarizing, the relation of the parameter generation mechanisms:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NONE ≤ IDENT = TYCHECK ≤ ACCAMP = OBJAMP .

This predicate motivates the definition of an access amplification primitive. Let

struct template = ty, reqacc, ampacc

and define the primitive:

AMPLIFY(r,E,t) ≡ **if** type(obj:E[r]) = ty:t ∧ reqacc:t ⊂ acc:E[r]
then E[r] ← (obj:E[r],ampacc:t ∪ acc:E[r])

where t is a template.

AMPLIFY checks that parameter right names an object of the expected type and includes the required access subset, then generates a right to that object using the accesses specified in the template field ampacc of 't' to augment the original accesses. Intuitively AMPLIFY computes a new access field for a specific right, based on information outside the environment. It can be shown that no previously defined primitive can accomplish such a function. Observe that not all protection systems implement an amplification facility. A later chapter will treat one system HYDRA (Wulf et al [WU73]) which does. Amplification was motivated by parameter generation considerations. However the final form of the primitive does not rely on either parameters or boundary crossing and could be applied in other situations.

Protection and Execution

Having investigated in depth a set of parameter generating mechanisms, we now consider the setting in which these mechanisms are generally used; environment crossing by processes. To do so we first consider execution within environments. A programmed system is assumed to be a 'closed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

universe' in that all actions that affect the state of the system are taken by processes within the system. Consequently all actions that affect protection are also taken within the system. This section relates execution to protection in order to clarify precisely how and when environment boundary crossing is performed. The section surveys the space of environment boundary crossing policies and defines one which will be used in subsequent chapters.

A process sequentially performs operations. There exists an initially defined set of (hardware implemented) operations on objects that can be invoked, but whose means of performance is not specified further. For protected execution we assume that every process is associated with a single execution environment at all times. Every operation has a definition that specifies a set of rights which must be in the execution environment for a process to successfully perform that operation. We rely upon the execution mechanism (by which a process sequentially invokes hardware defined operations) to guarantee that any dynamic protection checking as required by the Enforcement Rule is performed for the hardware operations. The initially defined operations, when invoked, do not precipitate any environment boundary crossing for their means of performance is left unspecified.

In the simplest case only a single right is required to successfully perform an operation. For example, in order for a process executing in environment E to write into a memory cell object M requires the single right (M,WRITE) to appear in E. In this case the operation implements the algorithm which is associated with the WRITE access.

Complex procedures can be defined that require multiple rights in order to be successfully invoked.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

EXAMPLE 4.3:

Let EXCH be an access defined for cell objects. The intended use of EXCH is to allow a process to exchange the content of two cells, but not to be able to arbitrarily write on either. The operation Exchange(A,B) is to be successfully invoked only if both the rights (A,EXCH), (B,EXCH) are in the execution environment of the invoker.

⊠

Both initial and software defined operations can require multiple rights for their successful use. It is because multiple rights can be required before an individual right can actually be exercised (through invocation of a operation) that the (object name, access list) pairs have been called rights, rather than 'capabilities' as they are in earlier research papers (Dennis & Van Horn [DVH65]). A right is, in fact, just what its name implies: the permission to perform some action. Whether a process is actually 'capable' of exercising that right or invoking the operation that implements an access may depend on what other rights are in the execution environment of that process.

Creation of operations is not restricted to hardware designers. Software created operations are defined by procedures which are data structures that specify the rights needed to successfully invoke the procedure as well a program to be executed to accomplish the task defined by the procedure. Such operations can require use of rights not available at the procedure invocation site. In such cases the operation must be performed in a separate environment. To accomplish controlled environment boundary crossing, the initial operations are extended to include a set of environment boundary crossing operations. These operations differ among systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Environment Boundary Crossing Policies

To capture more precisely the definition of execution, let each environment be restricted to be the site of execution of no more than one procedure by a single process. Associated with each execution environment is a sequencing control object called an operation_counter whose value is used by the execution mechanism to distinguish the next operation to be invoked. Once created an operation_counter object is accessible only in the environment in which it was created. Both the environment and operation_counter are erased together.

Boundary crossing concerns issues of execution control (for example, co-routine entry or strictly nested procedure entry to program execution). Choices between the various kinds of execution control is not a consequence of the protection policies of a system. As an example we first consider co-routine execution control.

Specification: Coroutine control permits a process to suspend execution in one environment and cross to another existing execution environment to resume execution within that environment where it was last suspended.

Implementation: For each site of execution an environment is created with its associated operation_counter naming the entry point of the program to be executed in that environment. The (only) environment crossing operation $COCALL_{\&}(E)$ updates the operation_counter associated with $E_{\&}$. Then it causes the process to leave its current environment, $E_{\&}$, to continue execution by next invoking that operation named by the operation_counter in the now current execution environment E.

This example will be developed no further, but should serve as a contrast to the detailed example which follows. It is intended to suggest that the nested procedure discipline which will now be developed is arbitrarily selected and is not a consequence of our model of protection or execution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Execution Control by Nested Procedure Invocation

As a standard environment boundary crossing mechanism, later chapters use nested procedure invocation control. This requires two support operations:

CALL which creates a new environment fitted with rights needed by the called procedure, and then causes the calling process to begin execution at the entry point of the called procedure within the new environment.

RETURN which enables a process to reverse the environment crossing which brought it to the current environment.

All rights are extended to include the GRANT, COPY and DELETE accesses used in Chapter III to control the flow of rights into and out of environments. CALL and RETURN use the primitives COPY and GRANT to transfer rights between environments only if a right includes the access names, COPY and GRANT respectively.

Both CALL and RETURN transport parameters from the current environment to a target environment. Though CALL provides amplification of parameters, RETURN merely GRANTs indicated return parameter rights to the target environment. Since RETURN causes erasure of the RETURNing environment, COPYing and GRANTing of rights are equivalent.

For the definitions of CALL and RETURN, we require two data structure declarations:

struct procedure = **template** t[1:j], **env** d, entry

struct template = ty, reqacc, ampacc

A procedure is a data structure that consists of a parameter template list used by CALL to peform parameter checking and possibly access amplification, and an environment containing copiable declared rights (including one to execute the code object named by 'entry'). CALL is invoked from E_* with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parameters r_0, ..., r_n. CALL first creates an environment E_0 and sequentially transports to E_0 (via COPY or GRANT) the parameter rights indicated by the caller. AMPLIFY is applied to each parameter in case the procedure is to have access to the parameter object in addition to that passed by the caller. AMPLIFY uses the parameter template 't' as described earlier in this chapter. CALL is described below as though a procedure. The curly bracketed notation is the parameter template specification requiring that r_0 must include CALL access to the procedure object with the formal name P. (See Appendix A for further explanation of notation.) MIN produces the minimum among a list of numbers. LENGTH counts the number of entries in an environment. Storage allocation operations are notated as 'createenv' and 'create operation_counter'.

```
procedure CALL*(r0={P of procedure,CALL}, ..., rn) =
    begin createenv E0 ;
        for i ← 1 until MIN(n,LENGTH(t:P))
            do begin if COPY ∈ acc:E*[ri] then COPY(ri,E*,E0)
                        else if GRANT ∈ acc:E*[ri] then GRANT(ri,E*,E0)
                                else parameter error ;
                    AMPLIFY(ri,E0,(t:P)[i])
               end
        for i←0 until LENGTH(d:P) - 1 do COPY(i,d:P,E0) ;
        create operation_counter OC ;   OC ← entry:P ;
        COPY((E*,RET),E,E0) ; GRANT((Z of process,RUN),E*,E0)
    end
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CALL, as well as RETURN, is assumed to run in a fixed environment E at every use. Because all execution environments are created during CALLs, E includes the right to RET to each created execution environment. Thus CALL can place into a new environment a right to return (RET) to the CALLing environment. This non-grantable, non-copiable, non-deletable right is the means by which a multiple execution environments of a process are linked. RETURN is defined so that a process can cross back only to that one existing environment from which a CALL caused creation of the RETURNing environment. The right to RUN the process object is used to indicate that one environment in which a process is executing. Execution has been suspended in any environment that does not contain a right to RUN a process. Thus the execution environments of a process can be thought of as a stack, the top one containing the right to run.

procedure RETURN$_*$(r_0={E_0 **of environment**,RET}, ..., r_n) =

begin for i ← 1 **until** n **do if** (GRANT ∨ COPY) ∈ acc:E_*[r_i] **then** COPY(r_i,E_*,E_0) ;

COPY((Z **of process**,RUN),E_*,E_0) ;

erase E_*

end

The RETURN operation COPYs to the environment which caused entry to that RETURNing environment all parameter rights which includes GRANT or COPY access, as well as the right to RUN the process. Then RETURN erases the RETURNing environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Summary

This chapter defines in detail the operations to implement one environment boundary crossing policy based on nested procedure calls and returns. This environment boundary policy is used for development in subsequent chapters.

Environment boundary crossing is often accompanied by parameter passage. We define and then partially order a set of parameter right generating mechanisms in terms of the parameters each is capable of generating. A new kind of paramater passage called amplification is defined. Its purpose is to provide a controlled means to permit a called procedure environment with greater access to an object named in an parameter right than the caller possessed. Parameter right passing is examined in detail because parameter passage operations are short, well-defined and often executed, and therefore are candidates for implementation in microcode or hardware.

When objects are defined in terms of constituent rights, amplification is used to give a called environment all or a subset of the rights to manipulate component objects of which the parameter object is constructed. The definition of constituent rights provides a way to define several objects that have some, but not necessarily all, identical components. Two objects can represent different organizations of the same information. Further implications of this definition will be treated in the chapter on dynamic type creation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION OF THE PROTECTION MODEL

Chapter V

The objective of this chapter is to demonstrate that the structure of our protection model as developed in the first several chapters is an accurate tool for functionally expressing existing protection systems. As a framework for doing so, we observe three broad categories of user needs to which protection systems must be responsive and illustrate each by example. The three categories of user needs are:

-need to maintain private information

-need to communicate information to others

-need for several individuals to have access to the same information simultaneously.

Example operating system protection facilities that minister to these needs are used to illustrate each of the categories as well as to demonstrate that the model does indeed provide a tool for usefully describing diverse protection systems.

Isolation

We first consider the case in which all information (recorded within memory block objects) is private. An object is accessible from only a single environment and is private to the one process executing in that environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As an example, consider a memory protection system which completely isolates each of a fixed number of users from another. Let each user be a process executing at all times in a private environment. All objects are assumed to be predefined. We do not specify the mechanism of implementing the Enforcement Rule.

Two right transfer operations specify the right transfer policy. There exists a single central environment E in which unused memory blocks are collected. To return all access to a memory block M to the central environment E, a process running in environment E_* performs the right transfer operation defined by

procedure $Divest_*(M) = GRANT((M,all),E_*,E)$

Only the process in E is able to distribute a memory block M to a user environment E_i using the right transfer operation defined by

procedure $Allot_*(E_i,M)$ = if $E_* = E \wedge (M,all) \in E$ then $GRANT((M,all),E,E_i)$

Divest and Allot are the only right transfer operations which alter the content of an environment. We can observe immediately from these definitions that

1. no communication is possible between users if we rely upon the central process to erase the contents of a block obtained through a Divest-ing operation before Allot-ing the block to another user.
2. though users cannot communicate with one another a user can communicate with the central process by Divest-ing itself of a block containing a message.

For this system implementing isolation, the Environment Binding policy is null: no environment crossing occurs at all.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Although isolation is important, it is rarely implemented alone. At a minimum information is sequentially accessible in different environments as it is between users and the central process in this example.

Conservation of Rights

Users need to communicate information to one another. A simple way to accomplish this is to pass to another environment all rights to the object containing the information to be communicated. Rights are conserved, in fact use of objects is conserved. No more than a single process has the right to access a particular object at a time.

To illustrate conservation in a realistic way, we consider the actual example of transput stream processing in the T.H.E. operating system [D68]. This example is more ambitious than previous ones because it involves an application of protection, not merely the description of a protection system itself. It uses a right transfer operation that will be defined and the nested procedure execution control as implemented in CALL and RETURN defined in Chapter IV.

EXAMPLE 5.1:

T.H.E. Transput Stream Specification: Each programmed machine (PM) has a fixed number of initially defined one direction transput streams through which all input and output flows in the form of segments. For each stream there is at least one designated constant machine (CM) which services that stream (among

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

others) as required. The constant machines input or output sequences of segments called documents.

Implementation: Each constant machine and programmed machine is a process. Information is stored in segment objects to which all rights are grant only. Transput streams are storage environments in which rights to information segments are placed after they have been produced, but not yet consumed. Accesses TO, FROM, GIVE, GET, PSELECT, and CSELECT apply to environment objects. PSELECT and CSELECT are accesses that permit selection of a stream by a machine to produce or consume documents respectively. When a machine selects a stream it acquires the right to GET segments from the stream or GIVE segments to the stream. Within the GIVE and GET operations the accesses TO and FROM are available so that rights can actually be moved into or out of the environment that implements a stream. The right transfer operation for grant only objects is

procedure Transfer(r={X,GRANT}, r_1={F **of env**,FROM}, r_2={T **of env**,TO}) =
GRANT(r,F,T)

which uses the primitive GRANT to pass right r out of environment F into environment T. We use CALL and RETURN defined in Chapter IV to enforce the standard procedure nesting discipline. CALL enforces parameter specifications. (In fact the purpose of defining Transfer as a procedure is to obtain the checking performed by CALL for the three parameters required by Transfer.) For example, it insures that the second parameter names an object of type environment to which the caller has FROM access. The CALLed environment will be denoted E inside procedure definitions. It is assumed that a process executing in an environment always has TO and FROM access to that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

environment.

To add segments to or remove segments from a stream, the PMs and CMs invoke the procedures Give and Get respectively. Only these procedures can invoke Transfer as indicated by the declared rights of their procedure definitions. Successful invocation of Get (or Give) for a transput stream E, occurs only if the caller has the right (E,GET) (or (E,GIVE)).

procedure Get(r={E_i,**of env**,GET GRANT→FROM}) = (X,CRIT),(Transfer,CALL);
critical X {select right (S **of seg**,GRANT) in E_i; Transfer((S,GRANT),E_i,E)};
RETURN((S,GRANT),(E_i,GET GRANT))

procedure Give(r={S **of seg**,GRANT},r_1={E_i **of env**,GIVE GRANT→TO}) =
(X,CRIT),(Transfer,CALL);
critical X {Transfer(r,E,E_i)}; RETURN((E_i,GIVE GRANT))}

The right to GET (GIVE) segments to a stream environment is grant only thus must be RETURNed to the caller. We use the language construction **critical** to indicate code sequences that are uninterruptable with respect to one another. Here we use a critical section so that only one process can be altering a stream at a time.

To control the capability of adding and removing segments to streams, a machine must acquire GET and GIVE accesses. PSELECT (CSELECT) allows a machine to select a particular stream to begin document production (consumption). Each CM is initialized to have the rights (E_i,CSELECT COPY) for each user transput stream i from which the machine can ever consume segments. Similarly producer CMs possess a right (E_j,PSELECT COPY) for each transput stream j for which they can produce a document. In addition each CM has one grant only right (SELCT,GRANT) used as a token to insure the CM selects no more than one stream at a time. Since all streams are private to one

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PM, each PM is initialized to be able to access each of its private streams with either CSELECT or PSELECT. PMs are also initialized to have multiple (SELCT,GRANT) tokens so that they concurrently process multiple streams.

The document selection procedures use a common table segment called MARK (with accesses R (read) and W (write)) to remember whether a particular stream is currently selected for consumption or production. Stream selection for consumption is performed by

procedure SelectC(r=(SELCT,GRANT), r_1={E_i **of env**,CSELECT COPY→GET GRANT}) =
(MARK,RW),(Y,CRIT),(UNSELCT,GRANT);
critical Y **if** E_i unselected stream for consumption
then {mark E_i selected; RETURN((UNSELCT,GRANT),(E_i,GET GRANT))}
else RETURN((SELCT,GRANT))

The procedure SelectC used to select a stream for consumption requires the token (SELCT,GRANT) which the caller loses and the right to CSELECT a stream. By amplification performed at CALL time the called environment in which SelectC executes includes a right to GET segments from the transput stream named by the caller. If selection is possible, the right to GET segments will be RETURNed to the caller along with an UNSELCT token. Otherwise the SELCT token is RETURNed to the caller so that it can attempt selection of another stream.

Selection for production is performed similarly by

procedure SelectP(r=(SELCT,GRANT), r_1={E_i **of env**,PSELECT COPY→GIVE GRANT}) =
(MARK,RW),(Y,CRIT),(UNSELCT,GRANT);
critical Y **if** E_i unselected stream for production
then {mark E_i selected; RETURN((UNSELCT,GRANT),(E_i,GIVE GRANT))}
else RETURN((SELCT,GRANT))

Procedure Unselect removes the relation between a CM and a selected stream if the caller returns the UNSELCT token and a right to either GIVE or GET segments.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
procedure Unselect(r=(UNSELCT,GRANT), r1={Ei of env,null}) =
                        (MARK,RW),(Y,CRIT),(SELCT,GRANT);
    if (GIVE ∨ GET) ∈ acc:E[r1]
        then {mark Ei unselected; RETURN((SELCT,GRANT))}
            else error
```

In this example not only are rights to segments conserved, but the use of segment objects is conserved--i.e. at all times only a single environment contains access to a specific segment.

Conserving objects is sufficient to permit processes to use one anothers work. One process is able to copy the contents of either program or data objects and grant access to the new object to others so that both have access to the information. Object conservation is a sufficient facility for programming arbitrary patterns of communication between processes.

Because we have purposely embedded so much of the control of actions in the movement of rights, it is quite easy to show that several properties are true.

Property 5.1: A CM can service only an initially fixed set of streams.

Proof: A CM is intialized to have CSELECT or PSELECT access to a set of streams. It could acquire rights to select other streams only

1. by calling a procedure that RETURNs such a rignt and by inspection, no procedure does.
2. by removing such a right from a transput stream environment which the CM consumes. All streams are initially empty and are filled using the Give procedure. Whenever Give is successfully CALLed, the parameter right to be transferred into a stream environment always names a segment, therefore environment rights do not appear in transput streams.

A CM cannot gain the right to service additional streams. ⊠

Property 5.2: A CM selects one transput stream at a time.

Proof: To select a transput stream a CM requires the right (SELCT,GRANT).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

By successfully selecting a stream a CM loses the right (SELCT,GRANT) and gains (UNSELCT,GRANT). Since transput streams only contain rights to segments, the only way a CM can regain (SELCT,GRANT) is through a successful CALL to Unselect which causes loss of the token (UNSELCT,GRANT). Unselect cannot be CALLed again until another UNSELCT token is acquired. Since a CM can only have one token at a time, it can select at most one stream at a time.

⊠

Property 5.3: A segment is sequentially available to the PM and a CM in an order dependent upon the direction of the stream through which it travels.

Proof: We consider an output stream first. The producer PM creates a segment of information. To GIVE the segment of the output stream, the PM must grant its right through CALL to the environment in which Give executes. The CALL to Transfer causes the segment right to be eventually granted into the output stream environment. The right is inaccessible to the PM, but a CM can gain access to it for consumption only by granting it out of the stream via Get.

Similarly we can trace the grant path of a segment created by a CM to an input stream and then to the execution environment of a PM. Thus the grant only path enforced by the parameter specification for Get, Give and Transfer cause Property 5.3 to be true.

⊠

Sharing

By nature an operating system permits sharing of resources. The third requirement that influences the protection of objects is that multiple processes need simultaneous access to the same information. Sharing by an unlimited number of processes is achieved by allowing duplication of rights (via the COPY primitive.) Vastly different phenomena occur depending on which access rights are duplicated.

Assume that all rights to shared objects specify non-altering accesses (i.e. the associated algorithm does not cause changes in the content or structure of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the object). Then the effect of sharing an object from the perspective of the users is equivalent to copying that object (creating new objects which contain the same information) and granting non-alter access to the new objects to other processes. All processes have read access to the same information. None can alter it so none can detect whether the same resource is being shared as well as the same information. Sharing of pure procedures is a common and useful instance of this category of sharing.

Where an arbitrary number of processes share access to a data base, unlimited interaction is possible. Note that the effect of a shared data object can be provided by a protection system which only permits conservation by defining the data base to be a private resource of one process which accepts and acts upon requests from other to alter or read the data base.

EXAMPLE 5.2:

We elaborate the file system example 2.5.

Directory Structure Specification: The file system is a rooted directed graph with files as terminal nodes and directories as non terminal nodes. Each directed arc from a directory to another node represents the ability of a process with that directory to perform some operation on the other node.

A user can permit others to share access to his files, yet still retain the ability to retract that permission.

Implementation: Files are objects with unspecified accesses. Directories are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

objects with accesses ADD, REMOVE and INDIRECT. Each directory is a table of rights to directory or file objects. A directed arc from a directory corresponds to a right in that directory and terminates in the node named in that right.

Rights exist only in directories or execution environments. Whenever a user enters the system, a process, created to represent that user, is given a right to at least one user directory.

File system operations permit rights to files and directories to be moved within the directory structure and from the directory structure to execution environments, but not from environments to directories.

A process can remove a right from a directory D only if the process has the right (D,REMOVE) and augment a directory D only if it has the right (D,ADD). The appropriate operations are defined by

```
procedure Remove(r = {D of dir,REMOVE},s,a) = ... ; †
                DELETE(s,D,a)

procedure Add(r = {D of dir,REMOVE},r1 = {D' of dir,ADD},s) = ... ;
                COPY(s,D,D')
```

A process can create a new directory (and similarly a file) with the operation

```
procedure Createdir(r = {D of dir,ADD}) = ... ;
            allocate directory representation ; CREATE(DIR,D)
```

which places a right to perform all access to the new directory in an old directory specified in the actual parameter.

† The declared rights for the file system procedure are not shown. In addition, we assume (rather than program) each operation to be an indivisible operation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A process acquires a right from the directory structure using

procedure Acquire_{E_*}(r = {D **of dir**,INDIRECT},s) = ... ;
if D[s] exists **then** COPY(s,D,E_*)

where E_* is the execution environment of the caller.

By repeated use of Acquire, a process with right (X_1,INDIRECT) in its execution environment can obtain use of the right (X_{k+1},A) where X_1, ..., X_k name directories and for all i, $1 \leq i \leq k$ (X_{i+1},INDIRECT) $\in X_i$.

To initialize the possibility of sharing, each directory D established for a new user identity (though not every directory) contains the rights (D,REMOVE ADD INDIRECT) and (DSHARE,REMOVE ADD). Directory DSHARE is a system directory used for initializing the flow of rights between user directories. A user can Add the right (D,ADD) to the DSHARE directory for other users to REMOVE and use to send rights to directory D.

Property 5.4: Users can share different accesses to the same files.

To share the right (F,A) to a file F in directory U1 with a second user having directory U2, we create a new directory by the name of D so that all rights to D are in U1:
Createdir((U1,ADD)).
Add((U1,REMOVE),(D,ADD),(F,A)) is then used to move (F,A) into the newly created environment.
Add((DSHARE,REMOVE),(U1,ADD),(U2,ADD)) provides U1 with the right to add a new right to the directory U2.
Add((U1,REMOVE),(U2,ADD),(D,INDIRECT)).

Processes with access to U2 can indirectly acquire the right (F,A). ☒

Property 5.5: A user can retract permission to share access to an object.

Proof: Add directory and file rights are acquired from the directory structure when a new user enters the system. Since rights can only be moved within the directory structure or into execution environments, but not from execution environments to the directory structure, once a user exits and reenters the system, he must reacquire all directory and file rights

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from the directory stucture. Therefore a user can retract permission to share access to an object, but retractions only take effect in between user sessions.

⊠

For this example of a file protection system, the Right Transfer Rule is bound by the definitions of the operations Add, Remove and Createdir, as well as the operation to create files that was not defined. The Environment Binding Rule is bound by Acquire which permits a process to obtain a right to another directory. A process can be thought of as executing in the sum of the directories for which its has access rights in its execution environment.

⊠

In summary the preceding discussion can be reduced to tabular form:

Protection facility provides: \ Object Use Phenomena:	communication	shared objects	multiple processes share same access to objects
isolated use of objects	no	no	no
conserved use of objects	yes	no	no
conserved rights	yes	yes	no
duplication of rights	yes	yes	yes

Additional Examples

In addition to the previous examples, two more have been selected to add additional diversity: The first example is of hardware memory protection and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the second considers protection in a programming language as exemplified by PASCAL. Besides describing systems, we can use the model to compare them. In addition, Appendix C contains a comparison of the construction of the environment in which to execute a called procedure for two very different systems: a Virtual Memory system (Price [P73]) and HYDRA (Wulf et al [W73]).

EXAMPLE 5.2: OS/360 Memory Block protection

Specification: Each process ('task' in the OS/360 terminology) has a program state word containing a four bit key. Each memory block has a four bit key associated with it as well as a store-fetch bit determining whether store protect or fetch and store protect are both in operation.

A process can access a cell in a memory block if its program state word † contains a key matching that of the memory block and the access desired is permitted by the store-fetch access bit for that block. A program key p matches a memory block key k if p = 0 or p = k.

Protection Description: All objects are memory blocks with accesses FETCH and STORE. Each key distinguishes a different environment E_0, ..., E_{15}. Environments are disjoint with the exception of E_0 which contains all rights at all times, thus contains all other environments. (This means that we assume all execution in key zero is also in privileged mode as is usual in OS/360 but not forced by the hardware.[IBM])

The Enforcement Rule is implemented within the instruction cycle program. The right transfer policy is implemented in one instruction: the Set Storage

† The key is found in the channel address word if the process is 'running' on a channel processor.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Key (SSK) instruction which is performed only in privileged mode and permits a storage key to be altered. If M names a memory block, k a key and f is a boolean indicating whether fetching is protected or not, then

procedure SSK(M,k,f) = **for** n ←1 **until** 15 **do** DELETE ((M,all),E_n,all);
if ¬f **then for** n ←1 **until** 15 **do** COPY((M,FETCH),E_0,E_n);
COPY((M,STORE FETCH),E_0,E_k)

The Environment Binding policy is also implemented by a single privileged instruction called Load Program State Word (LPSW). The LPSW instruction causes a process to cross from its current environment to another by resetting the key in the program state word of the process. A process can be forced to cross to a new environment by an interrupt which loads the program state word including the program key value which is copied from a fixed location.

This example differs from most used in this thesis in that it shows a process in one environment altering the execution environments of other processes as well as environment boundary crossing triggered by an event (an interrupt) outside the process.

⊠

EXAMPLE 5.3: Protection in programming languages

As a concrete example, we describe some aspects of protection in the programming language PASCAL (Wirth [W71]). PASCAL defines a fixed number of variable (object) types and the operators applicable to them. No distinction is made between operators which require multiple operands and accesses applicable to a single object since, as in most programming languages, if a variable is accessible at all, then any applicable operation can be performed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

on it.

Some example types and accesses are:

Types	Accesses
real scalars	+,-,*,/,min,max
file	put,get,reset
record	field select
powerset	intersection, complement
	union, membership

The compiler is responsible for observing the Enforcement Rule. Protection checks at compile time predict the environment at run time (by keeping a symbol table.) For each operation compiled the compiler determines the type of each object used and generates code to perform only the appropriate operation on that object. (PASCAL requires that the type of an object be known at the time a use of it is compiled.)

The only right transfers not associated with environment binding are declarations of new variables. During each declaration of a new object (for example 'P: **powerset** ...'), any current object with local name P is deleted from the current environment E (in terms of our primitives: DELETE (P,E,all)) and a new object of type powerset is created (CREATE (POWERSET-TYPE,E)) and made available in the environment of declaration by the local name P.

Environment boundary crossing occurs at procedure invocation and return. Procedure declarations are lexically nested. When a procedure is declared a template environment can be established for it consisting of all rights in the environment of the declarer at the time of declaration. Calling a procedure, say Q, that has template environment T, the execution environment is constructed by copying each right in T to E_Q. In our primitives COPY(r,T,E_Q). Then for each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

formal f (if any) DELETE(f,E_o,all) and COPY(a,E_0,E_o) where a is the result of evaluating the parameter and copying it from the environment of the caller, E_0.

Normal procedure exit causes the process to cross back to the calling environment. A procedure exit can also be taken by executing go to an exit label which causes a return to the most recent calling environment where that label is defined. Exit from an environment causes it to be inaccessible, thus effectively erases it.

⊠

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

THE BOUNDARIES OF EXECUTION

Chapter VI

In previous chapters, the model of protection has been defined and investigated, then used to describe example protection facilities. The next task and one which occupies the remainder of this thesis is to capitalize on the structure that protection contributes to the systems in which it is embedded. In this chapter we observe that environments provide external structure that can be used as a basis for developing a technique to state and prove properties that restrict execution. These properties depend upon the given protection system and the set of initial environment values.

To illustrate the external structure that protection provides, two alternative programs will be presented: one in which controlled use of objects is implemented internally in terms of the program's use of shared variables and the other in which the external structure is provided by a protection system. The programs are a channel program and a command sending program, known as the interrupt handler. They share a buffer called CMD (short for 'command').

A first version of the two programs requires a variable BUSY which has a value of 1 if the channel is performing or may perform the CMD, and 0 otherwise. BUSY is initially 0.

```
CH: repeat while BUSY = 0 do ;          IH: repeat while Busy = 1 do ;
        obey CMD;                               write new CMD;
        BUSY ← 0                                Busy ← 1
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Property 6.1: The channel and interrupt handler alternate exclusive use of CMD.

Proof: If BUSY is initially 0, all possible alterations of BUSY are from execution of these two programs, and the channel and interrupt handler initially begin execution at their respectively labeled statements, then the theorem can be proven. BUSY may take on only two values. If BUSY = 0 (1) then the interrupt handler (channel) eventually performs its second statement after which it sets BUSY to 1 (0), thus preventing itself from accessing CMD again until the channel (interrupt handler) has had the opportunity to complete its accesses to CMD and reset BUSY. ☒

The proof of Property 6.1 relies heavily upon the known **internal structure** of the two programs involved. Both programs assume that the other reads and resets BUSY in a specified way--for example it would be most unfortunate if one program set BUSY to 2. In fact each program depends upon itself in the same way. But the internal structure of individual or cooperating programs may not be verifiable or even predictable. If such were the case in this example, **external structure**--external to the content of the programs--may be imposed to guarantee the validity of the property.

Assume that the channel and interrupt handler execute their programs in environments E and E' respectively. Define one right (CMD,R W) to be in E' initially. (R and W represent read and write access to the object named CMD.) Define the procedure Give to permit a process to transfer rights from its execution environment, E_s, to any other environment. The procedure Give is the only environment altering operation permitted:

procedure $Give_s(r,E_d)$ = GRANT(r,E_s,E_d)

The channel and interrupt handler programs can be redefined as:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CH: **repeat while** ¬((CMD,R W) ∈ E) **do** ;
obey CMD;
Give$_{\ast}$((CMD,R W),E')

IH: **repeat while** ¬((CMD,R W) ∈ E') **do** ;
write new CMD;
Give$_{\ast}$((CMD,R W),E)

This solution looks almost like the first solution. The difference is that environments may not be arbitrarily altered as was possible with the variable BUSY. No matter what aberrant course of execution the channel or the interrupt handler pursue, there can exist only one copy of the right to access CMD and thus this right exists in a single environment and the property is true.

Since we also assume that reference to CMD is allowed only if the right (CMD,R W) is in the environment of execution, then because rights are conserved by the procedure Give only one program can access CMD at a time. In the first solution, CMD was always available for access by either the channel or the interrupt handler and cooperative agreement restrained invalid accesses. In this solution the agreed use of CMD is externally, rather than internally enforced.

⊠

Protection States

Environments provide a basis for stating properties restricting execution. A process executing within a single environment sequentially performs operations as permitted by the Enforcement Rule, which causes rights in the execution environment to be read, but not altered. We assume for convenience that all protection checking is performed dynamically. Performance of protection checks earlier than at the time of the exercise of a right is efficient,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

but otherwise equivalent as discussed in Chapter II.

The choice of a right to be exercised next is limited to a finite number since the number of rights in an environment is always finite. Because execution is limited by the content of the environment in which it takes place, restrictions on possible changes in the content of that environment are also restrictions on future execution within that environment and are **independent** of the code executed.

The result of some operations is to alter environment type objects. We assume that all environment alterations occur sequentially and instantaneously, including the crossing of a process between environments. As in the chapter on right transfers, we define a **protection state** S as a set of environments with fixed values.

Let **T** be the set of all environment transforming functions defined by a protection system including the null function. Recalling the definitions of state derivation in Chapter III, we say that for every $t \in \mathbf{T}$, state S **directly derives** S' ($S \Rightarrow S'$) by t, if S' is the state resulting from performing the transformation t in state S. If t is a parameterized function, then all parameter rights for each use of $t \in \mathbf{T}$ are restricted to come from a single environment. Graphically each state can be represented as the value of an access matrix indexed by environments and objects. Each transformation function can be regarded as an operator which alters the matrix representation of the protection state in a well defined way.

S **derives** S' ($S \overset{*}{\Rightarrow} S'$) by the transformation sequence $t_1, \ldots, t_k$ if there exists a sequence of states $S = S_0, \ldots, S_k = S'$ such that $S_{i-1} \Rightarrow S_i$ by t_i,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$1<=i<=k$. The transformation sequence $t_1, ..., t_k$ used to derive S' from S is called a **transformation path** from S to S'. The set of all possible protection states can be divided into two disjoint subsets: those derivable from an initial state S and the complementary subset.

Defn: The **closure of state** S, denoted closure(S), is the set of all states derivable from S.

EXAMPLE 6.1:

To demonstrate an example of a finite closure, let the transformations be the protection primitives excluding CREATE and COPY. Each transformation specifies a well-defined change in one or two environments in a protection state.

T = {READ, DELETE, GRANT, AMPLIFY}.

Let S be an initial set of environments $E_1, ..., E_n$.

Property 6.2: The closure of S under T is finite.

Proof: READ does not alter a state, thus can be ignored.
Given any initial state, only a finite number of DELETE operations can be performed before all environments are empty.
GRANT can be applied to derive only a finite number of states with the originally given rights rearranged among environments.
Similarly, if AMPLIFY is used to alter a right existing in an environment to derive a new state, only a finite number of alterations are possible (limited by the number of accesses potentially applicable to each object.)
Since the number of combinations of a finite set is finite, closure(S) is finite. ☒

EXAMPLE 6.2:

Closures are not always predictably finite. If CREATE, COPY $\in$ T, then closure(S) includes an indeterminate number of states, because using CREATE, an arbitrary number of rights to new objects (including environment objects) can be created, and an environment can be extended to an arbitrary size by copies of rights.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In most, though not all, applications we are interested in partial states. We are particularly interested in the state transitions involving rights to specific already-created objects or where limitations are placed on creation of objects or duplication of rights; consequently indeterminately large closures can often be avoided or abstracted from, as will be apparent in later examples.

⊠

Each transformation function in T corresponds to an executable operation which implements it. We speak of executing the transformation function in the environment from which its parameters are drawn. By definition, a process executes in one environment at a time, and can suspend execution to cross to another environment. Thus a protection state can include environments in which execution is not possible. It must be guaranteed that transformations are made only if the corresponding operation could be executed, i.e. only if execution can occur in the environment from which the transformation parameters are drawn. To insure this, we use the convention define in Chapter IV to indicate the execution environment of processes. For each process P we define a single right, (P **of process**,RUN) which cannot be duplicated, deleted or granted. The presence of this right in an environment indicates that a process is executing in that environment.

For every use of a transformation function in a state derivation, $S \Rightarrow S'$ by t, a right of the form (P **of process**,RUN) must be in the environment in which the transformation function t is performed. An environment crossing by a process is easily represented by granting the right to run to the target environment. Now the tools are available to exhibit concrete examples of closures. We assume an environment boundary policy based on the the nested

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

procedure discipline as defined in Chapter IV in the operations CALL and RETURN. We also assume that the access field in every right is extended to potentially include the accesses: GRANT, DELETE and COPY which are used to control manipulation of the rights themselves.

Let an initial state S with a single environment E be given:

	process P	env E	procedure Q	file F	file H
E:	RUN		CALL	R W SHARE	

Procedure Q is defined as

procedure Q(r = {Z **of** file,RW}) = (H **of** file,R GRANT); †
begin ... **end**

requiring as a parameter the right to R (Read) and W (Write) an object of type file and having the declared right to read a specified file named H. The effect of executing a call to Q((F,RW)) is to invoke the CALL operation which transforms state S to S_1:

	process P	env E	procedure Q	file F	file H	env E_Q
E:			CALL	R W SHARE		
E_Q:	RUN	RET		R W SHARE	R GRANT	

Process P is executing in environment E_Q as indicated by the right (P,RUN) in E_Q.

† Refer to Appendix A for explanation of procedure definition notation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Any state directly derivable derived from S_1 results from a transformation which is performed in E_Q, for no execution can occur in environment E in state S_1. The CALL operation which created E_Q inserted into it a right to return (RET) to E. That right cannot be copied, granted or deleted from E_Q.

One possible return from E_Q to the initial execution environment is RETURN((E,RET),(H,R GRANT)) resulting in state S_2:

	process P	env E	procedure Q	file F	file H	env E_Q
E:	RUN		CALL	R W SHARE	R GRANT	
E_Q:				R W SHARE		

Note that because of the nested procedure discipline the called environment is no longer accessible. If it is not a manipulable object from some other environment, then the presence or absence of E_Q is irrelevant and the state can be simplified by removing it. $S_2 \equiv S_3$ and S_3 is defined as:

	process P	env E	procedure Q	file F	file H
E:	RUN		CALL	R W SHARE	R GRANT

Now we continue with several realistic examples.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 6.3: Access History of File Objects

The closure of an initial protection state can be used to deduce restrictions on future execution without knowledge of the code actually executed. These properties restricting execution depend upon the definition of T and the initial state.

As an introductory example of the use of closure, we derive a property that states that the sequence of accesses to a file object is generated by a certain regular expression. For each object type there exist procedures implementing access algorithms that define how instances of that object type can be manipulated. The procedures direct the way in which new environments are initialized. In particular a procedure definition dictates the conditions for access amplification. To deduce properties about the possible patterns of accesses to an object, we assume that the definitions of all such procedures are known. We assume that all general environment manipulating operations defined by a protection system are defined to be in T.

File Problem Specification: A file is referenced by the operations R (Read), W (Write), O (Open), C (Close). The structure of a file and the meaning of the operations performed on it are specified no further. A file is accessible to only one user who performs sequences of R or W operations on it. Each sequence begins with a single O operation and terminates with a single C operation.

Objective: Use closure to show that the access history of an individual file is generated by the regular expression:

$$(O\ (R \vee W)^* C)^* .$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

File Implementation: A file is an object accessible by R, W, O, C. The file subsystem defines:

procedure R(r = {X **of file**,R GRANT}) = RETURN((X,R GRANT))

procedure W(r = {X **of file**,W GRANT}) = RETURN((X,W GRANT))

procedure O(r = {X **of file**,O GRANT → RWC}) = RETURN((X,RWC GRANT))

procedure C(r = {X **of file**,RWC GRANT → O}) = RETURN((X,O GRANT))

Any changes these operations make to the content of a file object are unspecified here.

Define a file user to be a single process. (This strict definition will be relaxed in the next example.) Because we are interested in the access history of individual files the initial protection state consists of one environment, E, containing the initial right (F,O GRANT) for file F, as well as the rights (Q,CALL) where Q = R, W, O, C. (As in the past we use the same names for accesses and the procedures implementing the algorithms associated with the accesses.)

File Analysis: Let **T** = {CALL, RETURN}. The initial state S is

	process P	**env** E	**file** F	O	C	R	W
E:	RUN		O GRANT	CALL	CALL	CALL	CALL

Because CALL (and RETURN) are defined to be indivisible, the only state which can be directly derived from S results from CALLing O((F,O GRANT)) in environment E deriving S_1:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	process P	env E	file F	O	C	R	W
E:				CALL	CALL	CALL	CALL
E_0:	RUN	RET	RWC GRANT				

By definition of O, the only state which can be directly derived from S_1 comes from performing RETURN((E,RET),(F,RWC GRANT)) deriving state S_2:

	process P	env E	file F	O	C	R	W
E:	RUN		RWC GRANT	CALL	CALL	CALL	CALL

CALL enforces nested procedure execution and the file operations perform unconditional actions when invoked. Therefore since a RETURN always succeeds a CALL, we can treat the file operations as indivisible. **T** can be redefined so the T = {R, W, O, C}.

From the analysis above $S \Rightarrow S_2$ by O((F,O GRANT)). By similar analysis $S_2 \Rightarrow S_2$ by R((F,R GRANT)) or W((F,W GRANT)) and $S_2 \Rightarrow S$ by C((F,RWC GRANT)). Thus closure(S) = $\{S, S_1, S_2\}$.
Parameters can be omitted because there is only a single parameter value for each operation (thus C((F,RWC GRANT)) is denoted as C.)

Then $S_2 \stackrel{*}{\Rightarrow} S_2$ without leaving state S_2, only by the transformation paths $(R \vee W)*$. $S \Rightarrow S_2 \Rightarrow S$ by the path (O C) and $S \stackrel{*}{\Rightarrow} S$ by paths $(O\ (R \vee W)*\ C)*$.
Each transformation function accesses file F and all accesses to F are included in **T**, thus the access history of file F at any time after S is initialized as given

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

by the regular expression (O (R ∨ W)* C)* .

There was a natural assumption in the this example that the closure(S) = {S,S1,S2}, while state S1 actually represents an indeterminate of states. Each call to O from inside envionment E causes creation of a distinct new environment in which to perform O. But it is sufficient (for the properties to be proven in this example) to assume that the indeterminate number of states can be treated as a single state. Judicious use of simplifying assumptions allows the states to be partitioned into equivalence classes so that only one member of each class need be considered. Whether such simplifying assumptions can be made dependent on the properties that are to be investigated.

Example 6.4: Extending the File Problem

We extend the file problem to illustrate a more complex case of using simplying assumptions to reduce a closure of indefinite size to one of a known finite size set of equivalence classes.

Let a user be a hierarchy of cooperating processes related by a family tree. Each process gets its name from the initial environment in which it executes. Rights can be transported between the home environments of a parent process and an immediate child using the procedure Give.

procedure $Give_i(r,j)$ = **if** (i is child of j ∨ j is child of i) ∧ GRANT ∈ $acc{:}E_i[r]$
then $GRANT(r,E_i,E_j)$

Let the initial state S include all processes executing in their home

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

environments. A single copy of the right (F,O GRANT) appears in S. Although a parent can have an arbitrary number of children which in turn can create their own child processes, we need only consider the parent, one child representing all descendents of the parent and one environment to represent all environments in which O or C execute.

Define two states for file F:

open <=> (F,R GRANT),(F,W GRANT),(F,C GRANT) each appear in a single user environment

closed <=> (F,O GRANT) appears in one user environment.

Property 6.3: File F is either open or closed.

Proof: Let T = {CALL,RETURN,Give}.
(We assume that Give, as well as CALL and RETURN, are indivisible.) Assume there exists some state $S_2 \in$ closure(S) in which F is neither open nor closed. Consider any transformation path from S to the first occurrence of S_2, i.e. the transformation path for $S \Rightarrow S_1 \Rightarrow S_2$. F is either open or closed in S_1. Now determine what transformation yields S_2 from S_1:

1. Since Give transports rights and does not add, alter or delete any right, it cannot affect the state of file F.
2. No right in S is can be copied. CALL introduces no shared rights through its use of a procedure template and RETURN can only conserve the grant only rights it passes.
3. All file rights are conserved across a call to R or to W.
4. Neither O nor C are used to derive a third file state for R, because each converts one of the two given states to the other.

This exhausts the possible transformations, thus there is no transformation path to S_2, and F has no states other than open and closed. ⊠

Observe that O and C are indivisible with respect to all file subsystem operations on a specific file (as well as the Give-ing of file rights) because a process performs operations sequentially and by Property 6.3, a file has only two states. O and C cause transition between the states by replacing all rights to the file.

For simplicity we can let T = {CALL,RETURN,Give,O,C} where CALL now

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

applies only to R and W. Since CALL is indivisible, invocations of R and W are strictly ordered. (However the duration of a R and a W operation can overlap. Any synchronizing of the two must be embedded in the operations themselves.)

Property 6.4: The access history of a file is given by the regular expression

$$(O\ (R \vee W)^* C)^* .$$

Proof: A file is initially closed. The only access possible is O which causes the file to enter the open state.

While open, a file has an access history described by
O (CALL((R,CALL),(F,R GRANT)) ∨ CALL((W,CALL),(F,W GRANT))*
where CALLs to R and W are strictly ordered. Rewrite the access history as O (R ∨ W)* . F alternates states since by Property 6.3, it enters the closed state if and only if it leaves the open state and vice versa.

Therefore the access history of a file is generated by
(O (R ∨ W)* C)* .

⊠

Intuitively the reason the access history could be described as a regular expression was because change of the protection state depended on the value of one environment (the source of the transformation parameters.) Next we consider a more complex situation in which the protection state is altered dependent on the content of the entire protection state, not just the value of the environment in which the change is requested.

Example 6.5: Complex File Access History

Problem Specification: Define a file system to provide files with R (read) and W (write) operations so that

1. one user can perform an uninterrupted sequence of W's to a file,
2. multiple users can perform interleaved W-free sequences, and
3. users may do a peek: a W-free sequence of one R. (Peeks are motivated by wanting some users to be able to read information

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

without keeping the file busy for longer than the tolerable duration of a single read (R) operation.)

Implementation: A file object is can be manipulated using accesses: R, W, OPENW, OPENWF, CLOSEW, CLOSEWF. Each access is implemented by a procedure of the *same name*. An uninterrupted X-sequence is parenthesized by OpenX and CloseX, where X is 'W' or 'WF'. A Peek is the sequence: OpenWF; R; CloseWF.

The state of a file F, state(F), has three possible values: free, w, and wf. Let X, A and S name an object, access and state respectively. We speak of a right being in a protection state. (X,A) $\in$ S if (X,A) is a right in at least one environment in S.

Define

state(Z) = free <=> $\neg$ ((Z,R) $\in$ S $\vee$ (Z,W) $\in$ S), i.e. in state S no user has the right to read or write file Z.

state(Z) = w <=> (Z,W) $\in$ S, i.e. in state S a user has the right to write file Z.

state(Z) = wf <=> (Z,R) $\in$ S, i.e. in state S a user environment contains the right to read file Z.

Each right of the form (F **of file**,R) or (F,W) corresponds to one user. A group of processes cooperatively exercise one right passing it back and forth between them in order to act as one user. In the initial state S_0 each environment contains some subset of the rights: (F,Q COPY) where Q $\in$ {OPENW, OPENWF, PEEK} and (P,CALL) where P $\in$ {R, W, Peek, OpenW, OpenWF, CloseW, CloseWF}. Therefore, initially state(F) = free.

Because OpenW, OpenWF, CloseW, and CloseWF exist only to alter the protection state, we assume that they *are performed* sequentially, i.e.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

indivisibly. We do not consider protection states containing the environments in which these procedures are performed. When the procedures OpenW and OpenWF interrogate the state, their own execution environment is not included in the state.

OpenW requires a parameter file to which the caller has OPENW access. By amplification, OpenW acquires the right to W and CLOSEW that file. If no user is reading or writing the file, then the grant only right to W and CLOSEW the file is RETURNed to the caller, otherwise the CALL to OpenW is a null operation (notated by RETURN('no')).

```
procedure OpenW(r = {Z of file,OPENW → W CLOSEW GRANT}) =
      if state(Z) = free then RETURN((Z,W CLOSEW GRANT)) †
                        else RETURN('no')

procedure CloseW(r = {Z of file,W CLOSEW GRANT}) = RETURN()

procedure OpenWF(r = {Z of file,OPENWF → R CLOSEWF GRANT}) =
      if state(Z) = (free ∨ wf) then RETURN((Z,R CLOSEWF GRANT))
                        else RETURN('no')

procedure CloseWF(r = {Z of file,R CLOSEWF GRANT}) = RETURN()

procedure Peek(r = {Z of file,PEEK →  OPENWF}) =
                  (OpenWF,CALL),(R,CALL),(CloseWF,CALL);
      if Openwf((Z,OPENWF)) = 'no' then RETURN ('no')
            else { R((Z,R)); CloseWF((Z,R CLOSEWF)); RETURN()}

procedure R(r = {Z of file,R GRANT}) = RETURN((Z,R GRANT))

procedure W(r = {Z of file,W GRANT}) = RETURN((Z,W GRANT))
```

† We discontinue showing the parameter (E,RET) as an argument for RETURN since by the definition of CALL only one right to RET to an environment can be in an environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Verification: Show that problem specifications 1-3 are satisfied by the implementation.

Let **T** = {CALL, RETURN, OpenW, OpenWF, CloseW, CloseWF}.

The first two specifications can be precisely rephrased:

1. One user can perform an uninterrupted sequence of W's to a file F, i.e. for all states S, $\neg$ ((F,W) $\in$ S $\wedge$ (F,R) $\in$ S) $\wedge$ no more than one copy of (F,W) is in S.

2. Multiple users can perform interleaved W-free sequences, i.e. for all states S, $\neg$ ((F,W) $\in$ S $\wedge$ (F,R) $\in$ S) and for all n > 0, there is a state in which n copies of (F,R) appear in user environments.

Several observations will shorten the verification of the specifications:

Any attempt to use a transformation function which fails does not cause alteration of the protection state. Thus only successful uses need be considered.

The properties to be verified involve the existence or nonexistence of rights to files in a protection state. R and W conserve all file rights through CALL and RETURN, thus can be ignored.

A call to Peek produces a new environment containing a subset of the rights in the initial state, calls to Peek need not be considered explicitly.

Thus **T** is effectively reduced to {OpenW, OpenWF, CloseW, CloseWF}.

Property 6.5: For all states S *in* closure(S_0), $\neg$ ((F,R) $\in$ S $\wedge$ (F,W) $\in$ S).

Proof: By procedure definition the grant only rights (F,R) and (F,CLOSEWF) enter and leave a state in the same transformation as do (F,W) and (F,CLOSEW).

Neither (F,R) nor (F,W) is in the initial state S_0 and no transformation function can be employed to introduce both rights at once.

a. If there exists S_2 such that (F,R) $\in$ S_2 $\wedge$ (F,W) $\in$ S_2 then there exists a minimal derivation of S_2 such that $S_0 \Rightarrow S_1 \Rightarrow S_2$ where S_1 is that state for which $S_1 \Rightarrow S_2$ is the longest derivation such that every state in that derivation includes either (F,R) or (F,W). S_1 is not identical to the initial state since neither (F,R), nor (F,W) is in S_0. S_1 is not identical to S_2 since no transformation function can be employed to introduce both rights at once.

b. Assume (F,W) $\in$ S_1. The transformation generating S_1 was OpenW which set state(F) = w. The following transformation (ignoring R, W, and Peek)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cannot be OpenW or OpenWF (for both fail since state(F) = w) nor CloseWF (for if (F,CLOSEWF) appears in an environment in S_1 then (F,R) is also in S_1 violating assumption a. Thus (F,W) must not be in S_1.

c. Assume (F,R CLOSEWF) $\in S_1$. The transformation path by which $S_0 \Rightarrow S_1$ ends with OpenWF which sets state(F) = wf. The next transformation cannot be OpenW nor CloseW (for $\neg$ ((F,CLOSEW) $\in S_1$)) unless assumption c is violated.) Any number of OpenWF transformations increment the number of rights of the form (F,R) in S_1 (refer to the definition of OpenWF and CloseWF) causing state(F) = wf to remain unchanged. OpenW succeeds only if state(F) = free <=> no copy of (F,R) is in any user environment in the current state.

Neither (F,R) nor (F,W) enters the state first in the derivation of S_2, thus for all states S in closure(S_0), $\neg$ ((F,R) $\in$ S $\wedge$ (F,W) $\in$ S).

⊠

Property 6.6: For all states S $\in$ closure(S_0), no more than one copy of (F,W) is in any user environment at one time.

Proof: (F,W) enters a state as a grant only right. No procedure definition alters or amplifies it to be (F,W COPY). By the proof of Property 6.5, assumption b, if a first copy of (F,W) is in a state, no transformation will cause a second copy to be introduced.

⊠

Property 6.7: For all n > 0 there exists state S containing n copies of (F,R).

Proof: Let $S_0 \Rightarrow S_1$ by OpenWF((F,OPENWF)) so that state(F) becomes wf. OpenWF can be successively performed any number of times adding a new copy of (F,R) to the derived state. The protection state derived from a transformation path consisting of n successive executions of OpenWF((F,OPENWF)) will contain n copies of (F,R).

⊠

Specification 1 is proven by Properties 6.5 and 6.6.

Specification 2 is proven by Properties 6.5 and 6.7.

By definition of CALL and the procedure Peek, a user who is initialized to exercise (F,Peek) but not (F,OPENWF) can only perform W-free sequences of 1 R in duration. Thus specification 3 is satisfied.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 6.6: Memoryless Execution

In contrast to the file examples which dealt with properties limiting the access history of instances of one object type, this example will describe another application of closure: to verify properties that characterize all execution by a process during a given time interval. Closure is used to verify that information cannot be 'remembered' across multiple invocations of a procedure for some given procedure definitions and protection state initializations. We define a procedure to be **memoryless** if during one invocation of the procedure, information cannot be encoded within any object, other than a parameter object, so that the information can be referenced during a second invocation of the same procedure.

Problem Specification: Demonstrate that under certain assumptions it can be determined that a procedure is memoryless. †

Method: We restrict our investigation to the actions of a single process from the time it CALLs a procedure (say P), a RETURN is made from that CALL to procedure P and a second CALL is made to P. Because a process which cannot record any information over an interval of time can not maintain intermediate results, we specifically exclude the rights to parameter objects which the caller passed to P when determining if P has memory. We assume that the caller can reverse or destroy the effect of the use of any parameter right during the execution of P, before a second CALL is made to P. (For example, the caller may pass a block of memory for P to write on and erase the

† Selection of this example is motivated by statements in the literature that there exists no way to guarantee that a set of procedures is memoryless without certifying the code executed by those procedures. (Graham & Denning [GD71])

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

contents of the block upon return from P.) This assumption implies that no process can monitor or communicate with P using the parameter objects for the caller could not reverse such effects.

We make another crucial assumption: A process is autonomous and is not monitored. No confederate process records the rate of execution or the order of actions taken by P. The computation resulting from interrupts and multiplexing (for example accounting) must not record information P could obtain on a second invocation. We believe that for many cases where memoryless execution is desirable (for example, providing for execution of user written proprietary software) that assuming the absence of monitoring is not unreasonable.

For simplicity we assume that though CALL and RETURN, as defined in Chapter IV, transfer rights between environments, no procedure CALL requires access amplification. No additional right transfer operations are defined. We consider only hardware defined objects (for example, primary and secondary memory, devices) and procedure objects. †

We define a (read only) predicate R(r) ∈ {TRUE, FALSE} such that for any right r, R(r) ≡ TRUE if and only if right r cannot be used to alter any object. For all rights r to initially defined hardware implemented objects, we assume that R(r) is known. For example, R((X,WRITE)) ≡ FALSE where X names a disk device. For all procedures P_i, let B_i label an equivalence:

$$B_i: \quad R((P_i,CALL)) \equiv \bigwedge_{r \in d:P_i} R(r)$$

i.e. the right to call a procedure is read-only if all declared rights (those

† Both restrictions could be removed, but that would unproductively complicate the example.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

derived from the environment of declaration and not as parameters) for that procedure definition are also read-only. We assume that if from B_1, ..., B_n it can be deduced that $R((P_i,CALL)) \equiv R((P_i,CALL))$, Then $R((P_i,CALL)) \equiv$ TRUE. (Intuitively, this means that if it can be deduced that the only way a procedure has the means to record information is by a recursive call, then the right to call that procedure is read-only.)

A procedure P_i begins execution with parameter rights and the declared rights described in the definition of that procedure. If $R((P_i,CALL)) \equiv$ TRUE then no declared right can be used by P_i to record information. (P_i cannot alter any object without exercising a parameter right. Furthermore P_i cannot CALL any procedure that can record information on behalf of P_i in objects other than those P_i provides through parameters.) By definition if $R((P_i,CALL)) \equiv$ TRUE, then P_i is memoryless since all information encoded as a result of calling P_i is in parameter objects that the caller provided to P_i, and the caller can prevent retrieval of that information for use during a second invocation of P_i.

For the reader who has not considered this problem before, we point out that a memoryless procedure is very restricted. It is unable to invoke many of the functions a modern operating system provides. For example, the procedure to 'lookup' a file might register with that file the most recent date of lookup, thus recording data for P_i and innocently providing information storage.

To illustrate memoryless procedures we define procedure objects P and Q. The only other type of object defined is the segment with accesses READ and WRITE.

Define:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

procedure P((S **of seg**,READ WRITE)) = (S_1,READ),(CreateSeg,CALL),(Q,CALL);
begin . . . end

procedure Q((S **of seg**,READ WRITE),(T **of seg**,READ WRITE)) = (S_2,READ),(P,CALL);
begin . . . end

S_1 and S_2 are presumably specific read-only segment objects available to the procedures P and Q, but not to callers. Define R((S_1,READ)) ≡ R((S_2,READ)) ≡ TRUE.

Let the procedure called CreateSeg return READ and WRITE access to a newly created segment to its caller. We choose to weaken the strict definition of *memory* and define R((CreateSeg,CALL)) ≡ TRUE. To do so we must assume that no information that the segment controller may record during calls made to CreateSeg is available to P or Q or any process monitoring the execution of either. Further, we assume that when created, a segment is made accessible only to the environment of its creator and that the segment is destroyed when no rights to access it exist. In this case a created scratch segment is either returned to the caller of P via a parameter right or is destroyed at the return of P to the original caller because no right to the segment exists in any environment.

By definition,

R((P,CALL)) ≡ R((S_1,READ)) ∧ R((CreateSeg,CALL)) ∧ R((Q,CALL)). Thus

(1) R((P,CALL)) ≡ R((Q,CALL)).

Because R((Q,CALL)) ≡ R((S_2,READ)) ∧ R((P,CALL)), then

(2) R((Q,CALL)) ≡ R((P,CALL)).

By (1), (2) and substitutivity of equivalence R((P,CALL)) ≡ R((P,CALL)).

Thus R((P,CALL) ≡ TRUE. Therefore procedure P is memoryless as is procedure Q.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Computing the read-only predicate to a procedure is a shortened way of computing the closure of a protection state consisting of one environment that contains the declared rights of that procedure.

⊠

One application of this example is in the case of proprietary software. P and Q could be proprietary procedures whose code the authors refuse to make known to callers. Without compromising the content of the procedure bodies, a caller to whom the declared rights for the procedures are known can determine that CALLs to P and Q cannot result in the recording of caller provided information beyond the duration of one call.

Summary

The isolation of the environment from the code which directs execution and from the data unrelated to environment change provides sufficient structure to state properties that describe boundary conditions on the execution that can occur after a specific protection state is initialized. When objects directly corresponding to a physical component and software defined objects are homogeneously protected and accesses are tailored to each type of object, such properties limiting execution can be phrased at any level of abstraction, rather than merely on the level of reading and writing the ubiquitous bit.

The properties of interest are usually concerned with only a small part of the information in a protection state so that only partial states need be considered in proving those properties. Furthermore, in some cases, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

properties of interest make it possible to apply simplifying assumptions in order to reduce the indeterminate number of states in a closure to a small finite number of equivalent classes. Once the simplifying assumptions are determined, the effort to obtain a finite closure seems mechanical.

This chapter provided two kinds of applications of closure. One permitted verification of access histories for instances of a specific object type. The second application illustrated verification of a property that characterized the execution of procedures over an multiple CALLs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A PROTECTION FACILITY

Chapter VII

The unifying theme of the last half of this thesis is an exploitation of the structure that the protection model provides. This chapter investigates the ramifications of providing protection as a centralized operating system facility, uniformly available to all users. Such a facility is used to protect dynamically created, program defined objects as well as objects with physical realizations. We develop the description of this facility in two parts. The first half of the chapter deals with dynamic type creation, i.e. with definition of the operations necessary to create and protect new classes of objects with their associated accesses. The latter half considers the results of representing objects as the set of rights to component objects. Finally an example is given.

Implications of Dynamic Type Creation

As defined in Chapter I, we consider objects as partitioned into equivalence classes, each called a type. All objects of the same type have the same set of accesses applicable to them. In this section we are concerned with the creation of types and therefore with the specification of the accesses applicable to instances of a type. Every protection system provides controlled access to one or more types, i.e. classes of resources. The majority of protection systems adopt one of the following attitudes regarding what can be protected:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. A fixed number of resource classes and accesses are predefined. Each class contains a fixed number of resources (objects). (For example, only the class of memory blocks are protected from reading and/or writing and there exist k blocks of memory.)

2. Though the number of classes, and hence accesses, is fixed, objects may be dynamically created to expand the size of the classes. (For example, in a file system though only one class of objects is protected (i.e. files), new individual file objects can be created dynamically.)

A third alternative and the one we discuss here permits users to dynamically request that a new resource class is to be protected. We refer to such a request as **dynamic type creation**. It is the basis for providing protection as a user service applicable to user defined objects as well as the objects managed by the initial system builders. In particular, types need not correspond only to physical resource classes such as devices or physical memory. Neither must types correspond only to virtual resources created and maintained by an 'operating system'. Any user can declare a new type, specify the accesses to this type, and then rely on the protection system to enforce controlled access to instances of this new type.

We feel another implication of type creation is that it encourages program designers to modularize their programmed applications by using a subsystem structure and to present services provided by one module to other modules in terms of manipulable objects. A **subsystem** defines one or more type(s) of objects and the accesses applicable to instances of those type(s).

The subsystem, composed of a set of procedures which implement the algorithm associated with each of these accesses, maintains the physical existence of all instances of types it defines. In effect each subsystem realizes an abstraction from the machine on which it runs by providing its users a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

machine augmented by the definition of a new class of abstract resource. To the outside world the subsystem is known only in terms of the type(s) defined and the procedures which can be invoked to perform the accesses defined for the subsystem supported type(s). The interface between the subsystem and its users is controlled by the protection system.

Types as Objects

Before specifying a set of operations to support dynamic type creation, we present one way of defining types more precisely within our model.

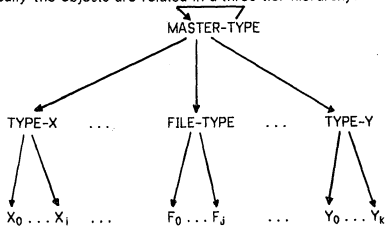
Types distinguish classes of objects. For each type we define a corresponding type-object used to represent (and control) the class of objects of that type. Access to a type-object in some way affects the whole class. One such access is CREATE which increases the size of the class. For example, let FILE-TYPE be the name of the 'permanent file type-object'. A process can create a permanent file only by exercising the right (FILE-TYPE,CREATE). Without that right, a process cannot cause new permanent files to be established. Now we are able to limit the ability to create files, not just control access to individual existing files.

Just as all objects of a specific type form a class of resources, the set of type-objects form a class. We define a MASTER-TYPE object which stands in relation to the type-objects just as each type-object stands in relation to instances of that type. The MASTER-TYPE object is used to control the proliferation of type-objects by restricting use of the right to CREATE instances of the MASTER-TYPE object. Demands made on the protection

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

facilities can be restricted by controlling the creation of new types (and type-objects) as well as by restricting the creation of instances of individual types.

Graphically the objects are related in a three tier hierarchy:



Each directed arc emanates from a type-object and terminates in an instance of that type. (The type of an object is the name of the object above it in the hierarchy.) So that all objects have a type we define the MASTER-TYPE to have the peculiar attribute of being an instance of itself.

Maintenance of Type-objects

The definition of subsystem states that for each type defined, the subsystem is responsible for

1. defining the accesses applicable to instances of that type,
2. implementing procedures to perform the accesses defined,
3. maintaining any representation of instances of that type, after they are created.

Note that in general the protection system is responsible only for creation and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

maintenance of rights to a new object. The subsystem that defined the type of an object is responsible for representation of that object. (For example, though message buffers are objects, the space for buffers is allocated by the communication subsystem, not the protection system.)

It is convenient to be able to describe protection as though it were provided by a subsystem supporting the existence of two types of objects: instances of MASTER-TYPE (i.e. the set of type-objects) and ENV-TYPE (i.e. the set of all environment objects). The protection subsystem defines a set of accesses applicable to type-objects. These accesses, each of which is described below, include CREATE, DEFINE, ERASE, and PERMITAMP.

1. CREATE(TYPE-X,E) is the right transfer primitive defined in Chapter III. Invocation of CREATE causes creation of a new right from an object name which has never before been used and a list of all accesses applicable to objects of type TYPE-X. The caller of CREATE, executing in environment E, is given all possible rights to the object for which a new global name has been generated. The protection system must know the list of accesses applicable to each type. Thus each type-object has as its representation that list. The MASTER-TYPE is represented by a list of the accesses we are now defining.

2. DEFINE(TYPE-X,A) adds access name A, to the set of accesses encoded in the representation of the object TYPE-X. DEFINE is required so that a type creator can specify those accesses which distinguish the type. (If A is already an access applicable to objects of TYPE-X, then no effective change in the TYPE-X representation occurs.)

3. ERASE(TYPE-X) tells the protection subsystem to forget that TYPE-X

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

existed; in particular it causes erasure of the representation of the object TYPE-X.

4. A fourth access is required if (and only if) access amplification of parameters is permitted. This access, PERMITAMP, is used to control the declaration of procedures whose parameter templates request access amplification at procedure invocation. To illustrate the breach of protection that would otherwise result: Assume that in any environment, it is possible to define procedure objects which specify access amplification of formal parameters. Then a process which has the restricted right (Z **of ty**,A) can define a new procedure:

procedure Q(r={X **of ty**, A → A'}) = RETURN((X,A'))

Executing Q((Z,A)) will allow the process to acquire (Z,A') where A' is any access applicable to objects of type **ty**. Consequently the ability to request access amplification must be regulated.

To declare a procedure which specifies access amplification for a formal parameter of type **ty**, the right (TY,PERMITAMP) to amplify access to all members of that type must be in the declaration environment. PERMITAMP is an access applicable only to type-objects and used by the procedure creating operation to decide if access amplification specified in a procedure definition is permitted.

We now consider the specification of the protection subsystem support of ENV-TYPE. The set of accesses applicable to environments differ from system to system, but will be those required by the right transfer and environment crossing operations. The associated access algorithms are implemented within these operations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As an example, let TO be an access to an environment object used to control movement of a right into an environment. Each right transfer operation and each environment crossing operation requires (E,TO) as a parameter right before they honor a caller's request to move a right into environment E. Note that if a right transfer operation is itself actually performed in an environment E which does not contain (E,TO), even this powerful right movement operation is restricted by the protection system so that it cannot divert rights to its own environment, E, for misuse.

Implementations of environments can vary widely as discussed in Chapter II. In case an environment is declared to be an object, the protection subsystem provides a procedure Createenv that manufactures environment objects. Only the environment in which Createenv is actually performed contains the right (ENV-TYPE,CREATE) which can be used to generate a right to a new environment for which the procedure Createenv provides a representation. †

Constituent Rights Revisited

We now follow an avenue of investigation opened by dynamic type creation. Since object types can be freely created, it is natural to relate objects to one another. One particularly useful relation is that of composition, i.e. one object is composed of other objects. Composition implies that first, the

† For the reader to maintain perspective we want to point out that the protection facility is a centralized authority not under the control of its users. This means that even if a user could maintain the protection data base (e.g., rights to use a certain type of object) in a more compact encoding than the protection facility does, the user cannot employ the efficient encoding and still use the protection facility.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

structure of an object is partially determined by the structure of component objects and second, the operations that can be defined for an object are limited to those that can be implemented using the operations provided by subsystems that support the component objects. To perform an operation on an object composed of other objects requires rights to manipulate those components.

To maintain the relation between an object and its components we associate with each object a set of constituent rights. † We use rights, rather than objects, because it is useful to be able to restrict access to component objects as will be apparent in a later example. In this section we extend the protection subsystem developed earlier to maintain the constituent rights of objects. Associated with each object is an environment used to contain constituent rights. Such environments are storage compartments, never to be used as execution environments.

The operation CREATE not only generates a new object name, but now creates and initializes an empty environment for that object. The access field of rights to all objects is extended by the new accesses STORE and LOAD. STORE enables a process to transfer a right from the current execution environment to the constituent rights environment of an object. LOAD allows the reverse transfer. Procedures implementing the algorithms for LOAD and STORE accesses are additions to the set of operations that define the right transfer policy of a system for they transfer rights between constituent rights environments and execution environments. (Procedures implementing LOAD and STORE can be constructed using the right transfer primitives COPY and

† Some objects, at a minimum the hardware implemented objects, are atomic in that they are not composed of other objects.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

GRANT.)

A process that has in its execution environment a right to an object regards that object as a whole to which accesses are applicable to obtain information or alter the object. As pointed out above, in order to define procedures to implement access algorithms, rights to manipulate the components of parameter objects are required. By the need-to-know principle rights that are call dependent should be in the execution environment of a procedure only for the duration of that particular call. For that reason we employ access amplification as described in Chapter IV.

A subsystem writer implementing the access algorithms for a certain type of object has PERMITAMP access to the type-object. Using that right he declares procedures that require access amplification of parameters (of that type) in order to gain LOAD and/or STORE access to the parameter objects at CALL time. With access to the constituent rights for the parameter object, the CALLed procedure can transfer them into the execution environment to use in performing the access algorithm.

The environment binding policy defined as in the CALL mechanism in Chapter IV is sufficient to construct CALLed environments containing amplified parameter rights. (Alternatively a more powerful CALL operation can be defined that allows only a subset of the constituent rights of a parameter object to be transferred to the execution environment of a CALLed procedure. To see how such a mechanism is designed, refer back to the discussion of uniform and nonuniform object amplification in Chapter IV.)

Our investigation has led to a new kind of environment construction that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

we call **amplification**. It permits construction of an environment containing rights that are dependent upon the procedure being called and the call parameters, but unavailable to the caller for any use. This contrasts with the usual environment construction where the environment of a CALLed procedure includes **all** those rights that the caller does not (or cannot) pass as parameters and that might be needed for any CALL. This implies that the environment will contain rights unneeded the majority of the time. An alternative method of environment construction, where amplification is not provided, is to have a holding environment which when CALLed will RETURN just those rights that amplification immediately provides. This merely puts off by one level of indirection, the environment that holds all CALL dependent rights ever needed. It is clear that when objects are associated with their constituent rights, amplification serves the need-to-know principle far better than conventional environment construction methods.

Example 7.1: Constituent Rights and Amplification

To illustrate the use of amplification, we define a mailbox communication subsystem. We assume that a synchronization subsystem exists which maintains objects of type **semaphore** with accesses P, V as well as a Createsem operation. † The protection subsystem described above defines objects of type **environment** with accesses TO and FROM as well as a Createenv operation.

The communication subsystem defines a new type of object called **mailbox**.

† It is assumed that the reader is familiar with the definition of synchronizing semaphores and the P and V operations defined by Dijkstra [D65].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

All messages are rights and pass from the execution environment of the sender into the mailbox and subsequently from the mailbox to the execution environment of the receiver. (There is no restriction on the type of object named in a message right. It could as well be a memory block as a process, environment or user defined type.)

A mailbox is composed of

1. several semaphores for synchronizing the use of the mailbox, and
2. an environment to contain messages sent, but not yet received.

A mailbox has at least two accesses, SEND and RECEIVE. For each there is a procedure of the same name to implement the associated algorithm.

For this example we consider only SEND. To invoke the procedure Send, a caller specifies a mailbox to which the caller has SEND access and a right that is to be the message. The procedure declaration for Send is

```
procedure Send(r = {M of mailbox,SEND → LOAD},msg) =
        begin . . . LOAD((E of environment,TO),M) ;
                    LOAD((S of semaphore,V),M) . . . end
```

By access amplification the Send operation gains LOAD access to a particular mailbox named by a parameter and can transfer some or all of the constituent rights of the mailbox M into the execution environment. In the procedure definition above Send acquires at least two constituent rights of the mailbox parameter--one to a semaphore and the other to the environment in which to put the message. (We assume that some naming convention permits selection of the desired rights from the constituent rights for the actual parameter mailbox object. Here we use content addressing.) Send can then perform a V operation on the synchronizing semaphore of the mailbox and use the TO access to move the parameter right with formal name 'msg' into the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mailbox environment to await a receiver. (Only constituent rights for the one parameter mailbox are available during each invocation of Send. In addition the caller had no access whatsoever to the constituent rights of a mailbox--not even the right to pass them as parameters to procedures.) Note also that if a new mailbox is created, it can be used as a parameter for a call to procedure Send without any special preparation.

Summary

This chapter has used the model as a starting point for briefly developing a new concept: a centralized protection facility. Such a facility provides two separable services--dynamic type creation and constituent rights support. The ramifications of both are suggested, but not developed in detail.

Questions of implementation, efficiency, cost and usability of a system that implements dynamic type creation and amplification of environments are better addressed when data is available from an actual system. Developed at Carnegie-Mellon University, the software operating system kernel HYDRA provides a protection facility that includes both concepts (Wulf et al [WU73]). Since actual data should soon be available for that system, no analysis of implementation is performed here.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SUITABILITY MEASURE

Chapter VIII

One significant problem of system design is how to functionally compare protection policies offered in different operating systems on the basis of the facilities each provides. Such comparisons are difficult to make when each policy is described in the vernacular of its implementation. The model provides enough structure to permit policies to be described in terms of environment maintenance and usage, so that the policies are comparable along some axes. In this chapter we adopt as a touchstone the need-to-know principle: at any time a process has in its execution environment only those rights required to perform the current task. With this principle providing a yardstick, we compare classes of operating system protection policies.

To perform the comparison we postulate **demands** defined as constraints on the content of environments, and constraints on process crossing between environments. Given a demand described along these two dimensions, we wish to compare how different systems satisfy or approximate satisfying such a demand.

We assume that an individual demand fixes a set of environments and their content as well as a set of processes and specifications on how the processes cross between environments within the set. It is to be determined how accurately a protection system can provide the configuration defined by that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

demand without interposing additional software (especially constructed to service that demand) and without enlarging the configuration itself (for example, adding processes or environments or tampering with the specifications.)

To discuss how protection systems can be compared with respect to meeting a particular demand, a definition of approximate satisfaction of a demand is required.

Measuring Accuracy

Protection systems vary markedly in the accuracy with which they permit an environment to be tailored to include precisely the rights needed to perform a computation. For example: Assume that a protection policy requires that environments be nested by containment. Assume further that the tasks to be performed within a set of environments (one task per environment) are **not** ordered so that each task requires a subset of the rights of the next task in the ordering. The more inclusive environments will contain rights not needed during execution within those environments, but forced to be there to satisfy the protection system policy.

To analyze how accurately an environment suits the execution performed within it, we assume that each execution environment is constructed to be the site at which a single task is performed. It is a characteristic of programmed systems that performance of a task is of a positive finite duration. It is possible and usual for the contents of the execution environment to change during a performance of the task. However there are only a finite number of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

rights which exist in the environment for the finite duration of one task performance. The following definition specifies one measure of how accurately an execution environment suits its use:

Defn: The **accuracy measure** of an execution environment is the ratio of the number of rights exercised in the environment compared to the total number of rights which could be exercised within the environment during the performance of one task.

For the purpose of calculating accuracy a right (X,A B) may be regarded as one right or two rights: (X,A), (X,B), as long as the distinction is consistently made. If repeated instances of a task (or different tasks) are performed in a single environment, then the accuracy measure for that environment is the average of the accuracy measures over a representative sample of individual task performances.

The extreme values of the measure are

1 -- where an environment contains precisely the rights required during the one (each) performance of its associated task. Such an environment provides 'ideal' protection. However such a match of execution and environment generally requires a priori knowledge. Where a right will be conditionally exercised in the performance of a task, the right cannot be entered into the environment of execution unless the choice to use it can be predicted.

0 -- a lower bound on accuracy which is approached as the number of unused rights in an environment increases.

The accuracy measure is a way to more precisely express the need-to-know principle. The closer to 1 an accuracy measure of an environment is, the more precisely the environment provides just those rights needed for execution in that environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The accuracy measure for an execution environment is dependent upon the execution which takes place in the environment which in turn is dependent upon the decomposition of a problem solution to specify the size and complexity of the task to be performed in that environment. The accuracy measure indicates whether an environment is 'overstocked' relative to its usage, but does not reflect 'understocking' which occurs if an unnecessarily fine decomposition generates multiple tasks each to be performed in almost identical (in content) environments.

Using the accuracy measure to quantify how closely an environment implementation meets the specification of that environment, we now develop a measure on how accurately a system implementation satisfies a demand that is phrased in terms of constraints both on environment content and process usage of multiple environments.

Let a demand that involves k specified environments have an implementation within a particular system, that involves k or less implemented environments. An accuracy measure can be computed for the implementation of each of the k specified environments. We reason that if more than one specification environment is realized by a single implemented environment, then the accuracy measure calculated for each of the specified environments will be lower than if each specification environment were implemented separately. This is because the implementation environment contains the union of the rights in several specified environments, i.e. more rights than required by any one of the individual specified environments that it implements. †

† We assume that in general the specified environments are distinguishable because they differ in content.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A demand can also specify between which environments a process can cross. When an implementation cannot meet such a specification, we assume that two specified environments between which a process is to cross will be implemented as one environment so that the crossing is rendered unnecessary. (In finding an approximate solution to a demand specification, the generation of new processes not required by the demand is prohibited.) Thus we reason that the inability of a protection system to meet the specification with respect to process crossing between environments results in fewer, but larger implemented environments. As discussed above, implementing two specified environments as one implemented environment lowers the accuracy measure for those specified environments so that an implementation's approximate solution to a crossing specification is reflected in a lowered average of the accuracy measures.

Defn: The **suitability factor** of a system with respect to a demand is the average of the accuracy measures of all environments required by the demand specification.

If the protection afforded within some environments were deemed more important than in others, then the definition of suitability could be altered to be a weighted average of the accuracy measures of the environments involved.

The suitability scale with respect to each demand ranges between 0 and 1 where a suitability factor of 1 means that the protection system implementation satisfies the demand exactly. Zero is the lower bound for mismatch between an implementation and a demand specification. Given any demand, protection systems are partially ordered along the suitability scale. (No distinction will be made between two systems which can provide an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implementation to satisfy a demand exactly.) Two partial orderings induced by two demands are not necessarily correlated unless the demands are related: for example, one demand is a special case of the other. We assume that intelligent (if not optimal) use of protection facilities is made in devising the implementation which responds to a demand in order to calculate the suitability factor for that system with respect to the given demand.

A detailed example of the use of the suitability factor will be presented to demonstrate that, though approximate, the suitability scale is useful for comparing systems.

Example 8.1: Protection System Comparison

This example illustrates that a group of representative systems can be partially ordered with respect to how well they satisfy a demand. We will not be interested in calculating numerical suitability factors. Instead we use the definition of factors and the definition of the model developed in past chapters to argue that the generic systems can be ordered with respect to the following demand:

CALL DEMAND: The execution environment used for each performance of a task contains only those rights needed during the performance of that task.

We assume that it is clear when a new task is being started, for example, the program to accomplish each different task is coded in a separate procedure and procedure invocation is a well-defined event.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Several generic systems will be sketched, then placed along the suitability scale with respect to the CALL demand. The notation S(Z) is used to distinguish the suitability factor for the system called Z.

Consider the following systems:

USER system:

A USER system defines each user i as a set of processes able to execute in either of two environments E_0 and E_i where the content of all user environments of E_i is contained in the supervisor environment E_0, and for all users i,j $i \neq j$ $(E_i \cap E_j)$ is null. S(USER) < 1 because a USER system permits construction of only two nested environments for each user. A user process performs all tasks in one of two environments. Where more than two tasks are performed, at least one specified environment must have a sensitivity measure of less than 1 so that the suitability factor of the USER system is less than 1.

OS/360 MVT memory protection [IBM] provides an example of a USER system. Using the object site coding and grouping of protection information, as discussed in Chapter II, each memory block is associated with one of sixteen storage key values. A user process executing in an environment has a right to access a memory block if that block has a key which matches that distinguishing the process' environment. Two keys 'match' if they are identical or if the environment key value is zero. Thus key 0 denotes the supervisor environment which has all access to all memory objects. The other keys denote fifteen isolated user environments.

PROCESS System:

In a PROCESS system each process executes in a set of private environments

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

linearly nested by containment. A PROCESS system can only approximate the CALL demand because each process has only a fixed number of environments in which it can perform the tasks required of it and these environments cannot be tailored for each task invocation.

S(USER) < S(PROCESS) because

1. a single process uses each environment in the PROCESS system whereas multiple processes use the same environments in the USER system. This implies that if there exists a right used by one process and not by another, both rights are in one USER environment, but a PROCESS system allows each process to have only those rights it needs. Thus the accuracy measures computed for the PROCESS system can exceed those of a USER system.

2. the PROCESS system provides a larger number of environments for each process than the USER system, which is fixed at depth two. In a PROCESS system the tasks a process must perform can be divided among a larger number of environments than two which implies that the environments can be made to contain less rights in general, than in the USER system.

MULTICS is one example of a PROCESS system where each process executes in a limited number of environments (or 'rings' in the MULTICS terminology) [072]. The environments are related by containment, but in a more complex way than merely linear nesting.

PROCESS-PROCEDURE System:

In a PROCESS-PROCEDURE system each task is implemented as a procedure. For each process-procedure-pair there is a specially tailored environment which contains all rights the process could need during any call to that procedure. It is possible to construct two procedures such that given process p, two environments E, E' are created in which p performs these procedures, and there exist rights r,r' such that $r \in E$ and $r' \in E'$ and neither r nor r' is in $(E \cap E')$. Thus a PROCESS-PROCEDURE system permits construction of environments which are not restricted to be ordered by containment making a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

better approximation to satisfying the CALL demand than the PROCESS system. Therefore S(PROCESS) < S(PROCESS-PROCEDURE) < 1

CAL-TSS [L69a] is an example of a PROCESS-PROCEDURE system.

PROCESS-PROCEDURE-CALL System:

In this last generic system, environments are dynamically created each time a process calls a procedure to perform a task. An environment is formed containing the declared rights required by the procedure and the caller specified parameter rights.

S(PROCESS-PROCEDURE) < S(PROCESS-PROCEDURE-CALL) because environments constructed in the PROCESS-PROCEDURE case are reused for multiple calls and must therefore contain all rights needed during any call, whereas the environments of the PROCESS-PROCEDURE-CALL system are created for each call and potentially contain less rights--i.e. the environment is more accurately constructed for an individual call. Thus the suitability factor for the PROCESS-PROCEDURE-CALL system exceeds that for the PROCESS-PROCEDURE system. As pointed out earlier, without a priori knowledge of exactly what declared rights are needed for a particular call, the execution environment will still contain rights which are not needed in certain instances so that the suitability factor of the PROCESS-PROCEDURE-CALL system is still less than 1.

HYDRA [WU73] is one example of a PROCESS-PROCEDURE-CALL system augmented by parameter amplification.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Summarizing, the CALL Demand causes the four generic systems to be strictly ordered along the suitability scale of 0 to 1 as follows:

$$0 < S(\text{USER}) < S(\text{PROCESS}) < S(\text{PROCESS-PROCEDURE}) < S(\text{PROCESS-PROCEDURE-CALL}) < 1$$

Consequently specific instances of each generic class can also be ordered.

$$0 < S(\text{OS/MVT}) < S(\text{MULTICS}) < S(\text{CAL-TSS}) < S(\text{HYDRA}) < 1.$$

The use of the suitability scale to partially order systems by their ability to satisfy a specific demand has now been illustrated.

⊠

Summary

This chapter demonstrates how the facilities provided by different protection systems can be compared. The ability to compare protection systems at all, depends upon having one terminology with which to describe diverse systems. Our model provides such a set of terms and concepts (including environments, rights and the primitives).

In this chapter we select the need-to-know principle (one possible touchstone) as the basis for comparison. This principle can be stated as: a process performing a task in some execution environment should have access to only the objects required in the performance of that task. We define an accuracy measure for an environment to be the ratio of the number of rights exercised in an environment to the total number of rights that could be executed within the environment during the performance of that one task. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

accuracy measure of an environment expresses how well the need-to-know principle is supported for a specific environment.

To compare two protection systems, we first define the systems in the terminology of our model and then define a set of constraints on the content of the environments and on process crossing between these environments. Alter implementing these specified environments in each system, we calculate the average of the accuracy measures for the environments defined. This is called the suitability factor and can be used to determine which system best implements the set of constrained environment and thus best serves the need-to-know principle (for that one set of constraints).

Finally we define one specific set of constraints and order four generic systems that provide differing protection facilities and necessarily differ in the way each attempts to meet or approximately meet the constraints.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CONCLUSION

Chapter IX

This final chapter is divided into two sections; the first summarizes the results of our investigation while the second suggests some directions for further research.

Summary

Chapter I motivated the basic principles and defined the structure of our model of how access to objects in a programmed system is controlled. The variability in the model is restricted by three rules that are to be bound in order to describe any individual system:

-Enforcement Rule

-Right Transfer Rule

-Environment Binding Rule.

Chapter II investigates alternative encodings of execution environments required to enforce protection and satisfy the first rule. Chapters III and IV present a small set of primitive operations for altering environment values. These tools are used to bind the remaining two policy rules and are sufficient to usefully describe diverse protection policies as verified in Chapter V.

One contribution of this model is that it explicitly includes the portion of the protection policy related to dynamic environment boundary crossing by processes, without restricting the definition of how and when such crossing can occur. (For example, the model accommodates coroutine execution control

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and nested procedure call regimes equally well.) We believe another value of the model is that it clearly isolates the issues involved in protection. It isolates in primitives the mechanisms of performing enforcement of protection and manipulating data recording protection information, leaving policy to vary naturally in the face of different applications. In addition the model separates interpretation of objects and the ensuing complication of representation and resource allocation, from the contol of access to objects.

In Chapters VI through VIII we show that the structure of the model is useful for other than description of protection systems. Chapter VI demonstrates that when environments and the operations that alter them are well defined, properties characterizing execution within an initialized set of environments can be proven independent of the programs to execute in the environments. This tool is very powerful for it allows proof of properties for classes of programs, rather than for individual programs. Chapter VII discusses two notions (dynamic type creation and constituent rights) introduced if protection is offered as a service to a user population that can declare new objects to be protected. One interesting result is the emergence of a new kind of environment construction mechanism called amplification.

Chapter VIII addresses a serious current question in the area of protection: can two protection policies implemented in incomparable ways be compared? To answer affirmatively, we first define a measure of the accuracy with which an execution environment serves the need-to-know principle. That principle states that an execution environment should contain only those rights needed for the task currently being performed in that environment. Then to compare how suitably two protection systems provide support for a specific

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

application, described as a configuration of environments and changes in the value of those environments, we average the measures of accuracy with which each system actually provides the required environments. As an example we choose one set of environment constraints and order a set of five generic systems by how well each system can implement environments which meet the constraints.

Future Research

The technical problem of providing protection has become increasingly important as the computations performed by users of programmed systems become more interdependent and as society places heavier reliance on the performance of these systems. There remain many problems to be solved and we believe that the model presented in this thesis provides a useful starting point for both framing and addressing some of these problems. For example:

1. Protection costs both time and space. Some efficiency can be gained by implementing part of protection enforcement in hardware. The model separates mechanisms from policies. And it is the mechanisms that are candidates for hardware implementation. In particular the parameter right passing mechanisms used during environment boundary crossing (which are studied at length in Chapter IV) are excellent candidates for inclusion in hardware. In this thesis we introduce a new mechanism called AMPLIFY. Its value is still to be tested.

2. It is most important to prove that protection enforcement cannot be circumvented, that is that the Enforcement Rule holds for a specific protection system. The clear delineation of mechanisms separate from the policies controlling their invocation is a proper starting point for such a proof.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. This investigation was limited (cf. Chapter II) to cases in which environments were directly representable--that is, not dependent on arbitrary system variables. A more general version of the model would record environments as a set of predicates depending upon arbitrary variables. Such predicates must be dynamically evaluated to determine if a right is in an environment. The viability of recording environments in such a way is unknown, though its usefulness is clear. A users privilege to access sensitive information within a large data base may depend upon the value of the information being accessed. For example, a manager may have access to the salary field in an employee record only if the salary recorded is less than a fixed value. Whether such a general model could be devised and usefully applied is an open question.

4. A last problem for further research is that of minimizing the cost of protection enforcement. One way to accomplish this is to provide programming languages with syntax to express user protection needs. We demonstrated how restrictions on the access history of an object could be proven. In one example we proved that a file object with applicable accesses O, R, W, and C has an access history generated by the regular expression (O (R v W)* C)* . Access history restrictions on variable objects can be enforced by compilers, if the history can be expressed by a generator, described for example as one or a set of regular expressions. The compiler could force that objects be accessed in a way that observes the restrictions specified by a user defined access history generator. The analysis required to enforce such access history patterns appears to be very close to the flow analysis done by compilers for optimization. In some cases static, rather than dynamic, checking could be used.

In conclusion it is clear that the study of protection is actually the study

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of environment control in general. And environment control in operating systems and in programming languages are the same. Future work should serve to consolidate the two so that compilers can be used to help solve protection problems in operating systems. This consolidation should lead to a clear understanding of both systems and languages.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NOTATION

Appendix A

Programs and program segments are described thoughout this thesis using a higher level procedural language syntax. This syntax and some of the standard data structure definitions and their interpretations are described below.

Data Structures

To declare the existence of a new class of data structures requires defining the class name and the field names to be used in selecting components of instances of that class of structure. For example,

struct right = obj, acc

defines a class of data structures called 'rights'. Given a right r, obj:r selects one field which is interpreted to hold the unique (and global) name of an object and acc:r selects the other which holds a list of accesses applicable to the object named in the field obj:r. (We assume that there exists a function called 'type' which when applied to an object name will compute the type of that object.)

Where a field in a structure is to contain an instance of another class of data structure, that class name appears in boldface. An environment is defined as a table.

struct env = **right**[1:k]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To select an element of a table formed from linearly ordered homogeneous components, we use bracket notation so that if E is an environment, then E[x] is the x-th right in the environment. In most cases the actual selection mechanism is left unspecified. We use both offset addressing (for example, if x = 4 the E[x] is the fourth right in the environment) and content addressing (for example, E[(P,A)] is the first right in environment E, such that obj:E[(P,A)] = P and acc:E[(P,A)] includes A).

Procedures

Procedures are data structures used to define the code that implements an algorithm as well as to specify the environment in which that code is to be executed. The format of a procedure is defined by

struct procedure = **template** ps[1:n], **env** d, entry

The meaning of each field is that interpretation made by the CALL mechanism defined in Chapter IV. Every procedure is composed of a list of parameter templates, an environment containing declared rights required to execute the body of the procedure and an entry point (which we assume names one right among the declared rights. That declared right permits execution of the code in the body of the procedure.)

A parameter template is also a three field structure.

struct template = ty, reqacc, ampacc

CALL also interprets these fields: The ty field specifies the type of the object to be named in the parameter right. The reqacc (req-uired access) field contains a list of access names that must be in the access field of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parameter. The ampacc (amp-lification access) field lists the access names to be added to the contents of the acc field of the parameter that is placed in the CALLed environment.

A special notation is used for declaring procedure data structures to make them more readable. Parameter template definitions are enclosed in curly brackets with the required access and the amplification access separated by the symbol '→'.

For example, {X_0 **of type-0,**A_0 → A_0'} is a declaration of a parameter template structure s, such that ty:s = type-0, reqacc:s = A_0 and ampacc:s = A_0'. X_0 is the formal name used in the procedure body to designate the object that will be named in the parameter.

An example procedure declaration is

```
procedure P (r0 = {X0 of type-0,A0 → A0'}, ..., rn = (Z,B)) =
            s[1:k];
            begin . . . end
```

A procedure P requires n + 1 parameter rights to n + 1 objects some of which are described by a template, such as r_0. Others are explicitly named such as (Z,B). The declared rights are s[1:k] and the body of the procedure is the compound statment encoded between the **begin** and **end** brackets. At procedure declaration time the body for P is encoded in a memory object and the right to execute from that object is added to the declared rights with entry:P naming that right.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

INDEX OF DEFINITIONS

Appendix B

The definition of each term will be found on the page number associated with it.

access	6
access amplification	45
access matrix	10
accuracy measure	115
AMPLIFY primitive	53
capability	8
CALL mechanism	58
closure	81
constituent rights	47
COPY primitive	32
CREATE primitive	35
DELETE primitive	32
demand	113
derive	33
directly derive	33
dynamic type creation	102
Enforcement Rule	7
environment	7
Environment Binding Rule	10
GRANT primitive	33
immediate closure	50
need-to-know principle	113
nonuniform object amplification	48
object	6
protection	1
protection state	32
protection system	3
READ primitive	36
RETURN mechanism	59
right	7
Right Transfer Rule	9
subsystem	102
suitability factor	117
transformation path	81
type	6
uniform object amplification	48

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

COMPARING PROTECTION FEATURES

Appendix C

The purpose of this appendix is to demonstrate that our model provides a useful basis for comparing features of two very different protection systems. We will compare the protection aspects of two environment boundary crossing mechanisms: the CALL mechanism in HYDRA (Wulf et al [WU73]) and the Extended Calling Sequence (ECS) mechanism of the Virtual Memory system designed by D. L. Parnas and W. R. Price [P73]. This requires translating the protection aspects of the VM system into the terminology of our model.

System Descriptions: We provide a description of each system sufficient to make the desired comparisons, but insufficient to provide an overview of either system. Both mechanisms are used to create a site in which to execute a called program.

HYDRA: The CALL mechanism is as described in Chapter IV of this thesis. † As defined there performance of a CALL requires a procedure object with a structure defined by

struct procedure = **template** t[1:k], **env** d, entry

VM System: To translate the designers' description of this system into the terms of our model we define three types of objects: segments, spaces and transfer vectors. Segments correspond to blocks of memory and have accesses READ and WRITE.

Spaces are environment like objects containing rights to segment type objects only. Transfer vectors correspond closely to procedure objects in the HYDRA system and are defined

struct trvect = **boolean** accept[1:n], **pointer** p, entry

The pointer p names the space holding the declared rights for use by ECS in constructing the 'called space'. Spaces are assumed to be of fixed length

† In actuality the HYDRA CALL mechanism does include additional features not discussed in this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

n and the boolean vector accept is used to determine whether or not a parameter right to be placed in each of the n slots of the 'called space'. All parameter rights of concern are to segments, so we will not require specification of the accesses applicable to spaces and transfer vectors for this discussion.

Both procedures and transfer vectors specify entry points by naming a declared right that permits execution of the code associated with the procedure (or transfer vector) called. †

Since our objective is to show that the two calling mechanisms can be compared, and not to perform an in-depth comparison, we consider only the mainline (non-error) logical paths through mechanisms.

Both CALL and ECS create a new environment and space respectively, based on procedure and transfer vector objects. For the VM system we define a specific transfer vector T:

trvect T = X, (0,1,...,1,0,1,...,0), ep

The ECS mechanism for a call to T reduces to

$ECS_*(T,s_1, \ldots, s_n)$ =
create new space E;
for i ← 1 **until** n **do if** accept:T[i] = 1 **then** $COPY(s_i,E_*,E)$
else COPY(i,X,E)

We assume that the caller is able to specify a null parameter if desired. ECS constructs a new space E, copying segment rights from the callers execution space E_* when the accept vector of the transfer vector T indicates that a parameter is expected. Otherwise ECS copies a right from the template space X into E.

To perform the corresponding operation with CALL, we define procedure P:

procedure P = **template** t[1:j], **right** r[1:n-j], ep

† Much of the elegance of the VM design deals with the management of a window into a space through which only the currently used segments are available. This is not discussed here since segments within a space can be freely moved into the window. Thus the segment management does not pertain to protection.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To correspond to the ECS space of length n, we define procedure P to execute in an environment initially containing n rights and j parameter rights, where j is the number of '1-values' that appear in the boolean vector in T.

The CALL mechanism reduces to

```
CALL*(P,s1, . . ., sj) =
      create new environment E;
      for i ← 1 until j do
            if  type(E*[si])  =  ty:t[i]                    (1)
              ∧ reqacc:t[i]  ⊂  acc:E*[si]                  (2)
              ∧ COPY ∈ acc:E*[si]                           (3)
            then {COPY(si,E*,E);
                  AMPLIFY(E,si,ampacc:t[i])};               (4)
      for i ← 1 until n-j do COPY(i,r:P,E)
```

ECS used with transfer vector T and CALL with procedure P can be used to construct a space and an environment respectively containing the same rights.

We observe

1. Both mechanisms create a new environment (space).

2. Both copy declared rights from a source into the newly created environment or space. In the case of ECS that source is an object potentially accessible from other spaces, therefore
   -it can be altered dynamically
   -it can be the source for multiple transfer vector objects.
HYDRA maintains the declared rights in the procedure object which is not altered, once created. To define multiple procedures with the same declared rights, separate copies of those rights are recorded.

3. The HYDRA CALL mechanism is not restricted to handling a single object type and contrasts with ECS by providing type checking on line (1). This is unnecessary for ECS since all parameters are of type segment. Another contrast is a check (line (2)) that the acc field of a parameter right contains that access list required by the procedure, e.g. reqacc:t[i] for the i-th parameter. The CALL mechanism distinguishes between COPYing and GRANTing rights therefore checks to ascertain that a COPY can be performed on the parameter right (line (3)). Access amplification of parameters is defined on line (4).

We discontinue observing the similarities and differences between the two systems. The objective of this appendix is to demonstrate that protection

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

aspects of two systems can be compared when expressed in the terminology of the model. It was the author's experience that that the protection aspects of the VM system could be easily and usefully expressed in the terminology of the model, and that before doing so a clear basis for comparison of the two mechanisms was not discernible.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

[B61] Burroughs Corporation, The Descriptor - A Definition of the B5000 Information Processing System, Detroit, Mich., Feb. 1961.

[DVH66] Dennis, J. B. and Van Horn, E. C., Programming Semantics for Multiprogrammed Computations, **CACM 9**, 3 (March 1966), 143-155.

[D65] Dijkstra, E. W., Solution of a problem in concurrent programming control. **CACM 8**, 9 (Sept 1965) 569.

[D68] Dijkstra, E. W., The Structure of the T.H.E. multiprogramming system. **CACM 11**, 5 (May 1968), 341-346.

[ELC67] Evans, D. C. and LeClerc, J. Y., Address Mapping and the Control of Access in an Interactive Computer, **Proc. AFIPS 1967 SJCC**, Vol. 30, AFIPS Press, Montvale, N.J., 23-30.

[F68] Fabry, R. S., Preliminary Description of a Supervisor for a Computer Organized Around Capabilities, Quarterly progress report 18, Institute of Computer Research, University of Chicago, 1968, 1-97.

[G72] Gaines, R. S., An Operating System Based on the Concept of a Supervisory Computer. **CACM 15**, 3 (March 1972), 150-156.

[GD72] Graham, G. S. and Denning, P. J., Protection - Principles and Practice, **Proc. AFIPS 1972 SJCC**, Vol. 40, AFIPS Press, Montvale, N.J., 417-429

[G68] Graham, R. M., Protection in an Information Processing Utility, **CACM 11**, 5 (May 1968), 365-369.

[H70] Hansen, P. B., The Nucleus of a Multiprogramming System, **CACM 13**, 4 (April 1970), 238-250.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[IBM] IBM Operating System/360, Concepts and Facilities. C28-6535.

[I68] Iliffe, J. K., **Basic Machine Principles**, MacDonald, London (1968).

[IJ62] Iliffe, J. K. and Jodeit, Jane G., A Dynamic Storage Allocation Scheme, **Computer Journal** 5,3, (1962) 200-209.

[L69a] Lampson, B. W., CAL-TSS Internals Manual, Computer Center, University of California Berkeley, November 1969.

[L69b] Lampson, B. W., Dynamic Protection Structures, **Proc. AFIPS 1969 FJCC**, Vol. 35, AFIPS Press, Montvale, N.J., 27-38.

[L71] Lampson, B. W., Protection, Proc. Fifth Annual Princeton Conference on Information Sciences and Systems, Dept. of Electrical Engineering, Princeton University, Princeton, N.J., March 1971, 437-443.

[N72] Needham, R. M., Protection Systems and Protection Implementations, **Proc. AFIPS 1972 FJCC**, Vol. 41, AFIPS Press, Montvale, N.J., 571-578.

[O72] Organick, E. I., **The Multics System: An Examination of Its Structure**, MIT Press, 1972.

[OC71] Organick, E. I., and Cleary, J. G. A Data Structure Model of the B6700 Computer System. **SIGPLAN Notices** 6,2 (Feb 71), 83-145.

[P73] Price, W. R., Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems. Ph.D. Thesis, Carnegie-Mellon University, 1973.

[S72] Shroeder, M. D., Cooperation of Mutually Suspicious Subsystems in a Computer Utility, Ph.D. Thesis, Mac Tr-104, Mass. Institute of Technology, 1972.

[SS72] Shroeder, M. D. and Saltzer, J. H., A Hardware Architecture for Implementing Protection Rings, **CACM 15**, 3 (March 1972) 157-170.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[W71] Wirth, Niklaus, The Programming Language Pascal. **Acta Informatica** 1 (1971) 35-63.

[WU73] Wulf, W. A. et al, HYDRA: The Kernel of a Multiprocessor Operating System, Carnegie-Mellon University, Computer Science Department report, June 1973.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.