UNCLASSIFIED
Security Classification

AD 750173

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
NONE

3. REPORT TITLE
COOPERATION OF MUTUALLY SUSPICIOUS SUBSYSTEMS IN A COMPUTER UTILITY

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
INTERIM SCIENTIFIC REPORT

5. AUTHOR(S) (First name, middle initial, last name)
MICHAEL D. SCHROEDER

6. REPORT DATE
SEPTEMBER 1972

7a. TOTAL NO. OF PAGES
167

7b. NO. OF REFS
29

8a. CONTRACT OR GRANT NO.
N00014-70-A-0362-0006

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
MAC TR-104

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
NONE

10. DISTRIBUTION STATEMENT
DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

11. SUPPLEMENTARY NOTES
PH.D. THESIS, DEPT. OF ELECTRICAL ENGINEERING, SEPTEMBER 1972

12. SPONSORING MILITARY ACTIVITY
OFFICE OF NAVAL RESEARCH

13. ABSTRACT
This thesis describes practical protection mechanisms that allow mutually suspicious subsystems to cooperate in a single computation and still be protected from one another. The mechanisms are based on the division of a computation into independent domains of access privilege, each of which may encapsulate a protected subsystem. The central component of the mechanisms is a hardware processor that automatically enforces the access constraints associated with a multidomain computation implemented as a single execution point in a segmented virtual memory. This processor allows a standard interprocedure call with arguments to change the domain of execution of the computation. Arguments are automatically communicated on cross-domain calls -- even between domains that normally have no access capabilities in common. The processor, when supported by a suitable software system which is also discussed, provides the protection basis for a computer utility in which users may encapsulate independently compiled programs and associated data bases as protected subsystems, and then, without compromising the protection of the individual subsystems, combine protected subsystems of different users to perform various computations.

DDC
OCT 30 1972
D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

i

DD FORM 1 NOV 65 1473 (PAGE 1)
S/N 0102-014-6600

UNCLASSIFIED
Security Classification

14 KEY WORDS	LINK A ROLE	LINK A WT	LINK B ROLE	LINK B WT	LINK C ROLE	LINK C WT
Multics						
Protection Mechanisms						
Computer Utility						
Protection Domains						
Hardware Protection Mechanisms						
Controlled Information Sharing						
Time-Shared Computer Systems						
Virtual Memory						
Segmentation						
Access Control						

ii

MAC TR-104

COOPERATION OF MUTUALLY SUSPICIOUS SUBSYSTEMS IN A COMPUTER UTILITY

Michael D. Schroeder

September 1972

This research was supported by the Advanced Research Project Agency of the Department of Defense under ARPA Order No. 2095, and was monitored by ONR under Contract No. N00014-70-A-0362-0006.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor, Professor J. H. Saltzer, for encouraging and guiding the research reported in this thesis, and my readers, Professors F. J. Corbató and J. D. Bruce, for reviewing and commenting upon the drafts of the thesis.

I would also like to thank Muriel Webber for typing the drafts and final version of the thesis and drawing the figures. Without her assistance the thesis would have taken much longer to complete.

COOPERATION OF MUTUALLY SUSPICIOUS SUBSYSTEMS IN A COMPUTER UTILITY*

by

Michael D. Schroeder

ABSTRACT

This thesis describes practical protection mechanisms that allow mutually suspicious subsystems to cooperate in a single computation and still be protected from one another. The mechanisms are based on the division of a computation into independent domains of access privilege, each of which may encapsulate a protected subsystem. The central component of the mechanisms is a hardware processor that automatically enforces the access constraints associated with a multidomain computation implemented as a single execution point in a segmented virtual memory. This processor allows a standard interprocedure call with arguments to change the domain of execution of the computation. Arguments are automatically communicated on cross-domain calls -- even between domains that normally have no access capabilities in common. The processor, when supported by a suitable software system which is also discussed, provides the protection basis for a computer utility in which users may encapsulate independently compiled programs and associated data bases as protected subsystems, and then, without compromising the protection of the individual subsystems, combine protected subsystems of different users to perform various computations.

---

* This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, on September 20, 1972 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	2
ABSTRACT	3
LIST OF FIGURES	6
Chapter	
1. INTRODUCTION	7
Different Access for Different Situations	10
Cooperating, Mutually Suspicious Subsystems	12
Objectives	17
Background	24
Approach	28
Thesis Plan	31
Related Work	32
2. STATIC ACCESS, DYNAMIC ACCESS, AND CROSS-DOMAIN CALLS	37
The Segmented Virtual Memory	37
Multiple Domains	38
Domain of Execution	40
An Example of Mode Arrays	42
Cross-Domain Calls	44
Static Access and Dynamic Access	48
Dynamic Access Capabilities	49
Matching References to Capabilities	51
Creation of a New Activation Record	54
Summary	55
3. THE PROCESSOR DESIGN	
The Virtual Memory and Static Access Capabilities	57
Generating Virtual Memory Addresses	62
Call and Return	69
Cross-Domain Call and Return	74
Dynamic Access Capabilities	77
Propagation of Tags	86
Multibit Tags	93
Pointers as Arguments	106
The Escape Hatch	115
Processor Response to Exceptional Conditions	116

Chapter	Page
3. Continued	
An Associative Memory for Dynamic Access Capabilities	118
Entry and Label Variables	123
Bit Addressing	132
Summary	133
4. SOME COMMENTS ON THE SUPPORTING SOFTWARE	136
A Distributed Supervisor	137
The File System	138
Protected Subsystems	143
Permission Flags for Segments	144
Controlling the Gate Attribute	148
Examples of Access Control Lists	149
Other Issues	152
Summary	154
5. CONCLUSIONS	155
Areas for Further Research	160
BIBLIOGRAPHY	163
BIOGRAPHICAL NOTE	166

LIST OF FIGURES

Figure		Page
2-1:	Example access mode arrays for three domains of a process.	43
3-1:	The descriptor segment and related registers.	58
3-2:	Register and storage formats for address formation.	64
3-3:	Algorithm for effective pointer generation.	66
3-4:	Format of the argument list for an interprocedure call.	70
3-5:	Completed argument list format for an interprocedure call.	78
3-6:	A dynamic access stack frame for a single cross-domain call.	80
3-7:	Two consecutive cross-domain calls with one argument cascaded through both.	94
3-8:	Dynamic access stack containing frames for three unreturned cross-domain calls.	99
3-9:	Modified algorithm for effective pointer generation with a multibit tag.	100
3-10:	Format of a dynamic access capability giving read and/or write access to a subsegment.	103
3-11:	Algorithm of the SPPn instruction.	109
3-12:	Three cases of the SPPn instruction.	110
3-13:	Format of the associative memory for dynamic access capabilities.	119
3-14:	Specification of a single entry variable as an input argument.	126
4-1:	A representative directory hierarchy.	140
4-2:	Use of the directory hierarchy to organize information stored on-line.	141
4-3:	Initial state of the domain table for a process.	145

CHAPTER 1

INTRODUCTION

When considered in conjunction with computer systems, privacy and protection are easily confused. Privacy is a social and legal issue. Privacy means that the dissemination of information of or about a person or group is controlled by that person or group. New emphasis has been placed on guaranteeing rights of privacy in the face of the increased ability provided by large, sophisticated computer-based information systems to gather, catalogue, and analyze data. Protection, on the other hand, is a technical and economic issue. Protection means the mechanisms for specifying and enforcing control. Guaranteeing rights of privacy requires protection mechanisms, but many other issues are involved as well. Protection is a means and privacy an end.

In this thesis interest is centered on protection mechanisms within computer systems. The viewpoint is that of a computer system designer who is intent upon providing efficient protection mechanisms applicable to a wide range of problems. Questions of privacy influence this effort to the extent of implying criteria which must be met before such a computer system can be applied to those problems where privacy is an issue. The thesis, however, contains little explicit consideration of privacy.

To further define the scope of the thesis, consideration is limited to problems of hardware and software organization. While it is recognized that issues such as installation security, communication line

security, hardware reliability, and correctness of hardware and software implementations of algorithms must be considered in order to achieve the secure environment required for useful application of protection mechanisms, these topics are beyond the scope of the thesis.

Protection mechanisms are defined to be those components of a computer system that control access of executing programs to objects inside the system. This definition recognizes that executing programs carry out inside the system the intentions of the people outside the system. Thus, executing programs are the active agents that must be controlled in a computer system. The definition admits a large variety of mechanisms, ranging from write-permit rings on magnetic tape reels to sophisticated memory protection hardware and including access controls implemented in software.

All such protection mechanisms are directed toward two overall objectives. The first can be labeled user self-protection. People are not infallible. They frequently make mistakes when conceiving algorithms, specifying algorithms as programs, and applying programs to particular problems. Protection mechanisms permit a redundant specification of intention to be made and enforced. For example, the write-permit ring mentioned above allows the intention that a tape not be altered in a particular situation to be enforced on the basis of specifications other than those provided by the executing programs. In general, enforcing limits on the objects an executing program may access can detect certain types of errors and prevent inadvertant damage by incorrect access.

The second objective of protection mechanisms is control of user interaction. This objective surfaces in computer systems providing computation and information storage services to more than one person or group. The multiple users of a computer system may have different goals and be responsible to different authorities. A diverse community will use the same system only if it is possible for its members to be independent of one another. On the other hand, a great potential benefit of a multiple user computer system is its ability to encourage users to communicate, cooperate, and build upon one another's work. Such interaction is generally based on users sharing stored information. Thus, total user separation, while relatively easy to achieve, negates a potential benefit. Required to achieve this benefit is controlled user interaction. Controlling the ability of executing programs to access objects in the system is the internal manifestation of controlling user interaction.

While important in any system serving multiple users, the objective of controlling user interaction is a key part of the service objectives defining a computer utility. In the remainder of this thesis, protection mechanisms are considered in the context of a large, general-purpose, interactive, multiplexed computer system functioning as a computer utility. This context provides a severe test of protection mechanisms, and at the same time offers a large reward for devising economic, natural protection mechanisms that can guarantee total user separation when desired, permit unrestricted user cooperation when desired, and provide many intermediate degrees of control.

Different Access for Different Situations

The essential observation underlying all protection mechanisms is that executing programs need different access in different situations. For instance:

- The programs in different computations need different access. In a computer utility serving a university community, for example, programs executing in a computation directed by the registrar to manipulate academic records need different access to stored data than programs executing in a computation directed by a system administrator to generate monthly bills for computer usage.

- The same programs in different computations need different access. In the previous example, both computations may share the use of a program for manipulating hash-coded index tables.

- Different programs in the same computation need different access. A user-defined program executing in some computation may invoke a supervisor program to initiate I/O channel operation. Presumably, the executing supervisor program has access to channel assignment data and to the start I/O instruction of the processor -- access denied the executing user program. Yet both executing programs are logically part of the same computation.

- The same program in the same computation needs different access for different circumstances. For example, a closed subroutine providing string comparison functions for all programs written in PL/I needs different access when invoked by a supervisor program than when invoked by a user-defined program.

In general, the access permission that should be associated with an executing program depends upon the specific circumstances of its execution.

A significant result of previous work [7,19,20] on protection mechanisms has been the development of a model that applies to situations where the access of executing programs needs to differ. This model is based on the concept of domains which are sets of access capabilities. The access of an executing program depends upon the domain of execution in which the program finds itself. An attempt by

by an executing program to access something in the system will succeed only if a capability allowing that access is an element of the domain of execution. (Ways of defining the capabilities in domains and of determining the domain of execution at any given time will be discussed later.)

There is practically no limit to the types of capabilities that can be defined. A capability is simply an ordered pair specifying the name of something in the system and the operations which can be performed on that object. Thus, there can be capabilities to allow any well-defined operation on any named object in a system. Capabilities to read and write data items and to execute programs are obvious examples. Capabilities to do I/O over a specific channel, to add capabilities to domains, and to change the domain of execution to a particular domain are less obvious examples. The different types of capabilities present depend upon the particular system being considered.

The domain model provides a conceptual framework in which to consider the protection mechanisms of any system. For instance, the model is easily related to the examples given previously. By using a distinct domain of execution for each computation in a system, executing programs automatically assume the access capabilities associated with the computation of which they are part, as required in the first and second examples. By alternately using two domains of execution for a single computation, the supervisor/user distinction of the third and fourth examples can be dealt with. Multiple domains per computation can deal with more complex situations as well.

Cooperating, Mutually Suspicious Subsystems

One of the most complex instances of multiple domains in a single computation is the case of mutually suspicious subsystems cooperating in the same computation. This case provides a good test of a set of protection mechanisms. Since protection mechanisms that can deal with cooperating, mutually suspicious subsystems are the central concern of this thesis, further discussion of this case is appropriate.

An essential idea underlying the case of cooperating, mutually suspicious subsystems is that of a protected subsystem. A protected subsystem is a collection of programs and data bases that is encapsulated so that other executing programs can invoke certain component programs within the protected subsystem, but are prevented from reading or writing component programs and data bases, and are prevented from disrupting the intended operation of the component programs. The supervisor of most computer systems is a good example of a protected subsystem. A few systems (notably the CAL system [15,18,26,27] and Multics [5,21,23]) allow normal users to define their own protected subsystems and, with certain restrictions, make them available to be invoked by the programs of other users.

Protected subsystems are useful for implementing complex controls on access to data bases and for maintaining the secrecy of proprietary algorithms which are shared. In the first case the programs of a protected subsystem act as caretakers for the contained data bases and interpretively enforce arbitrarily complex controls on access to them. Programs outside the protected subsystem are allowed to access the contained data bases only by invoking the caretaker programs. The

algorithms manifest in these programs may judge the propriety of the requested access based on information provided by the system on the identity of the invoking program and the circumstances of invocation. The caretaker programs may even record each access request in one of the encapsulated data bases. No action by an outside program can disrupt the operation of the caretaker programs in judging, performing, or recording requested accesses.

In the case of maintaining the secrecy of shared, proprietary algorithms, the programs in a protected subsystem may be invoked by outside programs but not read by them. In addition, none of the temporary or permanent data associated with the programs in the protected subsystem may be read or written by outside programs. Thus, the programs in the protected subsystem may be used by other programs, but no aspect of their structure examined.

So far the properties of a protected subsystem have been developed from the point of view of a program outside invoking a program inside. It is also conceivable that in the course of its operation a program inside a protected subsystem might need to invoke a program outside. This reverse operation is useful only if it does not compromise the protected subsystem.

A case of cooperating, mutually suspicious subsystems arises when two or more protected subsystems are involved in the same computation. An example of this situation occurs if a program inside a protected subsystem invokes a program outside that happens to be inside another protected subsystem. An example involving a less direct form of cooperation arises if programs in two different protected subsystems are alternately

invoked by programs outside both. Arbitrary structures involving any number of protected subsystems can be imagined.

The domain model deals naturally with the notion of protected subsystems. Each encapsulation is just a different domain containing capabilities to execute the component programs and read or write the component data bases. The case of mutually suspicious subsystems cooperating in a single computation corresponds to a multidomain computation in which the domains involved have few access capabilities in common.

A computer utility which allows user-defined protected subsystems to cooperate in computations could be put to good use. For example, imagine that a consulting firm has developed and calibrated a demand model for interzonal traffic flow in a large metropolitan area. Model development, data gathering, and model calibration were done at great expense, and the firm wishes to sell the use of its model in order to profit from the investment. To this end the model is implemented in a computer utility. The resulting subsystem includes a set of programs which embody the prediction algorithms, and several data bases containing socio-economic parameters and door-to-door survey results. In order to vend this subsystem to other users of the computer utility, the consulting firm encapsulates it as a protected subsystem.

Now imagine that a planning board for the area is studying expressway networks to meet future transportation needs. As part of this task they have constructed a model which calculates level-of-service parameters for an expressway network, given the physical attributes and the traffic volume. This model, along with a wealth of related data, is

also represented as a protected subsystem in the computer utility. For one aspect of their study the planning board will perform a series of network equilibrium flow analyses for various expressway configurations. In these calculations the planning board will use the demand model vended by the consulting firm.

The resulting computation is a case of cooperating, mutually suspicious subsystems. The consulting firm wishes to avoid divulging their expensively gathered data and expensively devised algorithms, and wishes to make secure records of use for later billing. The planning board wishes to avoid premature, uncensored release of the various routing and cost information for potential expressway configurations that is accessible to their programs for this and other phases of the study.

The encapsulation of the consulting firm's demand model as a protected subsystem guards their proprietary interests. Access by the planning board subsystem is restricted to invoking a few carefully selected programs. The planning board subsystem cannot read the code or the data bases of the demand model, nor subvert its operation in any way. The consulting firm subsystem can make protected usage records as it operates from which bills can be generated later.

Likewise, the encapsulation of the planning board programs and data bases protects the interests of the planning board. The consulting firm subsystem cannot read or alter the sensitive data that is accessible to the planning board programs. The consulting firm subsystem just has temporary access to whatever input parameters the planning board subsystem provides each time that it invokes the demand

model.* Also, the only way that the consulting firm subsystem can influence the operation of the planning board subsystem is with the values of the answers calculated by the former for the latter.

This simple example involving two protected subsystems is representative of a large class of potential applications for a computer utility that supports cooperating, mutually suspicious subsystems. Such a facility would permit, among other things, the packaging of knowledge in an active form as program complexes and the marketing of this knowledge in a way that preserves the proprietary interests of the seller and limits the possibility of harm to the customers. Of course, this facility cannot guarantee that a marketed program complex correctly performs the functions that it is advertised to perform. Issues of program certification are beyond the scope of this thesis. The important point is that, even if a program malfunctions or behaves maliciously, the use of protected subsystems limits the resulting damage.

A computer utility that supports cooperating, mutually suspicious subsystems would favorably impact casual programming users as well as those affiliated with expensive proprietary projects or those manipulating sensitive data. An important activity of most programming users in a computer utility is sharing programs and program complexes with one another. The access environment in which the programs of such a user

---

*In some cases it may be desirable to guarantee that a protected subsystem invoked by another cannot remember and later divulge the values of input parameters. This problem appears to be very difficult to solve in a purely technical way and will not be considered in this thesis. A practical solution probably involves program certification and means for legal redress in cases of unauthorized release of such information.

normally execute on his behalf could be set up as a protected subsystem. Then the user could arrange for programs borrowed from other users to execute outside of this "home" protected subsystem. In this manner the borrowed programs could be invoked without giving them access to all the programs and data of the borrower. If the borrowed program is malicious, or malfunctions, the damage it can do is thereby limited. The lending user also would have the option to encapsulate the lent program or program complex in a protected subsystem of its own and thus insulate it from the programs of the borrower. The result of this pattern is an environment in which casual sharing of programs is encouraged by the ability to limit the potential damage to the participating users.

Objectives

The objective of the research reported here is to devise protection mechanisms that allow cooperating, mutually suspicious subsystems to be implemented efficiently and naturally in a computer utility. As has been suggested, this case corresponds to a multidomain computation in which the domains have few capabilities in common. The essential difficulty is that common capabilities seem to be a prerequisite to cooperation and communication among subsystems encapsulated in different domains. Thus, in this respect, cooperating, mutually suspicious subsystems is a "worst case" problem. Means must be provided for programs executing in different domains to communicate even when these domains normally have no capabilities in common. The mechanisms developed in this thesis, however, do not force the sets of capabilities that are the domains in a computation to be disjoint. There are no

arbitrary restrictions on the capabilities that can be included in the domains As a result, these protection mechanisms are able to enforce as well a large variety of protection relationships that are less restrictive than those associated with cooperating, mutually suspicious subsystems.

The words "efficiently" and "naturally" are used to convey the aim of developing practical mechanisms which meet the functional objective -- mechanisms that people actually might build and use. This requirement is manifest in three specific design criteria: economy, simplicity, and programming generality. Economy, obviously related to efficiency, encompasses two considerations. First, the cost of building the protection mechanisms should be small enough relative to the cost of the entire system that the system designers are willing to include such mechanisms in the system despite an untested market. Second, the cost of using the mechanisms should be low enough that it is not an important consideration in determining the degree of access control to be used in a particular application. Cost in the first case is the per system fixed cost of providing the protection mechanisms. Cost in the second case is the marginal cost of application, and includes the subsystem complexity and user inconvenience that result from the use of the mechanisms, as well as any associated extra storage space and execution time.

Simplicity often leads to economy, but is identified as a separate criterion for a more important reason. If a set of protection mechanisms is to be accepted and used, then there must be confidence that no way exists to circumvent it. The best way to achieve confidence

is to keep the mechanisms simple so that they may be completely understood. In this respect standard techniques for controlling complexity, such as functional modularity and adherence to a few overall design principles, can help.

Programming generality, the third criterion, is adopted as the specific manifestation of naturalness. First defined by Dennis [8,9,10], it is a property that has proven the key to encouraging users of a computer utility to communicate, cooperate, and build upon one another's work. As used here, it means that independent programs may be combined in various ways into larger units without understanding or altering the internal organizations of the programs. The idea is that independently written and compiled programs of one or more computer utility users can be used as building blocks which can be combined in various ways into different program complexes to perform different computations. In the course of such activity a single program or program complex may have to operate in many different protection environments. If the protection mechanisms of a computer utility are to be consistent with the criterion of programming generality, then, it must be possible to change the protection environment of a program or group of programs without altering the internal structure of the program or group.

The programming generality criterion generates some specific functional requirements for the protection mechanisms of this thesis. These requirements are now developed into a more precise statement of the thesis objective. The initial assumption is that a call with arguments is an accepted method of communication between independent programs and should be supported in any general purpose computer system such as a

computer utility. In keeping with the criterion of programming generality, many systems allow separately compiled, independent programs to communicate via a call that passes arguments. The only requirements for such communication are that the programs involved use the same calling conventions and that they agree on the number, kind, and order of the arguments. No explicit knowledge of the circumstances of execution need be embedded in the calling or the called program. A single call statement in some program can invoke many different programs depending upon the circumstances at the time the compiled version of the statement is executed.

It seems reasonable that the domain in which a program will execute when called not be part of the knowledge required in order to write or compile programs that invoke it. A natural consequence of this view is that the standard call used between independent programs should automatically change the domain of execution if, in a given circumstance the called program is supposed to execute in a different domain than the current domain of execution. The programmer or the compiler cannot do anything special for calls that cause the domain of execution to change (as opposed to calls that leave the domain of execution unchanged) because programming generality requires that such calls not be differentiated when a program is written or compiled.

The passing of arguments with a call needs further consideration in light of the requirement that a standard call between independent programs can change the domain of execution if necessary. An argument may be any data item or set of data items that the calling program can specify. There are two fundamental methods that can be used to pass

arguments with a call. One is to copy the values of the arguments into memory locations or registers where the called program expects to find them. The second is to leave most arguments in their normal memory locations and inform the called program of the addresses of the arguments. The criterion of economy makes it imperative to support the second method. The copying involved in the first method would be so expensive that the passing of large arguments (arrays, structures, etc.) would be discouraged. Further, the functional effect of passing arguments by value can be achieved when required with a mechanism based on passing arguments by address. The reverse is not true, for modification of an argument passed by value immediately changes only a copy of the source variable, not the source variable itself. More sophisticated linguistic constructions for passing arguments, such as passing them by name, also can be implemented based on mechamisms that pass arguments by address.

Passing arguments by address, however, can generate problems in the case of calls that happen to change the domain of execution. For such arguments to be referenced by the called program, the domain of execution of the called program must contain capabilities which allow access to the memory locations containing the arguments. The domain of the calling program and the domain of the called program essentially are required to have capabilities in common permitting the arguments to be referenced.

There are several ways that this need for common argument capabilities conceivably could be met. One approach is to allow cross-domain calls only if the capabilities in the calling domain form a subset of

arguments with a call. One is to copy the values of the arguments into memory locations or registers where the called program expects to find them. The second is to leave most arguments in their normal memory locations and inform the called program of the addresses of the arguments. The criterion of economy makes it imperative to support the second method. The copying involved in the first method would be so expensive that the passing of large arguments (arrays, structures, etc.) would be discouraged. Further, the functional effect of passing arguments by value can be achieved when required with a mechanism based on passing arguments by address. The reverse is not true, for modification of an argument passed by value immediately changes only a copy of the source variable, not the source variable itself. More sophisticated linguistic constructions for passing arguments, such as passing them by name, also can be implemented based on mechamisms that pass arguments by address.

Passing arguments by address, however, can generate problems in the case of calls that happen to change the domain of execution. For such arguments to be referenced by the called program, the domain of execution of the called program must contain capabilities which allow access to the memory locations containing the arguments. The domain of the calling program and the domain of the called program essentially are required to have capabilities in common permitting the arguments to be referenced.

There are several ways that this need for common argument capabilities conceivably could be met. One approach is to allow cross-domain calls only if the capabilities in the calling domain form a subset of

the capabilities in the called domain. Thus, any argument that the calling program happens to pass is accessible to the called program. If one domain is a subset of another, however, then the two domains certainly cannot encapsulate mutually suspicious subsystems. Thus, mechanisms based on this approach do not allow direct cooperation between mutually suspicious subsystems by a call from one to the other.

Another possibility is the a priori inclusion in a domain of the precise capabilities required to reference all arguments that will be passed with all possible incoming cross-domain calls. This method is not practical, however, because the binding of variable names to memory locations can vary dynamically as programs execute. This dynamic binding makes it impossible to include in advance the proper capabilities in the called domain. Even if this problem could be solved, a priori inclusion of capabilities for arguments would violate the criterion of programming generality as it applies to domains by requiring detailed knowledge of programs making incoming cross-domain calls in order to define a domain.

A third possibility is the establishment by mutual agreement between a domain and each potential calling domain of a special buffer for transmitting arguments. Capabilities to reference the buffer would be common to the domains. The use of a buffer for arguments on all calls, whether they change the domain of execution or not, really means that arguments are passed by value, a technique discarded earlier as uneconomical. The use of a buffer only for cross-domain calls compromises program generality.

proposed here extend the idea so that protected subsystems may be used as building blocks as well. Independently defined protected subsystems may be combined in different ways as cooperative partners in various computations without either explicitly modifying the definitions of the protected subsystems or altering the component programs.

Background

The protection mechanisms developed in this thesis are strongly influenced by the structure of Multics. Multics is a prototype computer utility developed at M.I.T.'s Project MAC in a joint effort with Honeywell Information Systems, Inc. and now operating at several sites. (See [5,21,23] for an introduction to Multics.) No claim is made that Multics provides the best possible environment in which to develop practical protection mechanisms which meet the stated objectives. Multics is used because it is the closest operational approximation to a computer utility currently existing and because it is structured to allow programs to be used as building blocks in the manner discussed earlier [10]. Because Multics is an operational system that has proven effective in meeting many of the objectives of a computer utility, there can be some confidence that protection mechanisms designed for a similar system environment are based on realistic assumptions. Protection mechanisms have a tendency to permeate a system. In this case the existing Multics can be used as a concrete system environment in which to test the protection mechanisms by tracing their full system implications. The fact that the protection mechanisms described here represent

the evolution and improvement of an existing, useful system lends strength to the contention that they are practical.

The relationship of the protection mechanisms described in this thesis to Multics is made clearer by briefly reviewing certain aspects of the system. Multics already includes a fairly sophisticated set of protection mechanisms, particularly for controlling access to stored information. On-line storage is logically organized as a collection of disjoint segments of information. Segments are variable length arrays of bits that may contain any collection of information, e.g., symbolic programs, compiled programs, data files, temporary storage for on-going computations, or system data bases. Any segment is potentially accessible to any executing program.

Multics is organized so that separately compiled programs may be used as building blocks. While this goal influences the organization of the entire system, several features are particularly important. Compiled programs are represented as segments of pure procedure so that multiple computations may simultaneously execute the same shared procedure segment. Interprocedure communication is by a standardized interprocedure call and interprocedure linking is done dynamically the first time a particular call is executed in a computation. The normal execution environment includes a push-down stack for procedure activation records, allowing procedures to be combined in patterns that may generate recursive invocations.

Machine language programs in Multics execute in a segmented virtual memory where segments are identified by number. The two-part address (s,d) identifies displacement d in the segment numbered s. Processors [17]

contain logic for automatically translating two-part addresses into absolute memory addresses. Translation is done using the segment number as the index into a table of segment descriptors called the descriptor segment. When an executing program wishes to reference a segment stored on-line, the segment must first be added to the virtual memory by assigning it an unused descriptor segment entry, an operation automatically performed by the supervisor when the first attempt to reference the segment is detected. (See [1] for a more detailed description of this virtual memory.)

The combination of an execution point and a virtual memory is called a process. A process with a new virtual memory is created for each user when he logs in to the system. Thus, each user's process has its own descriptor segment. In this context it is convenient to use the term computation to mean the activity of the process of a user.

The allowed access of executing programs to each segment stored on-line is specified by an access control list. An access control list entry essentially associates a domain name with some combination of read, write, and execute access to the segment. A domain name is the combination of the name of some system user with one of the integers 0 through 7, and designates one of the eight fixed domains associated with the process of some user. Each access control list entry defines a capability in the named domain that allows the indicated access to the segment. Manipulation of access control lists is controlled by other access control lists that are part of a hierarchy of segment catalogues.

As implied in the previous paragraph, Multics allows multiple domains to be associated with a single process. The domain of execution

is changed by the standard interprocedure call. Arguments are passed by address. The problem of arranging for the common argument capabilities between the calling domain and the called domain is solved by forcing all domains associated with a single process to form a linearly nested collection of sets with respect to segment referencing capabilities [14]. Cross-domain calls are allowed only if the called domain is a superset of the calling domain. This linear nesting generates a total ordering with respect to access privilege on the domains associated with a process that simplifies many aspects of the system in addition to cross-domain calls. As pointed out earlier, however, this approach does not allow direct cooperation by interprocedure call of domains encapsulating mutually suspicious subsystems. In fact, it does not even allow mutually suspicious subsystems to be part of the same process.

The protection mechanisms described in this thesis are similar in overall approach to those in Multics. They developed from considering ways to improve the efficiency of cross-domain calls for the case of linearly nested domains, and then considering ways to remove entirely the linear nesting restriction. The result of the first phase was a previously reported [25] hardware architecture for automatically handling cross-domain calls with arguments in the case of linearly nested domains. This design, implemented in a new processor for Multics being constructed by Honeywell, contains the seeds of many ideas that are expanded as the central part of this thesis into a hardware architecture for automatically handling the unconstrained case of cross-domain calls with arguments. In addition to devising hardware that automatically performs general cross-domain calls, the second phase included isolating those

software system aspects simplified by the linear nesting restriction in Multics and exploring the extensions required to take advantage of the improved hardware protection mechanism. While the research reported here was conducted within the specific context of Multics, the results presented in this thesis are quite general and applicable to a wide variety of system environments.

One additional aspect of the relationship of this thesis to Multics should be mentioned. The primary programming language in Multics is PL/I [16]. Not only are most user programs written in PL/I, but almost all of the system programs are as well [6]. Certain aspects of the high-level languages in use in a system can influence the design of the protection mechanisms. For example, if a standard interprocedure call is to be able to change the domain of execution, then the protection mechanisms must accomodate all the semantic implications of a call as defined in the various high-level languages in use. Because PL/I has proven fairly well suited in Multics to the goal of allowing separately compiled procedures to be used as building blocks, it will be used in this thesis as a representative high-level language to be accomodated by the protection mechanisms. Because of the richness of PL/I, the protection mechanisms developed in this context will support equally well many other high-level languages in use today.

Approach

Almost all interesting things in a computer utility are ultimately manifest as stored information. Processes, I/O streams, directories, access control lists, domains, capabilities, data segments, and

procedures, to name a few examples, are all represented as stored information. It follows that control of access to stored information is a basic function of protection mechanisms. This observation is reflected in the design of the protection mechanisms developed in this thesis.

The hardware component of these mechanisms is a processor that supports a multidomain computation implemented as a single Multics-like process, i.e., as a single execution point in a segmented virtual memory. This processor automatically controls access to a primitive set of information objects by interpreting and enforcing capabilities representing both the encapsulations defined by the domains of the process and the arguments passed with cross-domain calls. The encapsulation capabilities recognized are those allowing direct read or write references to segments and those allowing transfers of the execution point from one virtual memory location to another (including transfers that represent cross-domain calls). The cross-domain argument capabilities recognized by the processor allow most types of data items defined by current, general-purpose, high-level programming languages to be passed as arguments with cross-domain calls. The processor automatically adds the required argument capabilities to the called domain when a cross-domain call occurs and removes them when the subsequent return occurs.

The hardware processor provides a general foundation for a software second layer of the protection mechanisms. To get off the ground, one of the domains supported by the hardware is given capabilities allowing direct read/write access to the segments containing the capabilities that the hardware enforces. Thus, the hardware is circularly applied

to protect the stored capabilities that control it. This special domain becomes the domain encapsulating the most highly privileged part of the supervisor. Manipulation of the capabilities controlling the hardware by procedures executing in other domains can only be done interpretively by calling the supervisor procedures. Part of the supervisor is a file system that allows users to define the encapsulation of segments as protected subsystems and control the sharing of such protected subsystems. Control on such specification is enforced interpretively by the file system procedures based on information stored in the file system data segments. Software to provide other supervisor functions, such as multiplexing processors to create many simultaneous processes, is also encapsulated in the supervisor domain.

Notice that the hardware processor interprets and enforces only capabilities allowing direct access by executing procedures to a few simple information objects: segments, arguments, and domains. This fairly simple processor, however, manages to handle automatically the vast majority of accesses to objects by executing programs, creating a relatively efficient system. More complex information objects can be created and access to them enforced by using the basic facilities of this hardware. The method is to define protected subsystems that encapsulate segments containing the stored manifestations of these more complex objects with procedures which provide interpretive access to them. The file system, itself an example of this encapsulation strategy, provides the basic tools allowing users to perform such encapsulations for themselves. Examples of information objects to be discussed later that are defined and controlled by the file system are access control

lists, directories, and protected subsystems. Other information objects defined and controlled by the supervisor domain might be processes, I/O streams, and interprocess message queues. There is no limit to the complexity of the information objects and associated access controls users can define by creating their own protected subsystems.

Thesis Plan

The central material of this thesis is the development of the hardware processor. Chapter 2 presents the conceptual basis for the protection mechanisms in the processor. The properties of encapsulation and argument capabilities are derived and the steps involved in performing a cross-domain call are considered. These ideas are expanded in Chapter 3 into a description of a processor that fully supports a computation involving multiple independent domains and that automatically performs cross-domain calls with arguments -- even between domains that normally have no capabilities in common. Included in this chapter is a discussion of the machine code to be generated by compilers to take full advantage of the processor protection mechanisms and of the high-level programming language extensions required to allow programmers to control the cross-domain call mechanisms.

The software second layer is considered in less detail than the hardware first layer. In Chapter 4 a file system is described by outlining extensions to the Multics file system that permit it to deal with protected subsystems. The extensions necessary are the addition of protected subsystems to the objects catalogued by the file system and the recasting of access control lists in terms of protected subsystem

names rather than domain numbers. A first approximation for the remaining software system required to produce a computer utility in which segments and protected subsystems can be used as building blocks is obtained by adopting the rest of the Multics software organization and implementation with essentially no changes. The purpose of the material in Chapter 4 is to demonstrate one way to harness the processor protection mechanisms so that users can define and control the sharing of protected subsystems.

Chapter 5 concludes the thesis by summarizing the important ideas presented and indicating those areas where it appears that further research would be fruitful.

Related Work

Before beginning the body of the thesis it is appropriate to review the published work related to the thesis topic. The relationship of the research reported here to Multics has already been discussed and the pertinent references cited. A complete bibliography of publications on Multics appears in [5].

As indicated earlier, the concept of programming generality was developed by Dennis [8,9,10]. His main concern has been tracing the implications of this concept on language and system facilities for manipulating structured information. The implications of programming generality on protection mechanisms are not developed in his papers. Vanderbilt [28], building on the work of Dennis, has developed an abstract model of information structures for controlling sharing in a computer utility that is consistent with the criterion of programming

generality. He gives little consideration to methods of implementation, however, and certain characteristics of the model, such as prohibition on direct sharing of data among users and the inability to revoke access permission once given, would be unacceptable in a practical facility.

The domain model finds its origin in the spheres of protection of Dennis and VanHorn [7]. Lampson [19,20] first recognized the power of the idea as an abstraction for understanding protection mechanisms, and was responsible for developing the idea into a useful conceptual tool. A recent paper by Graham and Denning [13] uses Lampson's model to explain the diverse protection mechanisms in various existing systems.

Aside from the domain model, essentially no theory of protection mechanisms exists. The rest of the literature in the field consists of descriptions of proposed or existing systems with various protection properties. Wilkes [29] divides the various protection schemes into two classes: list oriented mechanisms and ticket oriented mechanisms. With the first approach, control of access to an object is specified in lists maintained by the protected subsystem (usually the supervisor) that is the custodian of the object. An attempt to access an object is validated by matching the name of the requesting domain against a list of domains authorized to access the object. Multics is an example of a system implementing sophisticated list oriented protection mechanisms. With the second approach, permission to access objects in the system is embodied in unalterable tickets which may be distributed to the domains in the system. Presentation of the proper ticket is all that is required to gain access to an object. The CAL system [15,18,26,27] implements sophisticated ticket oriented protection mechanisms that

allow multiple domains to be associated with a single computation. In the CAL system the domain of execution is changed by the explicit invocation of a special supervisor function, a method not consistent with the criterion of programming generality. A significant advantage of the ticket approach to access control is that only a fairly simple mechanism for manufacturing and controlling tickets need be embedded in the most privileged part of the supervisor. With the list approach, most of the file system must be protected at this level. A significant disadvantage of the ticket approach is the difficulty of revoking access once a ticket has been given to some domain in the system. With both list and ticket oriented mechanisms direct sharing of objects among the domains in the system can be implemented.

In addition to the two basic classes of protection schemes suggested by Wilkes there exists a third class which can be called message oriented mechanisms. (Actually, in some ways this class is a variant of the list oriented mechanisms.) With this approach no direct sharing of objects is allowed. Instead each object is under the exclusive control of some process. All sharing is done via interprocess messages that are managed by the supervisor. To reference a data file belonging to some process, for example, another process would send a message specifying the requested reference. The message is tagged by the supervisor with the identity of the sending process. The receiving process decides on the basis of this tag whether or not the request should be honored. The result of the request is communicated by a return message. The RC-4000 system [2] is an example of a message oriented system. While

the message oriented mechanisms have an appealing simplicity, in practice they are awkward to work with and inefficient.

Most commercially available computer systems have fairly primitive protection facilities intended only to provide user separation. IBM and other major manufacturers have recently become concerned about certifying their systems so that this total user separation can be guaranteed. Aside from Multics, which is available from Honeywell as a special product, however, no operational or announced commercial system can support user-defined protected subsystems, and very few even allow controlled sharing of programs and data among users.

Very little work has been done on hardware mechanisms for supporting sophisticated protection mechanisms. The protection facilities of most systems are implemented largely in software using hardware with fairly primitive protection mechanisms. The machines of the Burroughs Corporation [3] were the first to implement address mapping using segment descriptors, the technique that has proven to be the basis of most sophisticated protection hardware. Evans and LeClerc [11] were among the first to suggest that address mapping hardware including access control mechanisms could be constructed to help implement efficiently computations involving several domains. The hierarchic domain structure they propose is similar to the model developed by Vanderbilt. The Hitac 5020 [22] was one of the first hardware processors to implement more than the simple supervisor/user form of multiple domain protection. The scheme used is patterned after the linearly nested domains of Multics. The Hitac processor, which closely follows the implementation suggested by Graham [14], is not able to change the domain of execution

automatically. The most sophisticated multiple domain protection hardware that has actually been constructed is embodied in the new Multics processor [25] being built by Honeywell. Several projects are currently underway to construct processors with similar protection mechanisms, but no published material is available yet that describes these efforts. Fabry [12] has proposed hardware protection mechanisms specifically designed to support ticket oriented protection mechanisms, but which provides little assistance with the previously mentioned problem of locating already distributed tickets in order to revoke access.

CHAPTER 2

STATIC ACCESS, DYNAMIC ACCESS, AND CROSS-DOMAIN CALLS

The hardware component of the protection mechanisms described in this thesis is a processor that supports a multidomain computation implemented as a single Multics-like process. (Recall that such a process is a single execution point in a segmented virtual memory.) This processor contains logic for implementing a segmented virtual memory, for enforcing the various virtual memory access constraints that represent the encapsulation of procedure and data segments in several domains, and for performing cross-domain calls. In this chapter some of the functional characteristics of this processor are introduced.

The major emphasis of the discussion is on the types of capabilities the processor interprets and enforces, the dynamic behavior of these capabilities, and the steps involved in performing a cross-domain call. The discussion is fairly general, making little reference to specific implementation techniques.

The Segmented Virtual Memory

The segmented virtual memory of a process was introduced in Chapter 1 during the discussion of Multics. The structural characteristics pertinent to the material in this chapter are briefly reviewed here.

1. A segment is a variable length array of bits.

2. Each segment in the virtual memory is identified by a unique non-negative integer.

3. The two-part address (s,d) specifies the dth bit in the segment numbered s.

4. All references to memory generated by an executing machine language program, whether for indirect words, instruction operands, or the next instruction to execute, are specified by two-part addresses. The number of bits actually referenced by the address (s,d) is implied by the circumstances of the reference.

The processor logic required to implement the segmented virtual memory of a process is described in the next chapter.

As indicated in Chapter 1, in order for a procedure executing as part of a process to reference a segment stored on-line in the computer utility, the segment must first be added to the virtual memory of the process. In this chapter it is assumed that the virtual memory of a process somehow already contains all segments that are to be referenced.

Multiple Domains

Each domain associated with a process is defined by a set of capabilities for referencing parts of the segmented virtual memory of the process in various ways. Methods for specifying which capabilities are included in each domain will be considered in Chapter 4. The general form of any capability is:

(name,mode)

where the first element identifies some object in the system and the second element specifies the operations that the capability allows to be performed on that object. With respect to processor enforced control on access to the segmented virtual memory of a process, the relevant ranges of values for names and modes are fairly restricted. All relevant names

may be expressed as:

$$(\underline{s},\underline{d},\underline{\ell})$$

where $\underline{s}$ and $\underline{d}$ are a two-part address, and $\underline{\ell}$ is the number of contiguous bits starting at the named location. This set of names allows the identification of any contiguous chunk of bits in any segment, i.e., any subsegment. The range of values for the mode is limited by the architecture of the processor. It is possible to structure the mode to individually control any processor distinguishable operation or set of operations on the contents of a segment. For example, the use of the subsegment named in a capability as the operand of each separate machine instruction could be individually controlled.

For the purpose of encapsulating procedure and data segments in a domain, an even more restricted set of names and modes is adequate. With the exception of the gate capability introduced later, the capabilities used to express encapsulation that are recognized by the processor name only whole segments and allow independent control of just three operations. The operations are executing the contents of a segment as machine instructions, reading from a segment, and writing in a segment*. The encapsulation capabilities recognized by the processor, then, are

---

*Actually, capabilities that included no mode information at all would be adequate for the purpose of encapsulating procedure and data segments in domains. In this case, the presence in a domain of a capability naming a segment would permit full read, write, and execute access to the segment. Providing separate control on these three operations, however, is useful for several reasons. Separate control on reading and writing encourages the sharing of procedure and data segments among several domains by making it possible to share only the ability to read a segment. Separate control on executing a segment allows the protection mechanisms of the processor to detect accidental attempts to execute segments that do not contain machine code.

expressed as:

(s,m)

where s is a segment number and m is some combination of read, write, and execute permission flags. Only the combinations r, rw, re, rwe, and null, corresponding to segments of read only data, read/write data, pure procedure, impure procedure, and to no access at all*, respectively, make much sense.

The fact that the names in these capabilities are segment numbers leads to an especially convenient representation of the domains associated with a process. Each domain is represented as an array whose elements are the modes defining the allowed access to the various segments in the virtual memory of the process. The array is indexed by segment number. (The usefulness of a null mode now becomes apparent.) The processor uses the mode array corresponding to the domain of execution of a process to control virtual memory references. An attempt by an executing machine language program to read, write, or execute segment number n is allowed by the processor only if the corresponding flag appears in the nth mode in this array.

Domain of Execution

The issue of how the domain of execution of a process is determined now must be considered. This problem is approached by defining ways to control changing the domain of execution. An understanding of the sort

---

*Including in a domain a capability specifying null access to a segment is equivalent to having no capability at all for that segment in the domain.

of control required can be gained by reviewing the purpose of associating multiple domains with a process. A domain provides the means to protect procedure and data segments from other procedures that execute as part of the same process. Using domains it is possible to make certain capabilities available only when particular procedure segments are being executed. The code sequences in these segments therefore determine the use made of those capabilities. These code sequences implement the intended algorithms, however, only if execution starts at certain points. Thus, for domains to be meaningful, it must be possible to restrict the start of execution in a particular domain to certain program locations. These locations are called gates. Changing the domain of execution can occur only as the result of a transfer to a gate location of another domain.

Controlling the ability to change the domain of execution, then, requires devising capabilities that identify certain procedure segment locations as gates into particular domains. The name in such a capability needs to be more precise than just a segment number. It must identify a particular machine instruction in a segment. The mode must indicate that the named location is a gate and specify the name of the domain for which it is a gate. Thus, the gate capability has the form:

((s,d),(gate into domain D))

where s and d name a virtual memory location and the mode indicates permission to invoke the program section beginning at the named location and simultaneously change the domain of execution to domain D. The method of naming domains is ignored for now, although it is apparent from the previous discussion that any domain naming scheme that allows

the processor to find the mode array for the new domain of execution will work.

Unfortunately, the gate capability just described does not fit very well with the previously suggested mode array representation of domains. Moving d from the name portion to the mode portion of the capability, however, solves this problem, for the name that is left is just a segment number as before. Capabilities for all the gates in a single segment that are accessible from some domain then may be represented by listing the displacements of these gates in the mode array element for that segment. Less mode information is required if all these gate locations in the same segment are required to be gates into the same domain, a reasonable restriction. Representing capabilities for gates, then, is done by adding a gate flag to the other three flags already possible in mode array entries. If the gate flag is present in a particular entry, then a list of displacements of valid gate locations within the corresponding segment is also part of the mode information, along with the name of the domain for which these locations are gates.

An Example of Mode Arrays

Figure 2-1 illustrates mode arrays defining three domains associated with some process. Segments 0 and 1 are read/write data segments accessible only from domain A. Segments 2 and 3 are pure procedure segments executable only in domain A. A program executing in domain A can change the domain of execution to B by transferring control to location 64 in segment 4, and to C through location 32 in segment 8. Domains B and C are similar. Note that segment 6 is pure procedure executable in

seg#	mode array for domain A	mode array for domain B	mode array for domain C
0	rw	null	null
1	rw	null	null
2	re	null	null
3	re	null	null
4	g(64:B)	re	null
5	null	rw	null
6	null	re	re
7	null	rw	r
8	g(32:C)	g(32:C)	re
9	null	null	rw
.			
.			
.			

Figure 2-1: Example access mode arrays for three domains of a process.

both domains B and C, and that segment 7 is a data segment that can be read and written from domain B but only read from domain C.

Cross-Domain Calls

As discussed in Chapter 1, the criterion of programming generality leads to a pattern in which the domain of execution of a computation is changed when one program invokes another that requires execution in a different domain. Adoption of this pattern means that the gate capabilities just described will normally correspond to program entry points and that a transfer of control to one of these gate locations will normally be part of an interprocedure call*. Such a call, however, involves several important functions in addition to the transfer of control to an entry point, all of which must be handled properly if a cross-domain call is not to compromise the protection provided by either of the involved domains. Some of these additional functions lead to the need for the processor to recognize, manipulate, and enforce capabilities with different dynamic properties than the encapsulation capabilities just presented, if the processor is to perform automatically cross-domain calls. These new kinds of capabilities control the access

---

*A procedure segment may actually contain a set of hierarchically nested procedures permanently bound together by the compiler. An interprocedure call is the call used to invoke an entry point into the outermost level of such a nested set of procedures. With the exception of one special circumstance discussed later, entry points into the inner layers of such a nest of procedures are not intended to be invoked from independent, separately compiled procedures, and thus will not be gate locations. The calling methods used within a procedure segment to invoke the inner entry points are of no concern to the system or its protection mechanisms.

of programs executing in the called domain to the arguments passed with the call and to the return point. They must be added to the called domain when a cross-domain call occurs and removed when the matching return occurs.

As a basis from which to develop the properties of argument and return capabilities, the action of an interprocedure call is considered in more detail. An interprocedure call and return, as defined in a language like PL/I, can be broken into eight steps. The call operation itself involves:

1. specifying the arguments to be passed to the called procedure;
2. saving the current procedure activation record* so that it may be restored when the subsequent return occurs;
3. specifying the program location to receive control when the subsequent return occurs;
4. transferring control to the called entry point;
5. creating a new activation record for the called procedure.

Once invoked, the called procedure will occasionally reference the arguments provided by the calling procedure. Eventually, a return of control to the calling procedure may occur. This return operation involves:

6. destroying the activation record of the called procedure;
7. transferring control back to the return location that was specified by the calling procedure;
8. restoring to use the activation record of the calling procedure.

---

*The activation record, normally allocated in a push-down stack, is the local addressing environment for a procedure invocation and provides storage for information associated with a single invocation of a procedure. Addresses of arguments, the address of the return point, and PL/I "automatic" variables are examples of this kind of information.

The additional requirement of calls that change the domain of execution is that all of these steps take place without compromising the protection provided by either of the domains involved.

A cross-domain call occurs if the virtual memory location to which control is transferred in step 4 happens to be defined by a capability in the domain of execution (call it domain A) as a gate into another domain (call it domain B). In this case the return point specified in step 3 and transferred to in step 7 is really a gate back into domain A, for a transfer to this location by a procedure executing in domain B should change the domain of execution back to A. As seen earlier, the ability to transfer control to a gate and change the domain of execution can be controlled by a capability. The capability for this return point cannot be a permanent part of domain B. It if were, then procedures executing in domain B could use it whether the corresponding call from domain A were outstanding or not. What is required is that the capability for the return gate be added to domain B when the call occurs, and removed from domain B when this capability is used to make the return.

A sequence of cross-domain calls may occur in a process before any of them are returned. In the general case, that sequence may include the recursive invocation of some domains of the process. The proper dynamic behavior of the capabilities for return gates associated with such a sequence is achieved by associating with a process a push-down stack for return gate capabilities. Each time a cross-domain call occurs, the processor adds the associated return gate capability to the top of this stack, pushing-down whatever other return gate capabilities

may already be in the stack. At any given time, only the return gate capability at the top of the stack is available. When this capability is used to perform a cross-domain return, the processor removes it from the top of the stack, thereby making available the next older return gate capability. Thus, at any given time, the only cross-domain return that can be performed is defined by the address and domain name contained in the gate capability at the top of the stack. This stack, then, guarantees that the returns matching the sequence of outstanding cross-domain calls in the process will occur in the correct order, and that each such return will transfer control to the expected location and will properly change the domain of execution. This guarantee is valid even if more than one cross-domain call into some domain is included in the sequence, for procedures executing in one invocation of this domain in the sequence are not able to use the return capabilities associated with other invocations.

The capability for the return gate of a cross-domain call is really a special case of a capability for an argument passed with a cross-domain call. Capabilities to reference whatever arguments are specified in step 1 of the call operation must be added to the called domain when the call occurs, and must be removed when the return transfer of control in step 7 occurs. The stack for return gate capabilities can be expanded to control the availability of argument capabilities as well. Thus, when a cross-domain call occurs, the processor allocates a <u>stack frame</u> at the top of this stack, pushing-down whatever stack frames may already exist. Into this frame the processor places a capability for the return gate associated with the call and capabilities allowing reference to whatever

arguments are specified in step 1 of the call operation. Only the capabilities in the top frame on the stack are available at any given time. The top frame is removed, and the remaining frames popped-up, when a cross-domain return occurs.

Static Access and Dynamic Access

With the addition of the stack for argument and return capabilities, the capabilities that define a domain of a process are represented in two different ways. First, there is the mode array of the domain that specifies the encapsulation capabilities of the domain. In addition, whenever a particular domain is the domain of execution of the process, the top frame on the capability stack specifies the argument capabilities (including the capability for the return gate) that are temporarily part of the domain. To differentiate these two sources of capabilities for a domain, access available from the encapsulation capabilities to procedures executing in a domain is called static access, and that available from the argument capabilities is called dynamic access. The capability stack of a process is called the dynamic access stack.* Much of the rest of this thesis will be concerned with the difference between static and dynamic access, so the reader is advised to pause and be sure he understands the definitions of these concepts.

---

*The dynamic access stack is distinct from, though related to, the stack of activation records for procedure invocations.

Dynamic Access Capabilities

As stated above, dynamic access capabilities are created by the processor whenever a cross-domain call occurs. Their purpose is to give procedures executing in the called domain access to the arguments and the return gate associated with the call. In creating these capabilities, the processor must depend upon the specifications of arguments and the return point provided by the calling procedure. Since there is no control on the specifications which may be provided, the processor must make sure that the capabilities created in this way and added to the called domain do not give procedures executing in the called domain permission to reference the virtual memory of the process in ways that procedures executing in the calling domain cannot. Without this check, the protection mechanism of the processor could be circumvented easily by a pair of conspiring procedures in two different domains calling one another and specifying the creation of dynamic access capabilities to reference all parts of the virtual memory of the process.

If dynamic access capabilities are to control properly access to arguments, they must provide more precise control than the static access capabilities discussed earlier. In general, not all arguments will be whole segments of information. For example, if the entire activation record for a procedure invocation is stored within one segment, then a single automatic variable associated with the procedure may occupy only a few bits of that segment. It is certainly reasonable to specify such a variable as an argument. If dynamic access capabilities are to allow procedures executing in the called domain access to just the arguments passed with the cross-domain call, and nothing else, then these

capabilities must be able to name arbitrary subsegments, not just whole segments as do static access capabilities. The mode should provide separate control on reading and writing the subsegment named in a dynamic access capability, so the intention that the value of a particular argument not be altered can be enforced. Thus, the dynamic access capabilities recognized and enforced by the processor have the form:

$$(\underline{s},\underline{d},\underline{\ell},\underline{m})$$

where the first three elements name a subsegment and the last element is some combination of read and write permission flags.

The dynamic access capabilities for return gates need to contain the same information as static access capabilities for entry point gates. The name portion of a return gate capability should specify the two-part address of the return location, and the mode should indicate that the capability is for a return gate and should specify the corresponding domain of execution. It is very convenient, however, if the mode also includes one additional piece of information -- the two-part address of the activation record that is to be restored to use when the cross-domain return occurs. The calling procedure needs to be guaranteed that the correct activation record is restored to use when a cross-domain return occurs, as well as be assured that the return transfers control to the expected location and changes the domain of execution to the proper domain. If procedures executing in the called domain could somehow cause a different activation record to be restored to use, then the calling domain could be caused to malfunction after the return occurred. As will be seen in Chapter 3, the activation record of an executing procedure is defined by the two-part address in a particular addressing

register of the processor. Using the extra information in the return gate capability, the processor can verify that this register contains the correct value before performing a cross-domain return. Thus, the dynamic access capability for a return gate really gives permission to perform a very specific three-part operation: load the activation record definition register with a particular value, transfer control to a particular location, and change the domain of execution to a particular domain. As will be seen in Chapter 3, the return gate capability may be extended to control call and goto operations to entry points and labels, respectively, that are passed as arguments with cross-domain calls.

Matching References to Capabilities

A virtual memory reference is specified by a two-part address (s,d), an implied length ℓ, and an operation. Validation of such a reference requires matching it with some capability in the domain of execution.

Matching a reference to a static access capability is quite straightforward. The segment number s is used as an index into the mode array for the domain of execution. The read, write, execute, and gate permission flags in the selected entry indicate in the manner discussed earlier whether or not the reference is allowed, depending upon what operation is being attempted.

Matching a reference to a dynamic access capability is a little more complicated. Since more than one dynamic access capability may cover a particular segment, an associative search of the available

dynamic access capabilities is needed. What is required to allow the reference is the existence of at least one dynamic access capability covering the referenced subsegment and allowing the operation being attempted.

It is up to the procedure in execution to determine which of these two kinds of validation will be applied to each virtual memory reference. The processor never forces the use of one or the other validation method. On the other hand, it is important to maintaining the integrity of a domain to choose the proper method of access validation for each reference. Validation relative to the dynamic access capabilities in a domain should be used if the reference represents an attempt to access an argument provided with a cross-domain call. Otherwise, the reference should be validated relative to the static access capabilities in the domain. The reasons for this pattern of validation will be discussed in a moment.

The mechanisms of the processor for forming and manipulating two-part addresses make it natural to use the two methods of access validation in the manner just suggested. All two-part addresses that are derived from the argument specifications associated with the cross-domain call that started execution in a domain are automatically tagged by the processor. A reference using such a tagged address triggers validation relative to the dynamic access capabilities of the domain of execution. References with untagged addresses are validated relative to the static access capabilities of the domain. Unless an executing procedure makes an explicit effort to alter these tags, which it can do, the processor always manages to identify correctly addresses which were derived from

cross-domain call argument specifications, and perform the proper validation on references made using such addresses.

The reason to be careful with argument-related addresses is that they are derived from information provided by a procedure executing in another domain -- the domain which generated the cross-domain call. In general, there is no guarantee that they locate pieces of the virtual memory that are accessible from the calling domain*. For example, the calling procedure could maliciously or erroneously provide an argument-related address that in fact names some location in a critical data base encapsulated in the called domain -- a data base that is not accessible from the calling domain. If a procedure executing in the called domain writes through this address, it does so with the intention of changing the value of an output argument. If this reference were validated relative to the static access capabilities of the called domain, it would succeed, erroneously changing the critical data base. Validated relative to the dynamic access capabilities of the domain, however, the reference fails, for as seen earlier the dynamic access capabilities cannot give access permission that is not also available to the calling domain. Thus, dynamic access capabilities allow the called domain to protect itself against "trick" argument-related addresses provided by the calling procedure.

---

*While it is true that some of the addresses that can be derived from argument specifications will have been checked by the processor when used to create the dynamic access capabilities corresponding to cross-domain arguments, as discussed earlier, not all such addresses will have been so checked. Further, even though the capabilities cannot be altered once created, the specifications might be changed after being checked, and it is the specifications from which the called domain generates argument addresses. This matter is discussed further in Chapter 3.

Creation of a New Activation Record

One function that must be performed by a cross-domain call has not been discussed yet -- creating a new activation record for the called procedure. When a call changes the domain of execution of a process, allocation of a new activation record for the called procedure must take place without any possibility of interference from the calling procedure. Therefore, the called procedure cannot depend upon any information provided by the calling procedure in order to find space for a new activation record.

Activation records are normally allocated in a push-down stack. To prevent procedures executing in one domain from directly reading and writing the activation records (or their residues) of procedures that execute in other domains, this push-down stack will be spread over several segments -- one for each domain of a process. Static access capabilities in the various domains are usually arranged so that the stack segment for a particular domain can be read and written from that domain, but is not accessible from other domains except those encapsulating portions of the supervisor.

With this scheme, allocating a new activation record for the called procedure involves finding the stack segment for the domain of execution of the called procedure and then locating the beginning of the free area at the end of this segment. The processor makes this operation easy and safe by leaving in a particular addressing register after each call, cross-domain or otherwise, the two-part address of the beginning of the proper stack segment for the new activation record. From this address the called procedure can easily locate the free area at the end of this

segment, and allocate its own activation record, without having to depend on information provided by the calling procedure. This scheme requires that there be embedded in the processor some algorithm for generating the segment number of the proper stack segment given the name of the domain of execution.

Summary

A processor which supports a multidomain computation implemented as a single execution point in a segmented virtual memory, and automatically performs cross-domain calls, must recognize and enforce two distinct kinds of capabilities -- static access capabilities representing the encapsulation of subsystems in domains and dynamic access capabilities representing cross-domain arguments. These two kinds of capabilities have different dynamic properties.

All functions of an interprocedure call have protection implications when such a call is allowed to change the domain of execution. The static access capability for controlling access to entry point gates, the various dynamic access capabilities, and processor assistance in creating a new activation record for the called procedure are sufficient tools for performing these functions without compromising the protection provided by either domain involved in a cross-domain call.

In the next chapter these ideas are expanded into the design of a processor meeting the objectives outlined in the first chapter. That processor design will enable many of the ideas introduced in fairly general terms here to be presented in specific forms which make their validity and usefulness more apparent.

CHAPTER 3

THE PROCESSOR DESIGN

This chapter describes a hardware processor that supports a multi-domain computation implemented as a single execution point in a segmented virtual memory and that automatically performs cross-domain calls. This processor provides a specific context in which to explore some of the detailed implications of the objectives presented in Chapter 1 and in which to expand the ideas that were introduced in Chapter 2.

The processor described here is a paper machine. No realization of this design exists. It is claimed, however, that the processor described here could be built economically using today's hardware technology and that, if built, it would provide a practical hardware base for a computer utility. This claim is based on the facts that a processor with similar overall architecture but less sophisticated protection mechanisms is currently being built by Honeywell Information Systems, Inc. to serve as the primary component of a new hardware base for Multics [25] and that the new protection mechanisms described here add little to the overall complexity of that machine.

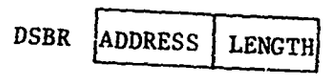
In the case of many of the component mechanisms in this processor, the specific implementation described represents a choice among several alternative implementations that all meet the basic functional requirements of the component. An important criterion in making such choices was the functional clarity of the various alternatives. In each case the implementation chosen is intended to expose the intrinsic problems

that are solved by a component of the protection mechanisms while at the same time illustrating that a practical, general implementation of that component is possible. For example, the method used to implement static access capabilities places a fairly small fixed upper limit on the number of domains that can be associated with a single process at any one time. This implementation is easy to understand and quite economic if a small number of fairly large domains are usually associated with a process. An alternative implementation that allows an essentially unlimited number of domains to be associated with a process was rejected because it introduced complexity that obscured the intrinsic problems being solved by this component of the protection mechanisms in the processor. The specific design presented here, then, should be considered as one example of how to organize a processor to meet the stated objectives. After understanding how this processor works, the inventive reader should be able to devise a large number of variations on this design which embody the same basic techniques but which may meet better the specific external constraints associated with a particular application of such a machine.

The Virtual Memory and Static Access Capabilities

As in Multics, the implementation of the segmented virtual memory is based on a descriptor segment that is stored in memory. Figure 3-1 illustrates the format of a descriptor segment entry (DSE). If the validity bit (DSE.V) is on, then a DSE contains the absolute address of the beginning of a segment in memory (DSE.ADDRESS) and the length of of that segment (DSE.LENGTH). The descriptor segment base register (DSBR)

Descriptor segment base register



Domain of execution register

Descriptor segment (stored in memory)

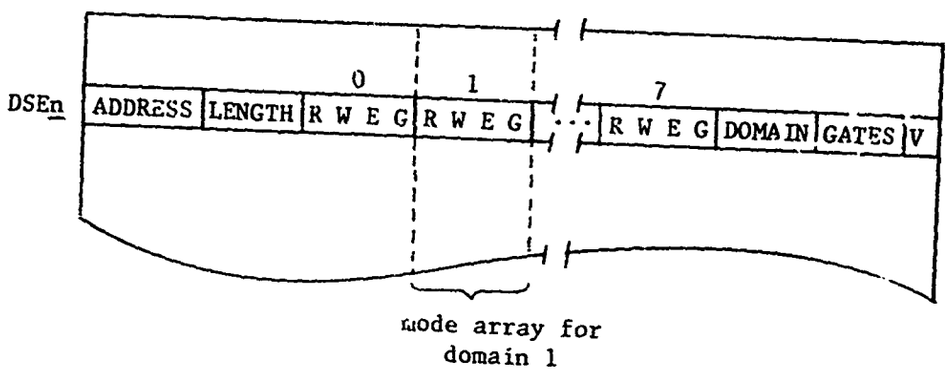


Figure 3-1: The descriptor segment and related registers.

of the processor contains the base address (DSBR.ADDRESS) and length (DSBR.LENGTH) of the descriptor segment, allowing the address translation logic of the processor to reference DSE's by absolute address. Automatic translation of a two-part address into the corresponding absolute address, done by using the segment number as an index with which to retrieve the appropriate DSE*, happens each time that the virtual memory is referenced, i.e., each time an instruction, indirect address, or instruction operand is referenced by the executing program. If each process is to have its own virtual memory, as in Multics, then each process has its own descriptor segment. If several processes are to share a single virtual memory, then those processes have a common descriptor segment.

As implied by this description, it is assumed that the storage space for each segment is contiguously allocated in a one-level memory system with enough capacity to hold all segments of interest. With most real implementations of segmented virtual memories, storage for segments is provided by a multilevel memory system using block allocation and paging. If used, block allocation and paging must be taken into account by the address translation logic of the processor. When properly implemented, however, these storage management techniques are functionally transparent from the point of view of protection mechanisms. The assumption of a one-level memory system and contiguous allocation of space for segments allows problems of storage management to be ignored.

---

*Including in the processor a small, fast associative memory for the most recently used DSE's can eliminate most of the memory references for DSE's [24].

As indicated in Chapter 2, the static access capabilities included in the domains associated with a process can be represented as mode arrays, one per domain. Mode arrays are indexed by segment number, as is the descriptor segment. This suggests the straightforward implementation for static access capabilities of combining the mode arrays of a process with the descriptor segment. The DSE format in Figure 3-1 illustrates this implementation. Each DSE contains a set of read, write, execute, and gate permission flags for each domain that can be supported by the processor. The domain of execution register (DR) records the number of the current domain of execution of the process. The value in DR is used to select the proper set of permission flags with which to validate virtual memory references.

The DSE for a segment is an ideal place in which to record the static access capabilities controlling references to that segment. Since the processor must examine the DSE for a segment each time that segment is referenced anyway, little effort is added to validate an attempted access against static access capabilities recorded there.* With this implementation of static access capabilities, changes in the static access of a domain can be made immediately effective, since access is validated each time a reference is made.

A key engineering decision is the number of domains that will be supported automatically by the processor. The size of a DSE increases as the number of domains to be supported increases. Based on experience

---

*If an associative memory for DSE's is used, the static access capabilities for a segment will automatically be in the associative memory whenever the corresponding DSE is there.

with Multics, the number chosen here is eight. Thus, there are 32 permission flags in each DSE. This choice is based upon the assumption that domains normally encapsulate relatively large subsystems and, therefore, that a few domains per process is sufficient.*

Whenever DR contains the value n, the nth set of flags in the DSE's control access to the virtual memory of the process. An attempt to read or write segment number s will be allowed only if DSEs.Rn or DSEs.Wn is set on, respectively. An attempt to transfer control to location (s,d) has three possible results. If both DSEs.En and DSEs.Gn are off, then the transfer is not allowed. If DSEs.En is on, then the transfer is allowed and execution continues in the same domain. If DSEs.Gn is on, then the segment contains gates into another domain. The displacements of the gate locations are recorded in DSE.GATES, and the name of the domain is recorded in DSE.DOMAIN. For ease of representation, all gate locations in a segment are collected together at the beginning of the segment. DSE.GATES indicates the number of such gate locations. Thus, d < DSEs.GATES guarantees that the transfer is to a gate location. If this check is true, then DR is changed to the value recorded in DSEs.DOMAIN, thus changing the domain of execution of the process, and the transfer is allowed. Having both DSEs.En and DSEs.Gn set on is not meaningful.

---

*An alternative implementation of static access capabilities is to place the mode array for each domain in a separate segment. This implementation removes the small fixed upper limit on the number of domains that may be associated with a process at any one time, but complicates the naming of domains, the representation of gates, and the structure of the associative memory for DSE's.

Best Available Copy

By hardware and software convention, domain 0 is always used to encapsulate the most privileged portion of the supervisor. Instructions designated as privileged will be executed by the processor only in domain 0. The instruction to load DSBR is an example of an instruction that must be privileged in order to maintain the integrity of the protection mechanisms in the system. The seven remaining domain numbers available to a process are managed by the supervisor in much the same way that the larger number of available segment numbers are managed. When a program executing as part of a process first wants to reference some segment stored on-line, the segment is assigned an available segment number in the virtual memory of the process by the supervisor. Likewise, when a program first wishes to associate with the process some protected subsystem whose definition is stored on-line, that protected subsystem is assigned an available domain number by the supervisor. Thus, while the maximum number of protected subsystems may be associated with a process at one time is eight, the particular eight can change with time, and different sets of eight or fewer protected subsystems can be associated with processes that have different virtual memories.

Generating Virtual Memory Addresses

In order to understand the discussion to be presented in the following sections of this chapter, it is necessary to know something about the generation of two-part addresses by machine language programs. For ease of description, the processor presented here is primarily a word addressed machine. Thus, the displacement portion of a two-part address is a word number, where a word is enough bits to hold an indirect

address or an instruction, and the implied length of most references is one word. The ideas presented here, however, can also be applied easily to a byte or bit addressed machine. The ultimate precision of referencing memory by arbitrary bit number and arbitrary bit length can be very useful in controlling access to cross-domain arguments that are not word aligned, and will be considered in that respect later.

A two-part address together with a protection tag is called a pointer. The tag portion of a pointer is a one bit flag* that indicates whether the static access capabilities or the dynamic access capabilities of the domain of execution should be used to validate references through the pointer. The tag will be considered in more detail in a moment. Figure 3-2 presents the registers and storage formats relevant to manipulating pointers. The instruction pointer register (IPR) specifies the two-part address of the instruction being executed. Since instruction retrieval is always validated relative to the static access capabilities of the domain of execution, the pointer in IPR does not include a tag. Arbitrary pointers may be kept in the pointer registers (PR1, PR2, ...) or stored in memory as indirect addresses (IND). Because segment numbers cannot be known when a procedure segment is compiled and because most procedure segments will be pure, instructions (INST) specify operand addresses by giving an offset (INST.OFFSET) relative to IPR or one of the PR's. INST.PRNUM = 0 indicates IPR-relative addressing and INST.PRNUM > 0 indicates PR-relative addressing. In the latter case

---

*The one bit tag is a simplification used now to facilitate the description of the processor. The multibit tag actually required will be introduced later.

Instruction pointer register

IPR	SEGNO	WORDNO

Pointer registers

PR1	TAG	SEGNO	WORDNO
PR2			
.			
.			
.			

Indirect address (stored in memory)

IND	TAG	SEGNO	WORDNO

Instruction (stored in memory)

INST	PRNUM	OFFSET	OPCODE	I

Temporary pointer register

TPR	TAG	SEGNO	WORDNO

Figure 3-2: Register and storage formats for address formation.

the value of the field indicates which PR. Indirect addressing may be specified by setting the indirect flag (INST.I) on. The final item in Figure 3-2 is the temporary pointer register (TPR). This is an internal processor register that is not directly program accessible. In TPR is formed the two-part address, called the effective pointer, of the operand of each instruction.

The algorithm for generating the effective pointer, given an instruction, is presented in Figure 3-3. Ignoring tags in pointers for a moment, a couple of examples will make clear the operation of this algorithm. The instruction:

PRNUM	OFFSET	OPCODE	I
0	200	ADD	off

specifies that its operand is the word 200 beyond the instruction in the same segment. The instruction:

1	200	ADD	off

specifies that its operand is the word 200 beyond the location specified by the pointer in PR1. The instruction:

1	200	ADD	on

causes the word 200 beyond that specified by the pointer in PR1 to be retrieved from memory and interpreted as an indirect address which specifies the two-part address of the instruction's operand. The instructions from these three examples may be written as:

```
ADD     ↑200
ADD     PR1↑200
ADD     PR1↑200,*
```

respectively.

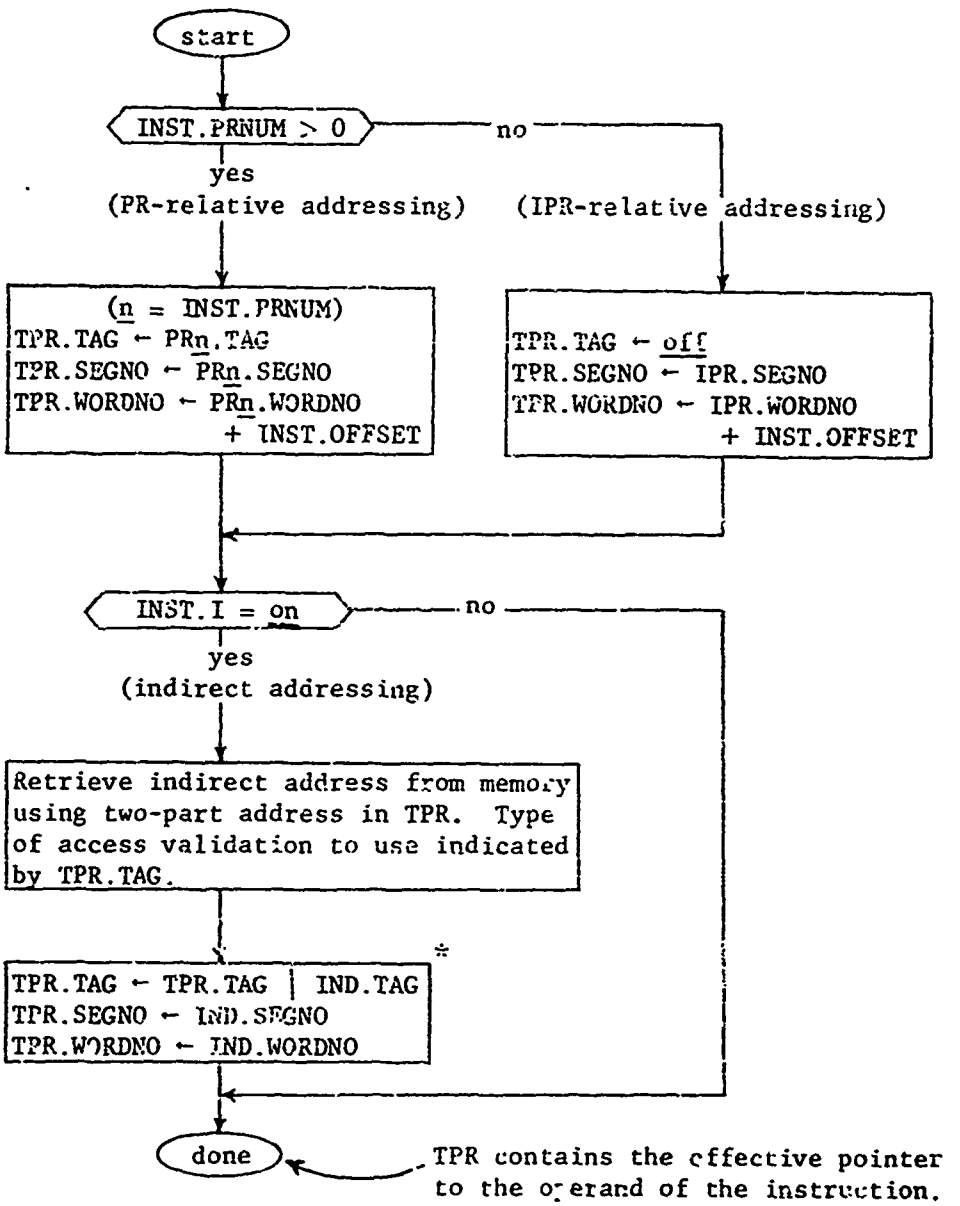


Figure 3-3: Algorithm for effective pointer generation.

---

*The symbol "|" indicates the logical OR operation.

Obviously, for this machine really to be practical a richer collection of addressing techniques would be needed. The processor as described does not even have index registers! The techniques shown, however, are sufficient to discuss the protection issues raised in this thesis and to illustrate the rules that other addressing techniques must follow. With respect to protection mechanisms, it is straightforward to add other more sophisticated addressing techniques to the processor.

Two machine instructions are representative of those used to manipulate pointers. The "effective pointer to pointer register n" instruction (EPPn) causes the effective pointer generated by the instruction, i.e., the entire contents of TPR, to replace the contents of PRn. "Store pointer from pointer register n" (SPPn) causes the entire contents of PRn to be stored at the specified location in the format of an indirect address. As an example of how these two instructions can be used, note that the following code sequence saves the pointer from PR4 and then restores it to PR4:

```
SPP4    PR1↑367
EPP4    PR1↑367,*
```

The role of the tags in pointers is now discussed in more detail. The tag differentiates pointers related to cross-domain arguments from other pointers. When the tag in a pointer is off it means that a reference through that pointer will be validated relative to the static access capabilities of the domain of execution. When the tag is on the dynamic access capabilities of the domain of execution will be used instead.

The processor is organized so that, as a matter of course, the tag is on in all pointers related to cross-domain arguments. The key to this feature is the effective pointer generation algorithm presented in Figure 3-3. It is the tag in TPR that determines which type of access validation will be applied to each indirect address reference and each instruction operand reference. As can be seen from Figure 3-3, the tag in TPR will be on at the conclusion of effective pointer generation for some instruction if the operand address specification in the instruction indicates a PR whose tag is on or locates an indirect address whose tag is on or both. A reference to an indirect address, if required, will be validated as indicated by the value of TPR.TAG at the point in the algorithm where the indirect address is retrieved from memory.

The technique described in the next section for transmitting arguments on a call is to prepare a list of indirect addresses locating the various arguments and then to communicate the location of this list to the called procedure via a PR. In the case of a cross-domain call the processor automatically sets the tag in this PR on. Since the called procedure normally will derive the addresses of the arguments through this PR using the effective pointer generation algorithm just described, all argument-related addresses will acquire a tag that is on as a matter of course, and therefore all references to cross-domain arguments will be validated relative to the dynamic access capabilities of the called domain as is desired. Referencing cross-domain arguments will be discussed in more detail later.

Call and Return

The next step in describing the processor is to introduce the implementation of the standard interprocedure call and return. This call and return convention provides the standard interface for separately compiled procedures in the system so that they may be used as building blocks in the manner discussed in Chapter 1. For this initial description protection issues are ignored.

According to Chapter 2, the first step of an interprocedure call is specifying the arguments to be passed to the called procedure. Following the custom in many programming languages, the procedure to be called is assumed to know the number of arguments to be passed, the order in which they will be specified, and the type and size of each.* Since arguments are to be passed by address, the minimum information which must be communicated to the called procedure is a list containing a pointer to the start of each argument. To communicate this list a specific pointer register chosen by system convention (call it PR*a*) is set to point to an area of memory that is to hold the list. This area is allocated in the activation record of the calling procedure. The area is then filled according to the format given in Figure 3-4 with pointers to the various arguments in the appropriate order. The pointer to the *i*th argument normally is constructed using a code sequence such as:

```
EPPn    "argument i"
SPPn    PRa|(2i+3)
```

*These assumptions can be removed by providing a more elaborate argument list than is defined here. The simple argument list presented is adequate to illustrate the relevant protection issues.

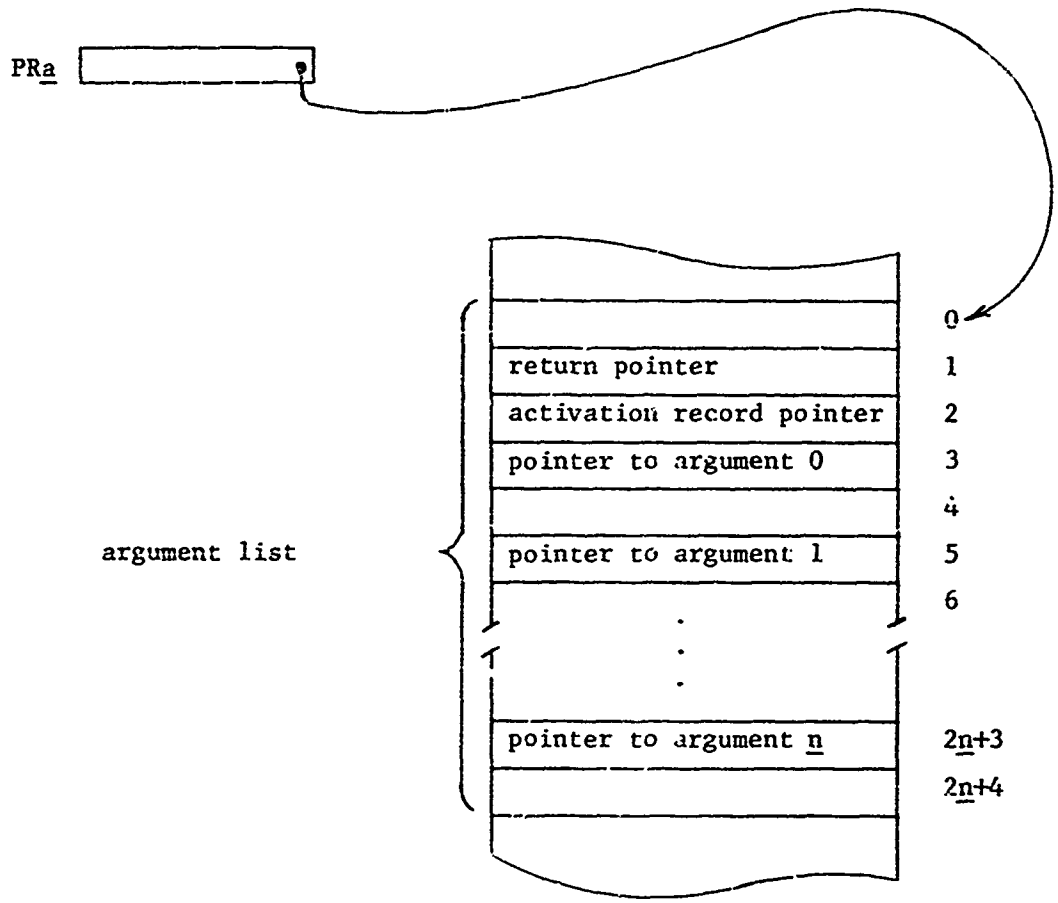


Figure 3-4: Format of the argument list for an interprocedure call.

where n indicates some PR that is temporarily free and "argument i" represents an operand address specification locating the ith argument.

The next two steps of a call are saving a pointer to the current activation record and specifying the return location. The beginning of the memory area holding the activation record in use is specified by another pointer register chosen by system convention (call it PRs). The return location is always one word beyond the instruction that transfers control to the called procedure. The instructions:

```
SPPs    PRa↑2
EPPn    ↑3
SPPn    PRa↑1
```

where n designates some unused PR, create the return block at locations 1 and 2 of the argument list as shown in figure 3-4 by storing as indirect addresses the pointer from PRs and the pointer from IPR (suitably offset).

Creating the argument list also includes storing additional argument-related information in the empty words shown in Figure 3-4. This extra information is used by the processor in the case of a cross-domain call to create the proper dynamic access capabilities. The definition of these remaining parts of the argument list is delayed until the next section.

Once the argument list is created, the next step of the call is actually performing the transfer of control to the called entry point. This is done by executing the instruction:

```
CALL    "entry point"
```

where "entry point" is an operand address specification that locates the

entry point being called. In addition to transferring control, the CALL instruction helps the called procedure create a new activation record for itself by leaving in a third pointer register chosen by system convention (call it PR*b*) a pointer to the zeroth word of the stack segment in which the called procedure should allocate its activation record. This stack segment has in its zeroth word a pointer to the beginning of the unused area at the end of the stack segment. The called procedure creates a new activation record of *ℓ* words there by executing the instructions:

```
EPPs    PRb↑0,*
EPPn    PRs↑ℓ
SPPn    PRb↑0
```

where *n* designates some PR other than one of the three reserved pointer registers. The first instruction loads PR*s* with a pointer to the beginning of the new activation record. The second and third instructions update the pointer in the base of the stack segment to point just beyond this new activation record.

Creating a new activation record completes the steps of the call operation. While executing, the called procedure may reference words in its activation record using operand address specifications of the form "PR*s*↑*i*" in instructions. It can generate in PR*n* the beginning address of the ith argument using the instruction:

```
EPPn    PRa↑(2i+3),*
```

When it comes time to return, the called procedure must perform the three steps of the return operation. First, its activation record must be released and added to the unused area at the end of the stack segment.

This is done with the single instruction:

SPP<u>s</u> PR<u>b</u>↑0

which resets the pointer in the base of the stack segment to point at the beginning of the released activation record. The second and third steps of the return operation are transferring control back to the calling procedure and restoring to use the activation record of the calling procedure. These steps are performed in reverse order by the following instruction sequence:

EPP<u>s</u> PR<u>a</u>↑2,*

RETURN PR<u>a</u>↑1,*

The first instruction restores to PR<u>s</u> the activation record pointer from the return block in the argument list. The second instruction* transfers control to the location specified by the return pointer from the return block. Thus, the calling procedure receives control at the proper return point and the proper activation record is restored to use.

Note that the only PR which is restored by the return operation to the value it contained prior to the call is PR<u>s</u>. It is up to the calling procedure to explicitly save and restore other PR's whose contents are valuable. Because PR<u>s</u> is automatically restored by the return, the activation record can be used by the calling procedure as a place in which to save the values of other PR's before the call, from which place they may be restored after the return.

---

*Ignoring protection issues, the RETURN instruction simply transfers control to the location specified as its operand. As will be seen in the next section, however, the RETURN instruction can perform some protection-related operations that normal transfer instructions cannot.

Cross-Domain Call and Return

The standard interprocedure call and return just sketched are now reconsidered in light of the requirement that they be able to change the domain of execution of the process. The overall strategy of implementing cross-domain calls and returns is based on the following two hardware enforced restrictions:

1. Only the CALL instruction may utilize a static access capability for an entry point gate.

2. Only the RETURN instruction may utilize a dynamic access capability for a return gate.

The implications of these restrictions are that all cross-domain calls will be performed by the CALL instruction, and that all cross-domain returns will be performed by the RETURN instruction. (The processor is also arranged so that no other type of instruction may change the domain of execution of a process.) Thus, the special functions associated with cross-domain calls and returns can be embedded in the CALL and RETURN instructions, respectively, with the assurance that these functions will be performed whenever a cross-domain call or return occurs.

If a call to location (s,d) from domain n is to be cross-domain, then the CALL instruction detects this circumstance by noting that DSEs.Gn is on rather than DSEs.En.* (The call will be allowed if d < DSEs.GATES and the new domain of execution will be DSEs.DOMAIN.) Detection of a cross-domain call causes the CALL instruction to perform

---

*It is assumed for most of this chapter that the effective pointer generated by the CALL instruction always has a tag that is off, and thus that validation of the call is relative to the static access capabilities of the domain of execution. In one of the last sections of the chapter the meaning of removing this restriction is explored.

two extra operations that are not performed if the call does not change the domain of execution. The first is creating a new frame on the dynamic access stack of the process and filling this frame with the dynamic access capabilities which will allow procedures executing in the called domain to reference the arguments of the call and transfer control back to the return gate. The second extra operation is to set the tag in the argument list pointer register (PR<u>a</u>.TAG) to <u>on</u> so that references to the cross-domain arguments by procedures executing in the called domain will be validated with respect to these dynamic access capabilities, rather than the static access capabilities of the called domain. After performing these two extra steps in the case of a cross-domain call, the CALL instruction completes in the normal manner by loading PR<u>b</u> with a pointer to the base of the stack segment in which the called procedure should allocate its new activation record, and then transferring control to the called location. The pointer in PR<u>b</u> is created the same way whether the call is cross-domann or not. The stack segment selection rule embedded in the processor is that the stack segment for domain <u>n</u> is segment number <u>n</u>.* Thus, PR<u>b</u>.SEGNO is

---

*This stack segment selection rule is adequate only for the case that each process has its own virtual memory. If it is desired to allow multiple processes with the same virtual memory, i.e., more than one execution point in a virtual memory, then a more sophisticated stack segment selection rule is required to provide each of the processes sharing a virtual memory its own set of stack segments. A simple way to provide for multiple sets of stack segments in a virtual memory is to include in the processor a register that specifies the eight consecutively numbered segments that are the stack segments. When a CALL instruction is executed, then PR<u>b</u>.SEGNO is calculated by adding the new domain number to the value in this stack segment selection register. The various processes sharing a virtual memory each would execute with a different value in this register, and thus each would

set to the number of the domain in which the called procedure will execute, PRb.WORDNO is set to zero, and PRb.TAG is set off (the called procedure will use static access to reference this stack segment). The transfer of control to the called procedure is done by loading IPR from the effective pointer generated by the CALL and, if necessary, reloading DR to change the domains of execution.

A cross-domain return is detected by noting that the tag is on in the effective pointer generated by the RETURN instruction. Detection of a cross-domain return causes the RETURN instruction to perform several extra operations that are not performed if the return does not change the domain of execution. The return gate capability in the newest frame on the dynamic access stack is used to verify that PRs contains the proper activation record pointer and that the transfer is to the proper location. If so, the RETURN instruction removes this frame from the dynamic access stack, thus destroying the dynamic access capabilities that allowed procedures executing in the called domain to reference the arguments of the corresponding cross-domain call and transfer control to the return gate. When the effective pointer generated by the RETURN instruction is loaded into IPR to perform the transfer, DR is changed back to the domain of execution of the calling procedure, as recorded in the return gate capability.

---

have its own set of stack segments. With this change the processor is fully able to support multiple processes sharing a single virtual memory. Each such process would have its own dynamic access stack but would use the same descriptor segment.

Dynamic Access Capabilities

In creating a new dynamic access stack frame for a cross-domain call, the CALL instruction uses the standard interprocedure argument list as the specification from which to generate dynamic access capabilities. The argument list defined in Figure 3-4, while it provides enough information for the called procedure to reference the arguments, does not provide quite enough for the processor to create the proper dynamic access capabilities. Spaces have been left in that format, however. for the additional information required to correct this deficiency. The complete argument list format, illustrated in Figure 3-5, uses a double word entry to describe each argument. This entry contains the size (in words) of the argument and flags indicating whether the argument can be read and/or written, in addition to the pointer to the beginning of the argument. To inform the processor how many dynamic access capabilities to generate, an argument list header is added that indicates the length (in words) of the list.

To be consistent with the criterion of programming generality, the full argument list as defined in Figure 3-5 must be generated by compiled code for all interprocedure calls, as it cannot be known in advance which calls will be cross-domain. Minor extensions to most languages will be required to allow symbolic programs to specify to the compiler which arguments of a call are input arguments and which are output arguments so that the read and write flags in the argument list entries can be properly set. Compilers must also be able to generate code which properly sets the size information in each argument list entry.

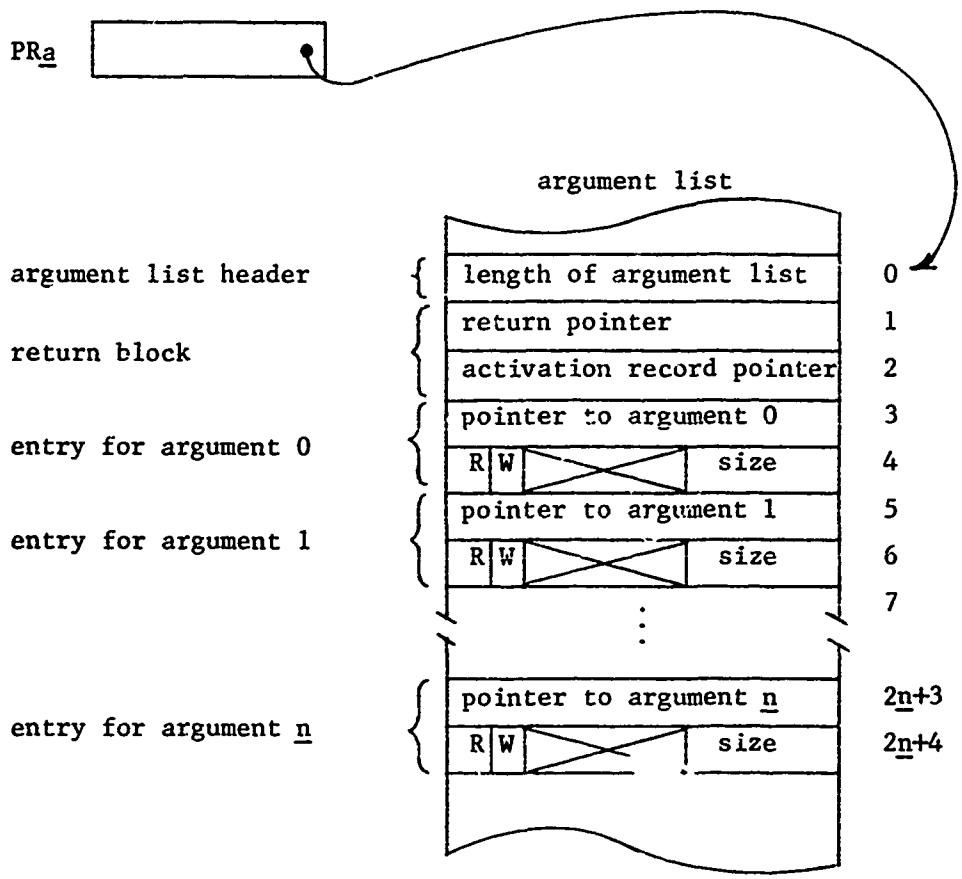


Figure 3-5: Completed argument list format for an interprocedure call.

The dynamic access stack (DAS) of a process, in which dynamic access capabilities are created by the processor on a cross-domain call, is located in memory -- presumably in a segment that can be read and written directly only from domain 0. The location and state of the DAS is specified by the DAS definition register (DASDR) shown in Figure 3-6. This register, in addition to being manipulated by the CALL and RETURN instructions in the case of cross-domain calls and returns, can be loaded and stored directly by privileged instructions. DASDR.SEGNO and DASDR.WORDNO1 specify the two-part address of the beginning of the most recently created DAS frame. DASDR.WORDNO2 specifies the word one beyond the last word in this frame. If no unreturned cross-domain calls exist, then DASDR.SEGNO and DASDR.WORDNO1 will specify the beginning location for the DAS of the process, and DASDR.WORDNO2 will equal DASDR.WORDNO1.

As can be seen by comparing Figures 3-5 and 3-6, the creation of a new DAS frame by the CALL instruction in the case of a cross-domain call is quite straightforward. PR<u>a</u> locates the argument list to be used as the information source and (DASDR.SEGNO,DASDR.WORDNO2) is the two-part address for the beginning of the new frame. While creating a new DAS frame the processor validates all references to the argument list relative to the static access capabilities of the domain from which the call is being made* and validates all references to the DAS relative to the static access capabilities of domain 0.

---

*According to the normal pattern of operation described earlier, an executing procedure has static access to its activation record and to any argument lists it creates for interprocedure calls. To simplify the processor description, the processor protection mechanisms are designed to accomodate only this normal pattern of operation. The

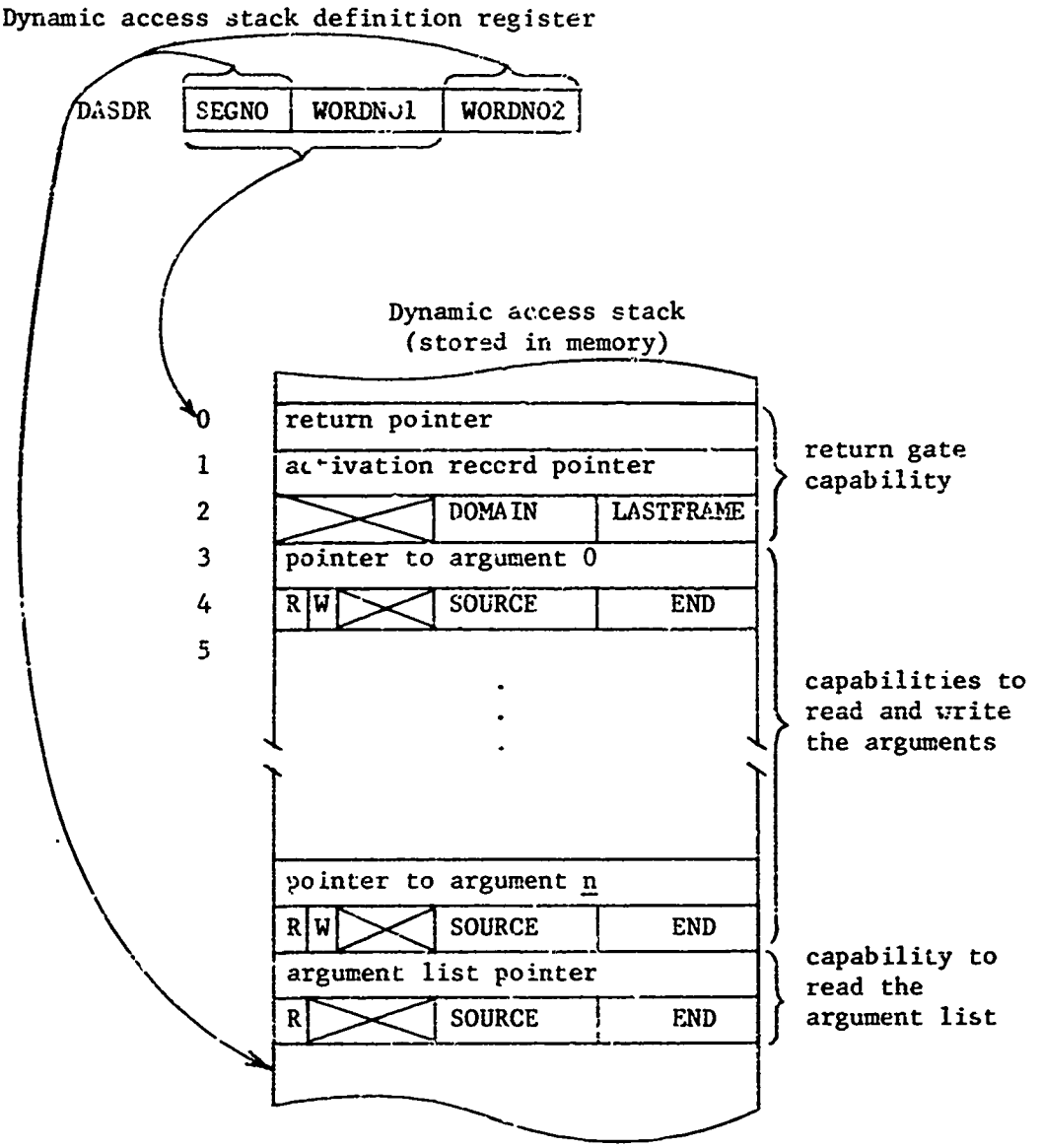


Figure 3-6: A dynamic access stack frame for a single cross-domain call.

The first capability in the DAS frame is the return gate capability. This contains the return pointer and the activation record pointer from words 1 and 2 of the argument list. It also contains the number of the domain to which the return should be made (DOMAIN), i.e., the domain of execution at the time the call is made. The final item in this capability (LASTFRAME) is the word number of the beginning of the previous DAS frame, as found in DASDR.WORDNO1.

Following the return gate capability in the DAS frame are the capabilities allowing read and write access to the various arguments. Each two-word argument entry in the argument list generates a two-word capability at the matching location in the DAS frame. The capability gives procedures executing in the called domain access to the subsegment specified by the argument entry. The first word of the capability is a pointer to the beginning of the argument, copied from the argument entry. The second word contains the read and write permission flags from the argument entry. The END field in the second word is the word number of the last word of the argument, derived by adding the size from the argument entry to the word number in the argument pointer and subtracting one. (The ending word number is easier to interpret than the size

---

footnoted statement in the text above is the first of several places where this decision is apparent in the processor design. If some procedure chooses not to conform to these standards and instead creates an activation record or an argument list to which it has dynamic access, however, the protection mechanisms will not be circumvented. But such unconventional behavior is not fully supported and that procedure will not be able to make cross-domain calls. While it is hard to imagine the circumstances under which a procedure would find creating a dynamically accessible activation record or argument list to be useful, the processor could be extended to fully support such.

when matching references to the dynamic access capabilities.)

The meaning of the SOURCE field in the second word of the capability is a little harder to explain. Recall from Chapter 2 that dynamic access capabilities must be constrained to give no more access than is available in the calling domain. In order to enforce this constraint, before making the corresponding dynamic access capability the processor must verify that the argument specified by an argument entry is in fact accessible from the calling domain in the manner indicated. There are two ways for the calling procedure to have access to the argument: access can be allowed by virtue of a static access capability in the calling domain or by virtue of a dynamic access capability in the calling domain. The tag portion of the pointer to the beginning of the argument in the argument entry specifies the opinion of the calling procedure as to which is the case. When the tag is on, specifying dynamic access, the calling procedure is indicating its intention of passing-on all or part of an argument from the earlier cross-domain call that began execution in the calling domain. When the tag is off, specifying static access, the calling procedure is indicating that the argument originates in the calling domain.

If the argument pointer tag is off, then the calling domain is the source of the argument and the processor must verify that the subsegment specified in the argument entry is accessible to the calling domain by virtue of its static access capabilities. The obvious way to perform this verification is to reference the DSE for the containing segment and check that the appropriate read and/or write permission flags are on. If the check succeeds the dynamic access capability can be created.

With this method the fact that the dynamic access capability exists is proof that the calling domain includes the required static access capability. A very important disadvantage of this method is that it makes it hard to change (or revoke) the static access from various domains of a process to a segment. Not only do permission bits in the corresponding DSE need to be changed (which is easy to do), but all dynamic access capabilities in the DAS giving access to pieces of the segment must be found and changed (which is hard to do). To get around this problem, no check of static access is made when the dynamic access capability is created. Instead, the number of the calling domain is simply recorded in the SOURCE field of the dynamic access capability as the source of the capability. The static access of the calling domain will be verified each time the dynamic access capability is used to validate a reference from the called domain. In other words, if a particular reference by a procedure executing in the called domain appears to be allowed because it matches the dynamic access capability, then the reference is also checked against the static access capabilities in the domain indicated by the SOURCE field. As a result, the dynamic access capability can give no more access than is available from the static access capabilities in the calling domain.

If the argument pointer tag is <u>on</u>, then the calling domain is not the source of the argument. The calling procedure is passing-on all or part of an argument from the earlier cross-domain call that began execution in the calling domain and the actual source of the argument is somewhere back along the chain of unreturned cross-domain calls in the process. In this case the processor must verify that the subsegment

specified in the argument entry is accessible to the calling domain by virtue of its dynamic access capabilities. These are the capabilities in the DAS frame just above the new frame being created. This previous frame is searched for a capability allowing the subsegment to be read and/or written, as required, and if one is found then the capability for the argument in the new DAS frame can be created. The SOURCE field in the new capability is set with the value from the SOURCE field in the matching capability from the previous DAS frame. As a result, when an argument (or successively smaller pieces of an argument) is cascaded through a chain of cross-domain calls, the identity of the source domain is propagated to the corresponding capability in each consecutive DAS frame.

In summary then, the SOURCE field in a dynamic access capability indicates the domain at the head of the chain of cross-domain calls down which permission to reference the corresponding subsegment has been passed. The existence of a dynamic access capability in the DAS frame for some particular cross-domain call is proof that permission to read and/or write the specified subsegment was passed along by each cross-domain call in the chain from the specified source domain to the called domain. Any time a dynamic access capability is actually used to validate a reference, the reference is also checked against the static access capabilities in the source domain for that capability.

Looking back for a moment at the return gate capability, it is now apparent that the DOMAIN field in this capability can be interpreted as indicating its source domain (which is always the calling domain) as well as the domain to which the return is to be made. It follows that,

on a cross-domain return, after verifying that PRs contains the activation record pointer recorded in the return gate capability and that the return is to the return location recorded in the return gate capability, a check must be made to assure that the return point is in a segment which by static access capability is executable in the calling domain.

After creating dynamic access capabilities for all the arguments, a final capability is created in the new DAS frame giving read access to the subsegment containing the argument list itself. (The called procedure needs to be able to read the argument list to find the arguments.) The pointer in this capability comes from PRa and the END field value is calculated from the argument list header. The source domain of this capability is the calling domain.

Creation of the new DAS frame is completed by updating DASDR to locate the new frame, i.e., DASDR.WORDNO1 is set from DASDR.WORDNO2 and then DASDR.WORDNO2 is set to indicate the word one past the end of the new frame.

Destroying the DAS frame on a cross-domain return is done by setting DASDR.WORDNO2 from DASDR.WORDNO1, and setting DASDR.WORDNO1 from the LASTFRAME field of the return gate capability in the destroyed frame.

Before leaving this discussion of dynamic access capabilities, the algorithm used to match a reference against the dynamic access capabilities of a domain needs to be stated more precisely. Assume a read reference to location (s,d) is to be validated with respect to the dynamic access capabilities of the domain of execution. The matching algorithm is to perform a linear search of the most recent DAS frame,

i.e., the frame located by DASDR, starting with word 3, looking for a capability that meets all of the following conditions:

1. read permission flag in the capability is on,
2. segment number from the pointer in the capability is s,
3. word number from the pointer in the capability is less than or equal to d,
4. END field in capability is greater than or equal to d.

The first capability that matches stops the search. The reference is allowed only if it also passes the check against the static access capabilities of the indicated source domain. If no match occurs, or the static access check fails, then the reference is not allowed. A write reference is handled in the analogous way. The linear search required can be eliminated in the case of multiple references to the same argument by the addition to the processor of a small, fast associative memory for the most recently used dynamic access capabilities. Such an associative memory will be described in more detail later.

Propagation of Tags

When a cross-domain call occurs the tag in PRa is set on. The purpose of this action is to cause all pointers generated in the called domain that are related to cross-domain arguments to have tags that are on. Then references through these pointers will be validated relative to the dynamic access capabilities added to the called domain by the cross-domain call, and not relative to the static access capabilities of the called domain. The propagation of the tag from PRa to other pointers related to cross-domain arguments is caused by the effective pointer

generation algorithm presented earlier. This algorithm guarantees that any pointer generated from a pointer whose tag is on will also have its tag on. If applied properly by procedures executing in the called domain, the algorithm guarantees that all pointers generated from PRa, and pointers generated from these pointers, etc., will have their tags set on.

An example will illustrate the method of tag propagation. Imagine that a subsegment containing a threaded list of variable-sized blocks is passed as argument 0 on a cross-domain call. The argument entry and the corresponding dynamic access capability will contain a pointer to the beginning of this subsegment and specify its length. Each block in the subsegment contains in word 1 the size of the block and in word 0 a pointer to the next block in the threaded list. The first block on the list is known to begin at word 0 of the subsegment. (It is quite possible to define and manipulate a data structure like this in PL/I using based structures.) This data structure originated in the calling domain, and therefore the pointer in the argument entry as well as all the list pointers in the data structure itself have tags that are off. When the cross-domain call occurs a dynamic access capability to read and write the subsegment is created and the tag in PRa is set on.

Now consider the machine language implementation of some operations the called procedure might perform on this argument. Execution of the following instruction sequence by the called procedure will generate in PRn a pointer to the 100th block in the list:

```
                    EPPn    PRa↑3,*
(99 times)          EPPn    PRn↑0,*
```

At the conclusion of this sequence the pointer in PRn to the 100th block will have its tag set on, even though all pointers in the 99 list blocks encountered had their tags set off. The tag from PRa is propagated all through the instruction sequence in PRn. (Refer to the effective pointer generation algorithm in Figure 3-3 to see that this really works.) All during the instruction sequence the references to the pointers in the blocks as indirect addresses will be validated with respect to the dynamic access capabilities of the called domain, and specifically will match the capability giving access to the whole subsegment that is argument 0.

Now imagine that the called procedure, after laboriously getting this pointer to the 100th block in the list, wants to save it in a temporary storage location in the current activation record. The instruction:

```
SPPn    PRs↑temp
```

does the trick. Note that this write reference to a word of the activation record will be validated with respect to the static access capabilities of the called domain, since the tag in PRs is off. However, the stored pointer will have its tag on. Later, the tracing of the list from the 100th block to the 150th block can be continued with the instruction sequence:

```
                    EPPn    PRs↑temp,*
(50 times)          EPPn    PRn↑0,*
```

The resulting pointer in PRn will still have its tag on.

There is a way to break the chain of propagation of the tag from PR<u>a</u>. If the called procedure now transfers the list pointer contained in the 150th block to another temporary storage location in the activation record using the code sequence:

```
LOAD    PRn↑0
STORE   PRs↑temp2
```

then the tag in the stored pointer will be <u>off</u>. The propagation of the tag from PR<u>a</u> is broken because LOAD/STORE generates a direct copy of the pointer from the 150th block whose tag is also <u>off</u>. The alternative sequence:

```
EPPn    PRn↑0,*
SPPn    PRs↑temp2
```

results in a stored pointer whose tag is <u>on</u>. When copying a pointer, EPP<u>n</u>/SPP<u>n</u> must be used rather than LOAD/STORE to make sure the relation of a pointer to a cross-domain argument is not lost. By violating this rule, however, a procedure can compromise only the integrity of its own domain. It cannot gain unauthorized access to information. Only self-protection is at stake.

Now imagine that the called procedure wishes to load into the accumulator the size from word 1 of the 100th block, the block whose address was stored earlier at "PR<u>s</u>↑temp". The instruction sequence:

```
EPPn    PRs↑temp,*
LOAD    PRn↑1
```

does the trick, and validates the read reference word 1 of the 100th block with respect to the dynamic access capabilities of the domain, since the tag in PR<u>n</u> will be <u>on</u>.

As the final example, imagine that the called procedure in turn generates another cross-domain call and passes as the zeroth argument this 100th block from the list. After saving the current value of PR<u>a</u> somewhere in its activation record, allocating space for the required new argument list, and resetting PR<u>a</u> to point at this new argument list area, the argument pointer in the zeroth argument entry would be created with:

EPP<u>n</u> PR<u>s</u>↑temp,*

SPP<u>n</u> PR<u>a</u>↑3

Note that this argument pointer will have its tag set <u>on</u>, correctly indicating to the processor when it constructs the required new DAS frame that the argument comes from a previous cross-domain call.

This is a good place to reemphasize why it is important that procedures executing in some domain be able to identify those pointers that are related to incoming cross-domain arguments and validate all references through such pointers against the dynamic access capabilities currently in the domain, rather than against the static access capabilities of the domain. As indicated in Chapter 2, the essential reason is that all pointers related to cross-domain arguments are derived from pointers provided by the calling domain. The pointers in PR<u>a</u>, in argument list entries, and in arguments themselves, e.g., as in the linked list example given above, are all arbitrary bit patterns set by procedures executing in the calling domain or some other domain further back along the chain of unreturned cross-domain calls for the process. When the cross-domain call occurred, the processor constructed the dynamic access capabilities in such a way that they are guaranteed to

pass-on to the called domain no more access than is dynamically or statically available from the calling domain. The processor even arranged things in such a way that if the static access capability which is the original source of permission in a dynamic access capability changes, then the change immediately propagates to the dynamic access capability. The processor, however, did not and could not check all of the pointers that can be generated starting with the pointer in PR<u>a</u>. There is no guarantee that, say, in the linked list example given above, the list pointer in the 99th block will not by malice or error point at a critical data base in the called domain that is not accessible from the calling domain, rather than point at the 100th block in the subsegment that is the cross-domain argument. When the called procedure makes a write reference via this pointer, it does so with the intention of writing something in the 100th block of the argument. If the pointer locates a portion of the critical data base, then the called procedure wants the write reference to fail. The propagation of the tag that is <u>on</u> from PR<u>a</u> guarantees that the write will fail, for this tag causes validation relative to the dynamic access capabilities created by the cross-domain call, and these capabilities will not provide write access to the critical data base. Without the propagated tag the write would succeed, for the static access capabilities of the called domain do allow write access to the critical data base.

Recall that a tag that is <u>on</u> in the effective pointer generated by a RETURN instruction is what triggers a cross-domain return. This tag propagates from the tag in PR<u>a</u> in the same way that the tags in argument-related pointers are propagated from the tag in PR<u>a</u>. To see this,

consider the last two instructions of the standard return sequence again:

EPPs PRa↑2,*

RETURN PRa↑1,*

It is clear that if the tag in PRa is on, as it would be when the corresponding call is cross-domain, then the effective pointers generated by both instructions will have tags that are on. In the case of the RETURN instruction this propagated tag triggers the cross-domain return. In the case of the EPPs instruction it generates an interesting problem. The purpose of the EPPs instruction is to restore to PRs the activation record pointer of the calling procedure. This pointer, as it was recorded by the calling procedure in the return block of the argument list, had its tag set off. (An executing procedure always has static access to its activation record.) Yet the EPPs instruction restores this activation record pointer to PRs with its tag set on. To correct this problem, part of the special action of the RETURN instruction on a cross-domain return is to set the tag in PRs off. Thus, when the calling procedure receives control following the cross-domain return, PRs properly contains the same activation record pointer it had before the cross-domain call occurred, even though the EPPs instruction restored the tag incorrectly.

The problem encountered here in restoring PRs on a cross-domain return foreshadows a general problem which will be considered later. On a cross-domain call, whenever the called procedure generates a pointer to be used by the calling procedure after the return occurs, care must be taken to set the tag properly for use by the calling procedure. For stored pointers a simple addition to the SPPn instruction, the

instruction used to create stored pointers, can detect these cases and automatically generate the correct tag. This algorithm will be presented in detail later in the chapter. The case of a pointer being communicated through a PR, however, cannot be detected by a similar algorithm added to the EPPn instruction. The only instance of this second case, however, is the communication of an activation record pointer in PRs. As seen in the previous paragraph, the tag in PRs can be adjusted easily when the cross-domain return occurs.

Multibit Tags

Exploring a little further the example of the argument that is a subsegment containing a linked list points out a flaw in the processor design as described so far. Consider the case of a cross-domain call from, say, domain 7 to domain 2 followed by a cross-domain call from domain 2 to domain 4, as illustrated in Figure 3-7. The subsegment containing the linked list is passed as a read/write argument on the first cross-domain call. The pointer in the 99th block of the list maliciously has been set by procedures executing in domain 7 to point at some data base in domain 2, a data base not accessible from domain 7. As seen above, procedures executing in domain 2 cannot be tricked by this bad pointer because the tag propagated from PRa will cause attempted read and write references through this bad pointer to fail. Suppose that instead of directly referencing this argument provided by domain 7, however, the called procedure in turn passes the subsegment as a read/write argument of the cross-domain call from domain 2 to domain 4. Another read/write argument of this second call happens to be that data

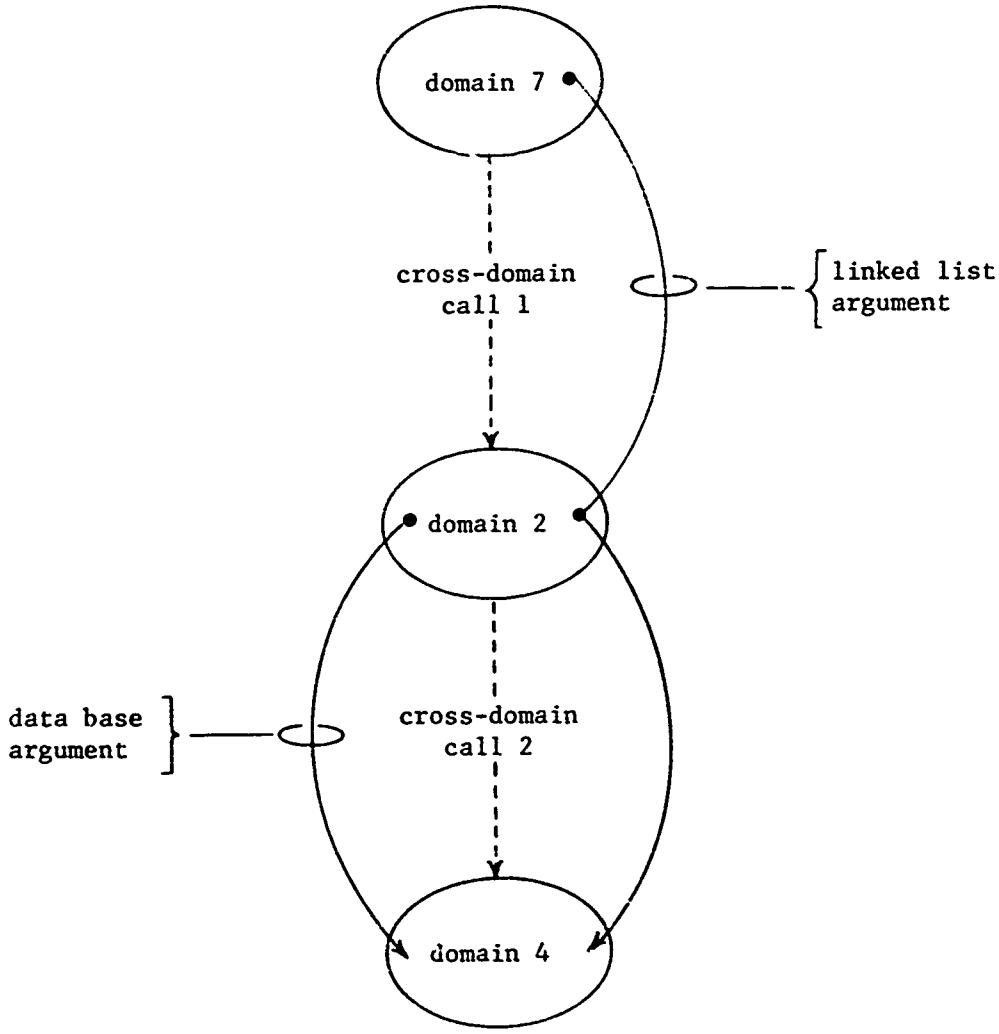


Figure 3-7: Two consecutive cross-domain calls with one argument cascaded through both.

base pointed at by the bad address in the 99th block of the linked list in the first argument.

Now consider the procedure called by this second cross-domain call. Executing in domain 4, it locates the 99th block in the linked list in the manner described earlier and then uses the contained pointer to make a write reference with the intention of writing in the 100th block. Because of the bad pointer, however, the write is actually to the data base that happens to be the other cross-domain argument provided from domain 2. Of course, the tag of the pointer used to make this reference will be <u>on</u> as before, and the reference will be validated with respect to the dynamic access capabilities associated with the second cross-domain call. In this case the reference will succeed because it is to a valid writable cross-domain argument of that call and thus a matching dynamic access capability will be found.

What really has happened here? In a sense, the procedure executing in domain 4 has not done anything wrong. It has permission to write in the data base argument in which it wrote. But this procedure did make a mistake, writing into domain 2's data base the data it was intending to put in the 100th block of the linked list in the other argument. The important thing about this mistake is that it was caused by procedures executing in a third domain. The problem illustrated by this example is particularly insidious in cases where the domain that receives the second cross-domain call encapsulates part of the supervisor and thus is trusted by the domain that made the second call. In this circumstance the procedure that made the second cross-domain call has no reason to check after the return has occurred that the trusted supervisor

has written reasonable values into the output arguments and therefore may not notice the mistake before it leads to a disastrous error. It is possible to construct many examples like this involving two or more cross-domain calls.

An analysis of why in the example given a procedure executing in domain 7 could cause a procedure executing in domain 4 to incorrectly alter data encapsulated in domain 2 reveals the general flaw in the protection mechanisms of the processor as presented so far. The reason the problem occurred is that a reference through a pointer generated from a cross-domain argument whose source was domain 7 was allowed because it matched a dynamic access capability whose source was domain 2. To prevent the general class of problem represented by this example from occurring, the processor must be altered so that a reference through a pointer generated from a cross-domain argument whose source is domain n will be allowed only if it matches a dynamic access capability whose source is domain n. This change requires expanding the tag portion of pointers to specify the source domain of the cross-domain argument from which the pointer was derived, if in fact the pointer was derived from a cross-domain argument.

The obvious value to record in a tag indicating dynamic access is the number of the source domain associated with a pointer. This number is then matched to the SOURCE field in dynamic access capabilities. There are two problems with this approach. First, when generating an effective pointer using the expanded tag, the processor must be able to determine which of two tag values indicating dynamic access corresponds to a cross-domain call that is further back in the chain of unreturned

cross-domain calls for the process. It is hard to make this determination quickly from domain numbers alone. The second problem is more important. To be consistent with programming generality it must be possible for the chain of unreturned cross-domain calls in a process to include recursive invocations of domains. When recursive invocations of domains occur the domain number by itself does not specify which invocation of a particular domain originally passed the argument from which a pointer was derived, and this distinction can be important. When the possibility of recursive invocations of domains is considered, it becomes apparent that the source of a cross-domain argument or a dynamic access capability is not just a domain but a domain invocation. What needs to be recorded in dynamic access tags and in dynamic access capabilities is the identity of a domain invocation, not just the number of a domain.

The dynamic access stack (DAS) of a process provides a convenient numbering for the domain invocations that exist at any given time. Each time a cross-domain call occurs a new DAS frame is created. The number of the frame can be used as an identifier for the domain invocation from which the cross-domain call came that generated the frame. It is these domain invocation numbers that are recorded in dynamic access capabilities and in tags specifying dynamic access. If the first DAS frame is given the number one, then the number zero can never identify a domain invocation. Thus, the tag value zero can be used to specify that reference through a pointer be validated with respect to static access capabilities. A non-zero tag specifies validation with respect to dynamic access capabilities and the non-zero value identifies the source domain invocation.

Certain aspects of the processor are now redescribed to incorporate

the expanded tag into the design. The length chosen for the tag field determines the maximum number of unreturned cross-domain calls that can exist in a process. A five bit tag, fixing this maximum at 31, is consistent with the choice of eight as the number of domains per process that the processor will support. To provide an easy means for generating DAS frame numbers and domain invocation numbers, a cross-domain call count field is added to the DAS definition register (DASDR.XDCC) as illustrated in Figure 3-8. If no unreturned cross-domain calls exist, and thus no DAS frame exist, then DASDR.XDCC = 0. Each time a cross-domain call occurs and a new DAS frame is created DASDR.XDCC is incremented by one. (DASDR.XDCC is not allowed to overflow.) A cross-domain return causes it to be decremented by one. At any given time, then, DASDR.XDCC contains the number of the most recent DAS frame. This value is also the number of the domain invocation from which the cross-domain call came that started execution in the current domain of execution. In the state shown in Figure 3-8, for example, three unreturned cross-domain calls exist. It is clear that at any given time only tag values less than or equal to DASDR.XDCC mean anything.

The modified effective pointer generation algorithm is shown in Figure 3-9. It looks fairly complex but has a simple effect. If only tags of zero are encountered during effective pointer generation, then the final effective pointer has a tag of zero, indicating that the reference through the pointer will be validated relative to the static access capabilities of the domain of execution. If only one non-zero tag is encountered (in the indicated PR or indirect address), then assuming the value of this tag is meaningful, i.e., ≤ DASDR.XDCC, the final effective

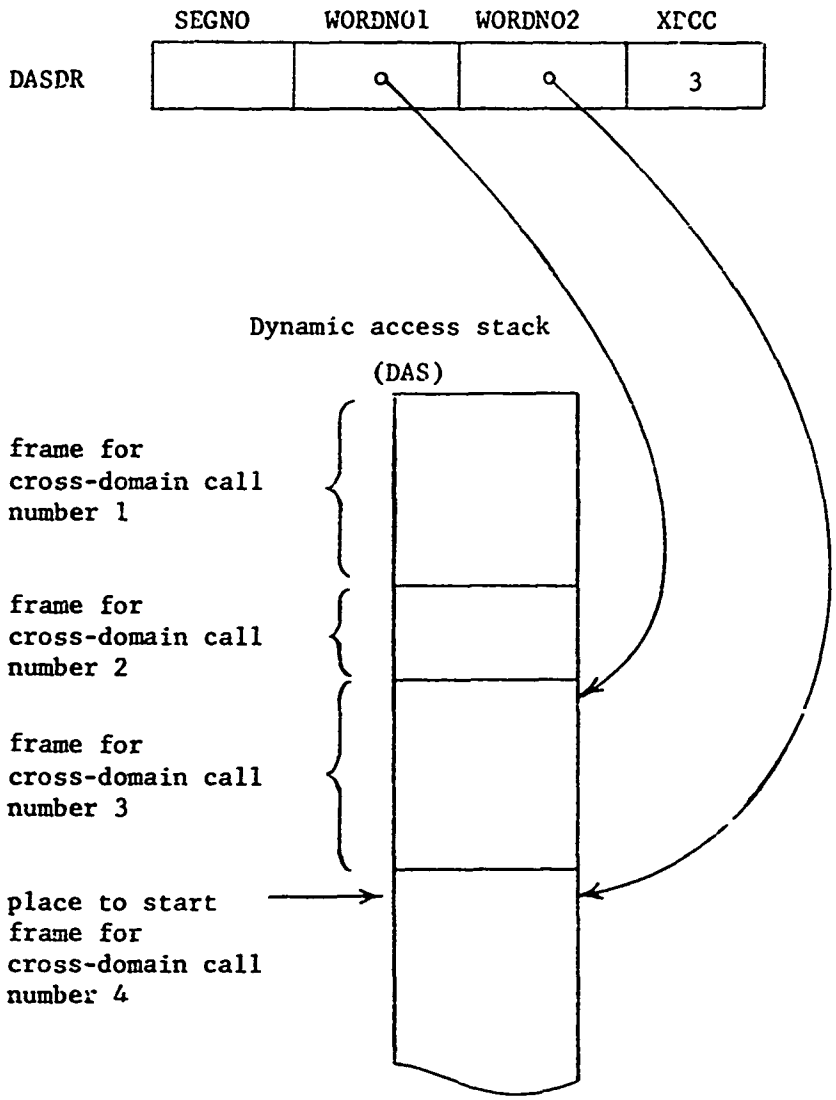


Figure 3-8: Dynamic access stack containing frames for three unreturned cross-domain calls.

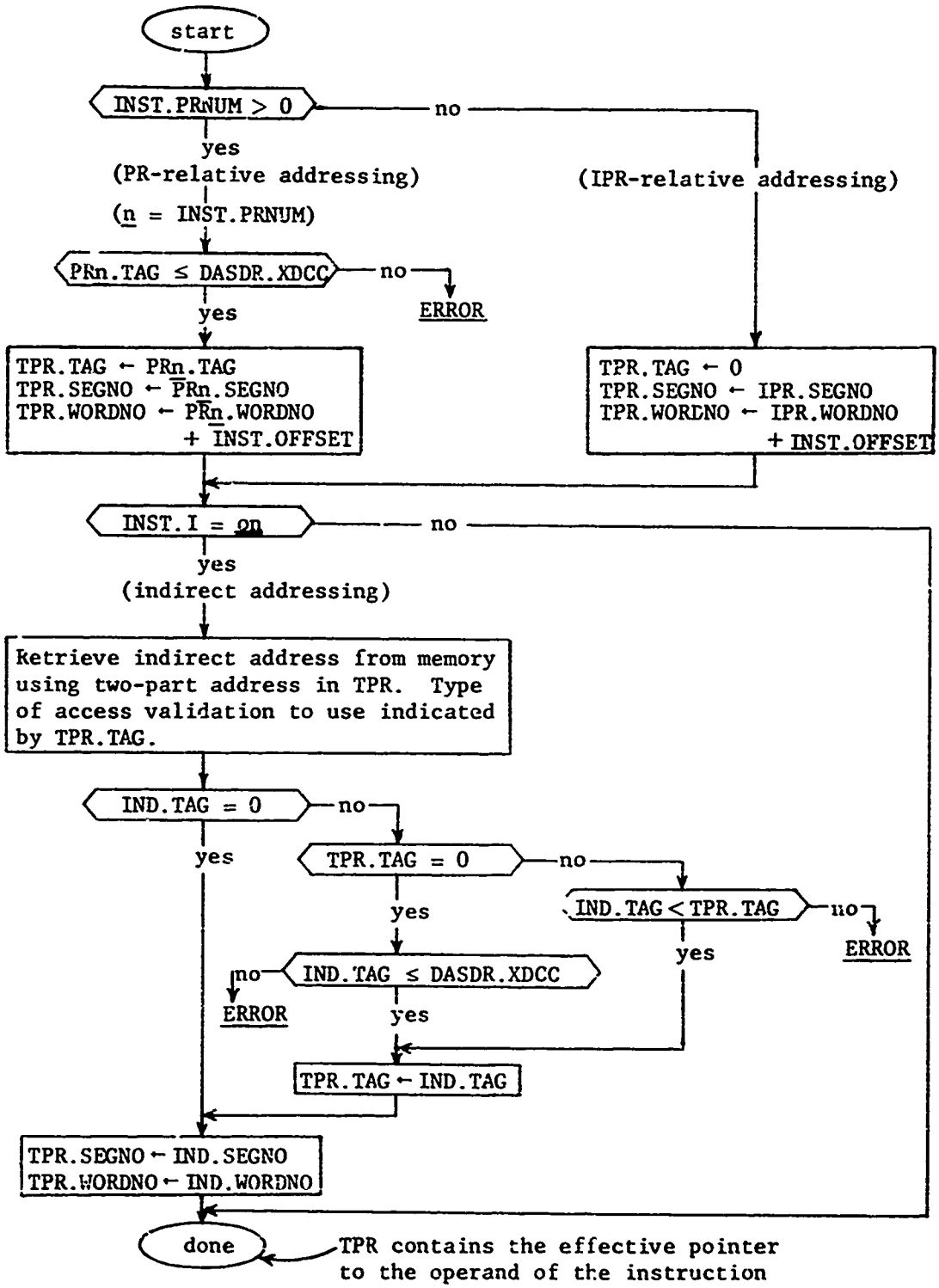


Figure 3-9: Modified algorithm for effective pointer generation with a multibit tag.

pointer has the same non-zero tag value, indicating that the reference through the pointer will be validated relative to the dynamic access capabilities currently in the domain of execution. The non-zero tag value is the source domain invocation number to be used in finding a matching dynamic access capability. The interesting case is when both PR-relative and indirect addressing are indicated, and both the PR and the indirect address have non-zero tags. Assuming no error is generated because the PR tag is greater than DASDR.XDCC, the indirect address tag is checked to make certain it is smaller than the PR tag, and if it is then the indirect address tag provides the value for the tag in the final effective pointer. That the indirect address tag be smaller than the PR tag is important. A non-zero PR tag with a value $\underline{t}$ indicates that the indirect address located by the PR is in the argument list or is in an argument of the $\underline{t}^{th}$ unreturned cross-domain call. From the vantage point of domain invocation $\underline{t}$ (the domain invocation from which the $\underline{t}^{th}$ cross-domain call came), only tag values less than $\underline{t}$ mean anything. When the argument list was created, $\underline{t}$-1 was the number of the most recent DAS frame. In that circumstance a pointer with a tag value greater than or equal to $\underline{t}$ would not mean anything. Further, if the effective pointer generation algorithm accepted a tag value strictly greater than $\underline{t}$ from the indirect address located by the PR, then one aspect of the previously described flaw would remain uncorrected. The effective pointer formed using information provided by domain invocation $\underline{t}$ would specify in its tag a domain invocation number greater than $\underline{t}$.

Propagation of the new expanded tag is by essentially the same means used for the one bit tag. When a cross-domain call occurs the processor

sets the tag in PRa with the incremented value of DASDR.XDCC. This is the number of the domain invocation from which the cross-domain call comes. For example, if a cross-domain call occurred when the DAS were in the state illustrated by Figure 3-8, then PRa.TAG would be set to four. Correct use of the effective pointer generation algorithm by procedures executing in the called domain guarantees propagation of this tag (and smaller non-zero tags if encountered) to all pointers derived from cross-domain arguments.

The purpose of the expanded tags, of course, is to provide an additional parameter to use when matching a reference to the dynamic access capabilities currently in the domain of execution. Figure 3-10 presents the detailed format of a dynamic access capability giving read and/or write access to a subsegment. The TAG field in the first word is used to record the number of the domain invocation that is the source of the capability. The SOURCE field in the second word is the domain number associated with the source domain invocation. The TAG field provides the value to be used when matching a reference through a pointer to the dynamic access capabilities currently in the domain. Once the dynamic access capability that matches a reference has been selected, the contained SOURCE field specifies which domain's static access capabilities must also allow the reference, as before.

Recall that a dynamic access capability to read and/or write a subsegment is created for each argument entry in the argument list of a cross-domain call. (One is also created to allow read access to the argument list itself.) With the expanded tag, creation of the dynamic access capability for an argument entry is still done pretty much as described earlier for the one bit tag. There are two cases. If the tag

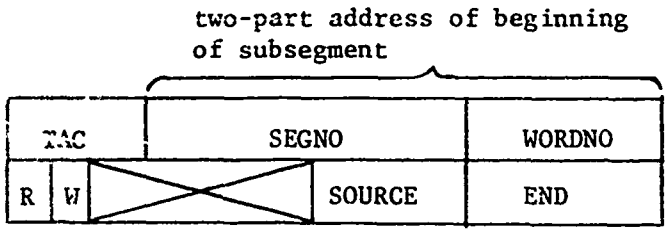


Figure 3-10: Format of a dynamic access capability giving read and/or write access to a subsegment.

in the argument pointer from the argument entry is zero, then the source of the argument is the domain invocation from which the cross-domain call is being made. In this case the SEGNO, WORDNO, END, SOURCE, R, and W fields of the dynamic access capability are set as before. The TAG field is set with the number of the domain invocation from which the call is being made. (Adding one to the value in DASDR.XDCC at the time the new capability is being created produces this number.) In the second case the tag in the argument pointer is greater than zero, indicating that an argument (or piece of an argument) whose source domain invocation is further back in the chain of unreturned cross-domain calls in the process is being passed-on by this cross-domain call. The value of the tag identifies the source domain invocation for the argument. In this case the DAS frame previous to the new frame being created is checked for a matching dynamic access capability as before. (The new matching algorithm presented below is used.) If a match is found then the new dynamic access capability can be created. The SEGNO, WORDNO, END, SOURCE, R, and W fields are set as before, with the source field value being copied from the matching capability. The TAG field in the new capability is set from the tag value in the argument pointer.

The dynamic access capability allowing references to the argument list is also constructed as before. Its TAG field is set with the number of the calling domain invocation, for the source of this capability is always the calling domain invocation.

The result of the way the TAG fields in dynamic access capabilities are set is that each capability in a DAS frame correctly indicates the number of the domain invocation that is its source.

As indicated, the value of the TAG field in a dynamic access capability is taken into account when matching a reference to the dynamic access capabilities currently in the domain of execution. While the only difference from the algorithm presented earlier for the case of one bit tags is the additional constraint that the tag from the pointer must match the tag in a capability, the entire algorithm for matching a reference to the dynamic access capabilities in a domain is presented again for the sake of clarity. Assume a read reference is made through a pointer containing the tag $\underline{t}$, where $\underline{t} > 0$, and the two-part address ($\underline{s}$,$\underline{d}$). The reference validation algorithm is to perform a linear search of the most recent DAS frame, starting with word 3, looking for a capability that meets all of the following conditions:

1. Read permission flag is <u>on</u>.
2. TAG field contains the value $\underline{t}$.
3. SEGNO field contains the value $\underline{s}$.
4. WORDNO field $\leq \underline{d} \leq$ END field.

The first capability that matches stops the search. The reference is allowed only if it also passes the check against the static access capabilities of the domain indicated by the SOURCE field. Again, validation of a write reference is analogous.

The multibit tag does not change the way the return gate capability in a DAS frame is matched. At any given time only one cross-domain return is possible in a process. The only valid non-zero tag which can be generated in the effective pointer of a RETURN instruction is the number of the most recent DAS frame. The pointer in PR$\underline{s}$ and the location to which the return transfer is directed must match the two pointers

in the single return gate capability in that frame as before.

This completes all but one of the changes and additions to the processor necessary to incorporate multibit tags. The last remaining addition is presented in the next section.

Pointers as Arguments

In the previous two sections the example of a cross-domain call argument that contained stored pointers, i.e., indirect words, was used. The stored pointers provided the links in a threaded list of variable-sized blocks. There are several subtleties related to the use of stored pointers in cross-domain arguments that must be explored.

The PL/I language includes a "pointer" data type. In a machine such as that being described here, PL/I pointer variables would be implemented as stored pointers. Thus, an argument that is a stored pointer or contains stored pointers will occur whenever a PL/I program specifies a pointer variable itself or a data structure containing pointer variables as an argument of an interprocedure call.

First consider the use of a pointer variable as an output argument of a cross-domain call. For simplicity of explanation the discussion is presented in terms of an output argument that is a single pointer variable, although the ideas presented also apply to output arguments which are larger data structures containing pointer variables. When the cross-domain call occurs, a dynamic access capability will be created allowing procedures executing in the called domain to write into (and read from) the subsegment containing the pointer variable. The calling procedure must recognize that, by specifying that a pointer

variable is an output argument, the called procedure in another domain is given the ability to write an arbitrary bit pattern into this pointer variable. Thus, after the return has occurred, the calling procedure must explicitly check that a reasonable value has been stored in this pointer variable before it is used to make a reference if the calling procedure has any reason to mistrust the called domain. Normally, when a pointer variable output argument is passed on a cross-domain call, the called domain is in fact trusted, e.g., the called domain may encapsulate part of the supervisor.

The interesting question about pointer variable output arguments for cross-domain calls is what tag value will be placed in the pointer. If t is the domain invocation number from which the cross-domain call is made, then only tag values strictly less than t will mean anything to the calling procedure. Yet to the called procedure in the next higher numbered domain invocation, a tag value equal to t certainly means something and such a tag value might be stored into the pointer variable output argument. To see this consider the following example. A procedure executing in domain invocation t passes as an input argument of a cross-domain call some data structure and as an output argument a pointer variable. The called procedure is expected to locate a certain item within the input data structure and return a pointer to this item in the output pointer variable. Because pointer variables are implemented as stored pointers, i.e., indirect addresses, the called procedure will write the value into the pointer variable output argument using the SPPn instruction. The pointer in PRn will have been generated by applying the effective pointer generation algorithm to the input data

structure in various ways, and thus will contain the tag value t propagated from PRa. If the [illegible] instruction simply stores the whole pointer from PRn as its operand, then the output argument pointer variable will also have a tag value of t, a tag value that means nothing to the calling procedure. From the point of view of the calling procedure and the domain invocation in which it is executing, the correct tag is zero, since after the return occurs the pointer variable will be used to reference the data structure which is statically accessible in that domain.

To solve the problem illustrated by this example, and others like it, the SPPn instruction does not always just copy PRn.TAG into the indirect address being created. It is able to detect the circumstances in which the indirect word tag should be set to zero, and will set it to zero in these cases. The algorithm is presented in Figure 3-11. When performing the SPPn instruction there are two tag values to consider: the value in TPR.TAG and the value in PRn.TAG. In the simple case that TPR.TAG is zero, then the operand of the instruction is not a cross-domain argument. The indirect address should be constructed directly from the pointer in PRn, tag and all. Whatever tag may have propagated into PRn needs to be preserved in the stored pointer.

If the tag in TPR is non-zero, then the operand of the instruction is a cross-domain output argument that is a pointer variable, and the potential exists for the sort of problem to occur that was illustrated by the example given earlier. Specific examples of three general cases which may occur are illustrated in Figure 3-12. In all cases the SPPn instruction being executed is writing a pointer into a pointer variable

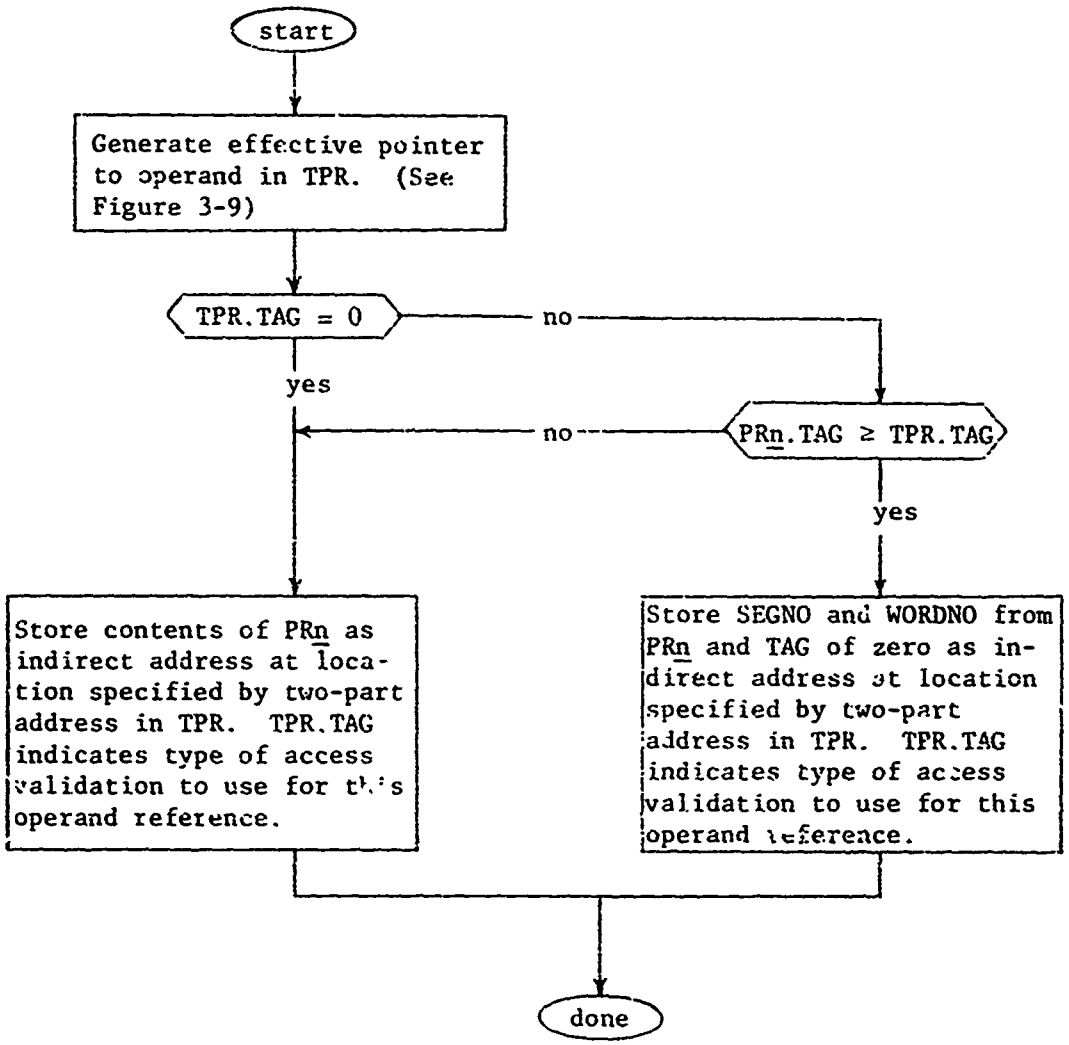
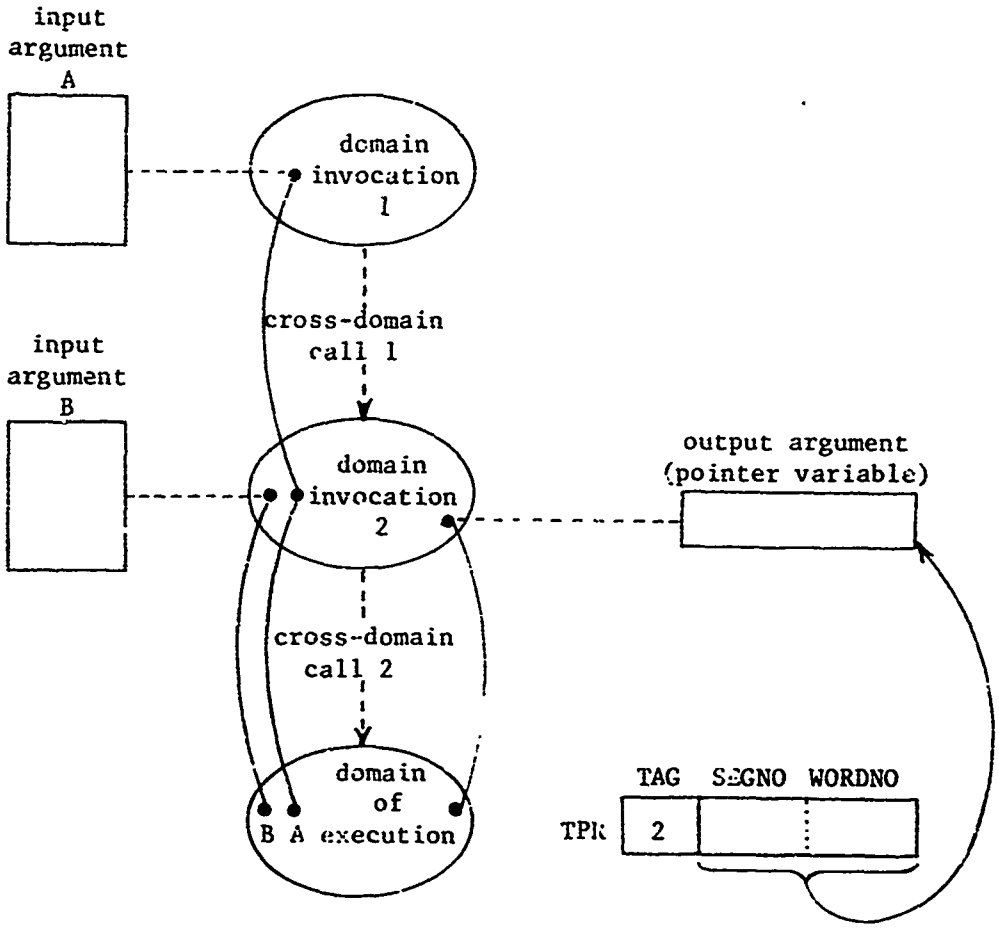


Figure 3-11: Algorithm of the SPPn instruction.



PRn:	TAG	SEGNO WORDNO
case 1	0	address derived internally
case 2	2	address derived from B
case 3	1	address derived from A

	Resulting TAG value in output argument
case 1	0
case 2	0
case 3	1

Figure 3-12: Three cases of the SPPn instruction.

that is an output argument of a cross-domain call. The source of this argument is domain invocation 2. Thus, the tag in TPR is two.

In case 1 the pointer in PR<u>n</u> was constructed from scratch or derived from pointers not related to cross-domain arguments, and therefore has a tag of zero. For example, the domain of execution might be part of the supervisor, and the procedure invoked by cross-domain call 2 might be the procedure which converts a symbolic segment name provided as an input argument into an output argument that is a pointer to the base of that segment. According to Figure 3-11, this zero tag will be stored in the indirect address. When control returns to domain invocation 2, the source of the pointer variable output argument, the zero tag will cause references through this pointer to be evaluated relative to the static access capabilities of that domain. This is correct since the pointer certainly was not derived from an argument of the cross-domain call from domain invocation 1.

In case 2 the pointer in PR<u>n</u> was derived from input argument B. The source of this argument is domain invocation 2, and PR<u>n</u> will contain a tag of two propagated from PR<u>a</u>. The SPP<u>n</u> instruction, however, will store a tag value of zero in this case, since PR<u>n</u>.TAG = TPR.TAG. This is the correct tag from the viewpoint of domain invocation 2, since this pointer locates part of a data base that is statically accessible from that domain. It is also the correct tag from the point of view of the domain of execution. If this stored pointer is later read (using an EPP<u>n</u> instruction) by procedures executing in this domain before the cross-domain return to domain invocation 2 occurs, then the fact that the stored pointer is a cross-domain argument with domain invocation 2

as its source will cause a tag of two to propagate back into the retrieved copy of the pointer. Thus, from the point of view of the domain of execution, no information is lost by cancelling the tag of two when the pointer is stored.

In case 3 the pointer in PRn was derived from input argument A. The source of this argument is domain invocation 1, and PRn will contain a tag of one propagated from the corresponding argument pointer in the argument list of cross-domain call 2. In this case the tag of one is stored unaltered in the pointer variable, for PRn.TAG < TPR.TAG. This is the correct tag from the viewpoint of domain invocation 2, since the pointer was derived from an argument whose source was domain invocation 1. Use of the pointer variable by procedures executing in domain invocation 2 to make a reference should be validated with respect to the dynamic access capabilities currently in this domain. It is also the correct tag from the viewpoint of the domain of execution, since it properly indicates the source domain invocation of the argument from which the pointer was derived.

Now consider the use of pointer variables as input arguments on cross-domain calls. This use of pointer variables does not lead to further additions to the processor design, but instead creates a new application for the existing protection mechanisms. Again, the discussion will be presented in terms of a single pointer variable used as an input argument, but all ideas presented can be applied to the case of an input argument that is a data structure containing pointer variables.

The usual reason for passing a pointer variable as an input argument is to explicitly inform the called procedure of the location of some data item. The called procedure will reference the data item through the pointer variable. If the call is cross-domain, and the data item is not a formal argument of the call, then a problem can result, for no dynamic access capability will be created allowing procedures executing in the called domain invocation to reference the data item. While it can be argued that passing a pointer to the data item as the argument rather than the data item itself is bad programming practice, it is frequently done anyway. This technique allows complex PL/I structures to be passed as implicit arguments, avoiding the syntactically messy specification of the structure as a formal argument. The technique can also be used to pass as implicit arguments data items which are too complex to be expressed as formal arguments with the syntax of the language.

For whatever reason, pointer variables are used in input arguments to pass implicit arguments. It would be nice if the protection mechanisms of the processor could deal with this case. Fortunately they can. So far the argument list associated with a cross-domain call has been treated solely as a specification of the formal arguments of the call. The argument list serves a dual purpose when a call is cross-domain: it informs the called procedure of the addresses of the argument and it tells the processor how to create dynamic access capabilities. The called procedure knows how many formal arguments to expect. If extra argument list entries were appended to the end of the argument list, after those for the formal arguments, it would not confuse the called procedure. It would, however, cause the processor to create

extra dynamic access capabilities in the case of a cross-domain call. This property can be used to cause the creation of dynamic access capabilities allowing access to implicit arguments.

If a compiler is to generate code which creates at the end of the argument list generated for a call extra argument entries for implicit arguments, then the compiler somehow needs to be told what extra entries are to be created. A variety of high-level language extensions can be imagined to serve this purpose. The specific extension described here fits well into PL/I. The extension is a built-in function named "window". The function has three arguments: a pointer, an integer, and a one bit flag. The appearance of the window function at some position in the argument list portion of a call statement causes an argument entry to be generated at the corresponding position in the argument list. The pointer provided becomes the pointer in the first word of the entry, the integer becomes the size in the second word, and the one bit flag controls the generation of the read and write permission flags: if off then only read permission is indicated, if on then both read and write permission are indicated.* Window functions will typically appear last in the list of arguments for a call statement, and will define the subsegments containing the implicit arguments.

Window functions can be used to create fairly complex patterns of access permission. In particular, one window function (or a formal argument) can provide read access to a large subsegment while several

---

*Write only access to a window could be specified as well by expanding this argument to two bits.

other window functions provide read/write access to selected pieces of that subsegment.

Window functions, then, allow input arguments that are pointer variables to be used to pass implicit arguments. The pointer variable is listed as a formal argument of the call and the implicit argument is defined by one or more window functions at the end of the argument list. It can be argued that window functions are contrary to the criterion of programming generality. In a sense this contention is correct. A call that works without window functions in cases where the domain of execution is not changed may require window functions to work cross-domain. The only answer is always to use window functions when implicit arguments are being passed, even if the call is not expected to change the domain of execution. The extra specification provided by the window functions, while not functionally required for calls that do not change the domain of execution, does help to make the intention of the program clearer to programmers and others who must understand the symbolic program.

The Escape Hatch

The protection mechanisms of the processor are built upon the philosophy that all references made by a procedure executing in some domain invocation through pointers derived from incoming cross-domain arguments should be validated relative to the dynamic access capabilities of that domain invocation. The mechanisms for propagating tags cause this kind of validation of all such references as a matter of course. There may arise circumstances, however, when a procedure wishes to

violate this philosophy intentionally. For example, two domains may share static access to a segment. A procedure executing in one may call a procedure that executes in the other to provide the latter with a pointer to the shared segment. The called procedure may store the pointer as an internal static variable to be used later. The segment is not an implicit argument of the cross-domain call and no dynamic access capability for the segment ever exists. The intentions of the called procedure are properly served if the stored pointer has a tag of zero. Yet, the standard pointer manipulation code will produce a non-zero tag in this pointer, for it is related to a cross-domain argument. In cases such as this, there needs to be a way to force the tag in a pointer to zero. For this purpose a built-in function should be provided.

Such a function for PL/I might be named "neutralize" and take a single pointer as its argument. The result is another pointer with the same two-part address, but a tag of zero. When applied to pointers that either are cross-domain arguments or are derived from cross-domain arguments, neutralize would produce an equivalent pointer with the difference that references through it will be validated with respect to the static access capabilities of the domain of execution. Needless to say, a procedure must be very careful when applying the neutralize function to a pointer, for use of this function has the potential to violate the integrity of the domain of execution.

Processor Response to Exceptional Conditions

Each reference made by an executing machine language program to a virtual memory location is validated by matching it to the static or

dynamic access capabilities in the domain of execution. The method of validation and the specific conditions under which a given reference will be allowed have been discussed in some detail. An important point not considered yet is the action of the processor in cases where validation indicates a reference should not be allowed. This occurrence is called an access violation.

An access violation is one of several types of exceptional conditions that the processor can detect. Other examples are the attempted execution of a privileged instruction from some domain other than domain 0, the detection of an error while performing a cross-domain call or return, an attempted reference beyond the end of a segment, and the attempted use of an invalid segment number. There may be many other processor detected exceptions as well. All, including access violations, cause the processor to change the domain of execution to domain 0 and transfer control to a fixed location. Presumably the software system is arranged so that this location is the beginning of the supervisor procedure which responds to exceptions. The processor is arranged so that the processor state at the time of the exception is preserved. The supervisor procedure can inspect this state information to determine the precise cause of the exception. Corrective action can be initiated if appropriate. A privileged instruction allows the processor state to be restored, so that if the cause of the exception is corrected then the disrupted program can be restarted at the point where the exception occurred.

The transfer to domain 0 in the case of an exception, although it may be a cross-domain transfer, does not affect the state of the dynamic

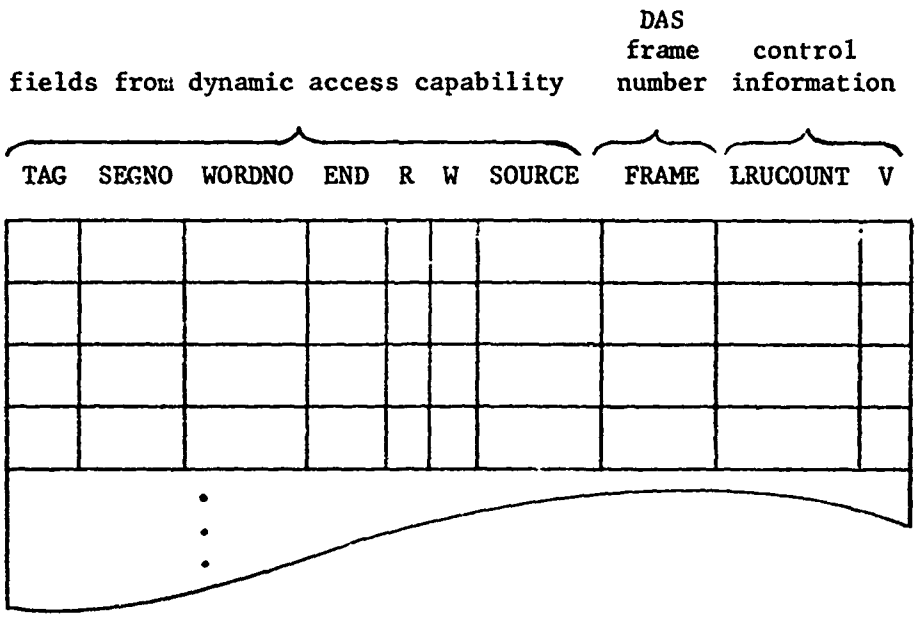
access stack or its definition register. The segment containing this stack, as well as the descriptor segment, are statically accessible from domain 0. In addition, privileged instruction exist to load and store the DSBR, DR, and DASDR of the processor. Thus, the supervisor procedures invoked by the exception are able to make any necessary alterations to the protection environments provided by the various domains of the process.

An Associative Memory for Dynamic Access Capabilities

As indicated earlier, the operation of validating a reference against the dynamic access capabilities currently in the domain of execution can be made faster in the case that multiple references match the same capability by adding to the processor a small, fast associative memory which stores the most recently used dynamic access capabilities. In this section the design and use of such an associative memory is presented. Those readers generally familiar with such associative memories can skip this section without loss of continuity.

The associative memory has a small number of registers (eight is probably more than enough), each of which can hold all fields of a dynamic access capability for read and/or write access to a subsegment. In addition, each register contains a few bits of identification and control information. The format of the registers is shown in Figure 3-13.

This associative memory (DAC-AM) has four cycles: SEARCH, WRITE, CLEAR-FRAME, and CLEAR-ALL. Consider the SEARCH cycle first. Whenever a reference is to be validated with respect to the dynamic access



input parameters for search: tag, segment number, starting word number, ending word number, type of reference (read or write), current DAS frame number (DASDR.XDCC)

output from search: match indicator, source domain number if match occurred

Figure 3-13: Format of the associative memory for dynamic access capabilities.

capabilities currently in the domain of execution, the DAC-AM SEARCH cycle is invoked before a linear search of the most recent DAS frame is performed. The input parameters to the SEARCH cycle are listed in Figure 3-1[illegible]. The first five define the reference being made. Both a starting and ending word number are provided so that double-word (and larger multiword) references can be validated. The last input parameter is the number of the current DAS frame, as recorded in DASDR.XDCC. The hardware performs an associative search of all DAC-AM registers, looking for one containing a dynamic access capability which allows the reference and which is from the indicated DAS frame. If a match occurs, then a positive match indicator is the result and the SOURCE field from the matching capability is produced.* In this case the linear search of the DAS frame in memory can be bypassed, and validation of the reference can proceed to the stage where the static access from the indicated source domain is checked. If no match is found, then a negative match indicator is the result, and the DAS frame in memory must be searched in the way previously described for a matching dynamic access capability.

The LRUCOUNT field of DAC-AM registers is used to order the dynamic access capabilities contained in the DAC-AM by recency of use. Each time a SEARCH cycle match occurs, the LRUCOUNT of the matching register is set to the maximum possible value, and all other LRUCOUNT fields that are greater than the original LRUCOUNT value of the matching register

---

*Actually, more than one DAC-AM register may contain a matching capability. Since the SOURCE fields in all capabilities that match will be the same, however, it does not matter which is chosen as the single matching DAC-AM register.

are decremented by one.

If a SEARCH cycle results in no match, but the linear search of the DAS frame in memory does succeed, then the dynamic access capability found this way is entered into the DAC-AM using the WRITE cycle. The various fields of this capability overwrite the contents of the DAC-AM register whose LRUCOUNT value is zero. This is the register that has gone longest without producing a match. The FRAME field is set from DASDR.XDCC to indicate the DAS frame from which the capability came. The LRUCOUNT field is set to the maximum possible value and the LRUCOUNT fields in all other registers are decremented by one. Thus, the DAC-AM always contains the most recently used dynamic access capabilities for a process. Repeated use of a small group of dynamic access capabilities, as is likely, will result in most searches of the DAC-AM succeeding. As a result, the more time-consuming linear search of the DAS frame in memory will not occur each time dynamic access validation is required.

When a cross-domain call occurs, the DAC-AM can be used in two ways. First, it can be applied to locating a matching dynamic access capability in the previous DAS frame when constructing a new dynamic access capability from an argument entry whose pointer has a non-zero tag. (This is a case where the ability of the DAC-AM to find capabilities matching multiword subsegments is used.) Second, it can be preloaded explicitly (using the WRITE cycle) with the newly constructed capability allowing read references to the argument list itself, since the likelyhood of this capability being used in the near future is great.

When a cross-domain return occurs* the CLEAR-FRAME cycle of the DAC-AM is invoked. This invalidates all registers whose FRAME field indicates that the contained capability is from the DAS frame being destroyed by the return. Invalidation is performed by setting the validity bit (V) off. Only registers whose validity bit is on can match on a SEARCH cycle. The LRUCOUNT fields in all registers so invalidated are adjusted so that these registers will be the next to be overwritten.

The final cycle, CLEAR-ALL, is used when the process is execution on the processor is changed. It causes all registers to be invalidated.

The DAC-AM will decrease the time required to validate a reference against dynamic access capabilities only in the case that multiple references are validated against the same capability. It will almost always be the case that the capability giving read access to a cross-domain argument list will be used several times. Multiple references to individual arguments are not uncommon -- particularly to arguments that are large arrays or structures. Considering the low cost of such an associative memory and the off-the-shelf availability of the control logic for performing the associative search and updating the LRU counters, the DAC-AM seems like a good investment.

In addition to the DAC-AM, the processor also can contain a similar associative memory for the most recently used descriptor segment entries (DSE's). Associative memories for address translation table entries are

---

*There is no reason to put return gate capabilities in the DAC-AM, since each is used only once. The position of the return gate capability in a DAS frame is always known so no search of the DAS frame is required to find it in any case.

of proven value in reducing the average time required to translate an address [24].

Entry and Label Variables

In PL/I, entry and label variables may be passed as arguments on interprocedure calls. It is possible for the called procedure to perform a goto operation on a label variable received as an argument and to perform a call operation on an entry variable received as an argument. The processor as currently defined can handle neither of these cases automatically if the call which passes the entry or label variable as an argument is cross-domain. The techniques underlying the processor protection mechanisms described so far, however, can be applied to these cases as well. This section investigates the processor extensions that would be required for allowing entry and label variables to be used as arguments of cross-domain calls.

Presentation of the material of this section is not meant to imply that the facility to pass entry and label variables on cross-domain calls must be included in the processor if it is to be practical. Indeed, a good case can be made for omitting this facility considering the additional complexity that it introduces. The section is included in the thesis only to illustrate a more sophisticated application of the basic techniques used in the processor than has been presented so far in this chapter.

First consider entry variables. As usually implemented an entry variable is a pair of pointers. The first locates an entry point while the second locates an activation record. If the entry variable locates

an entry point into the outermost level of a nested set of PL/I procedures, i.e., an external entry point, then the second pointer means nothing. If the entry variable locates an entry point into one of the inner layers, i.e., an internal entry point, then the second pointer locates an already existing activation record that corresponds to some unreturned invocation of the containing procedure. When a call through an entry variable to an internal entry point occurs, this activation record pointer must be made available to the called procedure to define part of its addressing environment.

A standard implementation of a call through an entry variable needs to be defined, just as a standard interprocedure call was defined earlier. The implementation chosen is substantially the same as that for a normal interprocedure call. The only difference is that the two instructions:

```
EPPs    "entry variable"↑1,*
CALL    "entry variable"↑0,*
```

where "entry variable" represents the address of the entry variable, are used in place of the single CALL instruction of the standard interprocedure call. The first instruction loads PRs with the second pointer from the entry variable. The second instruction is the previously defined CALL instruction with an operand address specification that locates the entry point specified by the first pointer in the entry variable. As a result of a call through an entry variable, the called procedure can expect to find PRa containing a pointer to the argument list, PRb containing a pointer to the base of the stack segment in which a new activation record should be created, and PRs containing the second

pointer from the entry variable. If the called entry point is external, then the called procedure will ignore pointer in PR<u>s</u>. If it is internal, then the pointer in PR<u>s</u> will be used by the called procedure to locate variables belonging to the appropriate outstanding invocation of the enclosing procedure block. That a call is through an entry variable is apparent to the compiler, so it can choose properly whether to generate the code for a standard interprocedure call or the code for a call through an entry variable.

There are a variety of ways to use entry variables as arguments. They may appear singly, in arrays, or as elements of non-homogeneous data structures. Consideration of the simple case of a single entry variable passed as an individual argument, however, demonstrates the problems generated by the use of entry variable arguments on cross-domain calls. Imagine that procedure A passes an entry variable as an input argument on a call to procedure B with the intention that B (or a descendant procedure) perform a call through this entry variable. With the protection mechanisms defined so far, an argument list like that illustrated in Figure 3-14 would be generated by A. An argument list entry specifies read access to the two word entry variable. If the A to B call is cross-domain, a dynamic access capability will be generated which allows procedures executing in the called domain read access to the entry variable.

When time comes for B to make the call through the entry variable a temporary pointer to the entry variable will be generated in PR<u>n</u> with the instruction:

EPP<u>n</u> PR<u>a</u>↑<u>m</u>,*

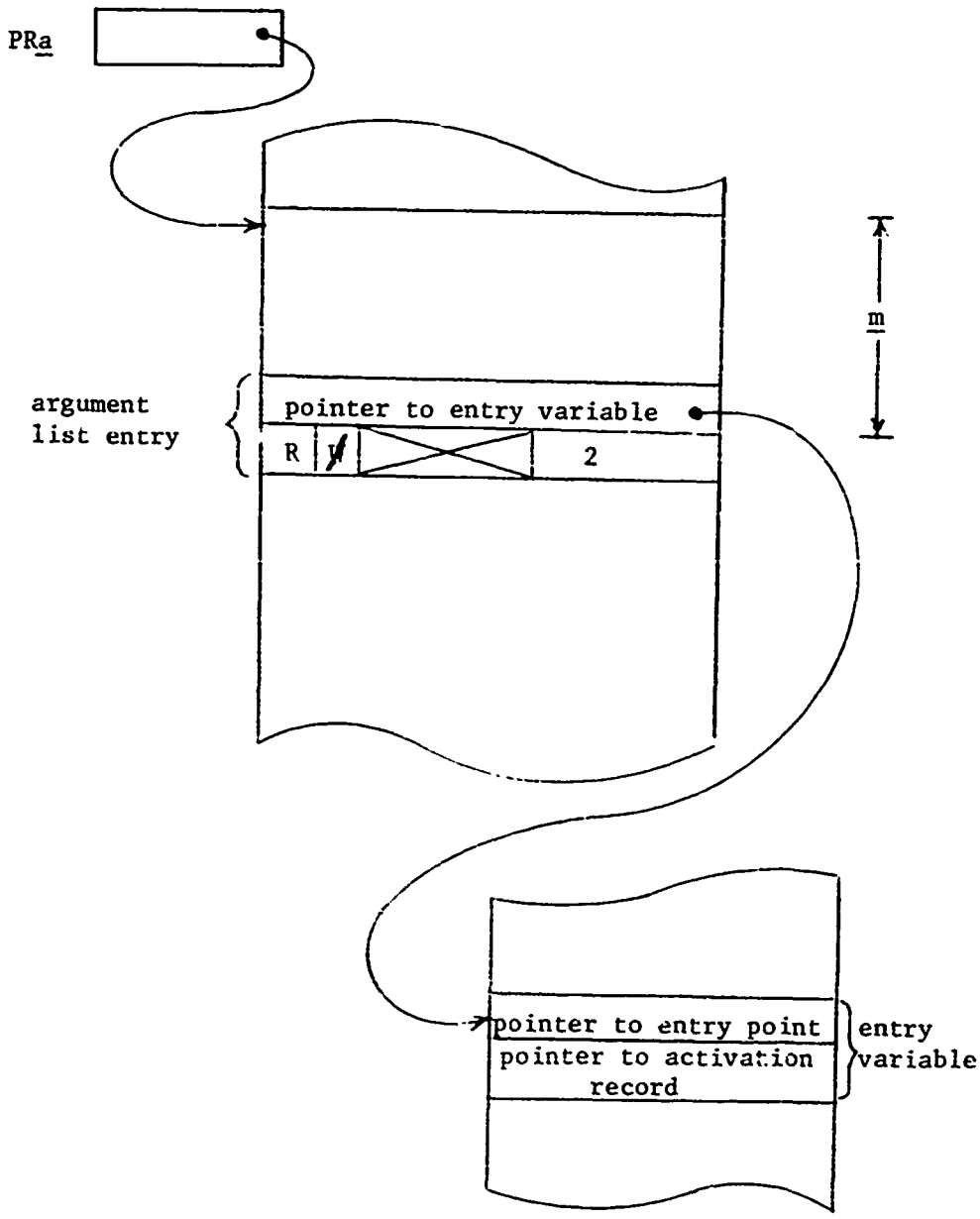


Figure 3-14: Specification of a single entry variable as an input argument.

where m is the offset of the argument list entry for the entry variable. Then an argument list will be created and PRa set with a pointer to it. Finally, the call through the entry variable will be made by procedure B with the instructions:

```
EPPs    PRn↑1,*
CALL    PRn↑0,*
```

Because the entry variable itself is a cross-domain argument, the temporary pointer to it generated in PRn will have a non-zero tag. This tag causes the indirect word read reference to the entry variable generated by the EPPs instruction to be validated relative to the dynamic access capabilities in B's domain invocation. This reference will succeed by matching the capability allowing read access to the entry variable; likewise for the indirect word read reference caused by the CALL instruction. The final effective pointer generated by the CALL instruction also will have a non-zero tag -- indicating that the call is to be validated with respect to dynamic access capabilities in B's domain of execution. It is here where the protection mechanisms defined in the previous section are inadequate. No matching dynamic access capability can be found because dynamic access capabilities allowing call access to an entry point have not been defined. The protection mechanisms discussed so far only can handle the case of a CALL instruction which generates a tag of zero in its effective pointer. The extension required to allow the use of entry variables as cross-domain arguments is the introduction of a new dynamic access capability type for call access to an entry point. Th call access capability must contain both pointers from the entry variable so that both the location to which the

call is directed and the activation record pointer loaded into PRs can be verified.

One way to implement call access capabilities is to add an entry variable identification flag to the read and write permission flags that may already appear in argument list entries. If this entry variable identification flag is set on, then the subsegment located by the argument list entry is specified to be an entry variable (or an array of entry variables if the indicated size is greater than two words) for which a dynamic access capability allowing call access should be constructed when a cross-domain call occurs. If an input argument is an entry variable (or an array of entry variables), then the corresponding argument list entry has both the read permission flag and the entry variable identification flag set on*. If the call is cross-domain, then because of the read permission flag the processor will generate a dynamic access capability allowing read access to the subsegment containing the entry variable as before, and because of the entry variable identification flag the processor will generate a capability allowing call access to the entry point located by the entry variable. (In the case of an array of entry variables a call access capability must be generated for each element in the array.) The call access capability contains the

---

*It seems a safe assumption that, when an entry variable is passed as an input argument, the calling procedure intends for the called procedure to call through the entry variable. Thus, the compiler can always generate code to set the entry variable identification flag on for input arguments that are entry variables. For entry variables that are output arguments the read and write permission flags should be set on, but the entry variable identification flag should be left off.

segment and word number portions from both the entry point pointer and the activation record pointer in the entry variable. It also contains a TAG and a SOURCE field like read/write dynamic access capabilities. The TAG field identifies the domain invocation that is the original source of the capability while the SOURCE field contains the number of that domain.*

With the addition of call access capabilities, then, CALL instructions which generate an effective pointer with a non-zero tag can be handled. When this case is encountered, the most recent DAS frame is searched for a call access capability allowing the intended call. In order for the call to be allowed, a call access capability must be found whose TAG field matches the tag from the effective pointer, whose entry point pointer segment and word number match the segment and word number in the effective pointer, and whose activation record pointer segment and word number match the segment and word number in PRs. The call must also be allowed by the static access capabilities of the SOURCE domain indicated by the matching call access capability. If the call

---

*When determining the source domain invocation for a call access capability both the tag of the pointer in the argument list entry and the tag of the entry point pointer in the entry variable must be taken into consideration. If the source is the domain invocation from which the cross-domain call is coming, then both tags will be zero. In this case the TAG field of the capability is set to the number of the domain invocation from which the call is coming and the SOURCE field is set to match. If the source is an earlier domain invocation, however, then the tag from either pointer may indicate this fact. In this case the smallest non-zero tag of the two indicates the source domain invocation number to be recorded in the TAG field of the capability. The previous DAS frame must be searched for a matching call access capability and the SOURCE field value from the matching capability copied into the new capability.

is allowed, then the CALL instruction completes in the manner discussed earlier. The value of PRs.TAG is forced to zero as the transfer occurs. (The called procedure, if the call is to an internal entry point, always wants static access to the activation record of the containing procedure block, and the tag in PRs may be set erroneously to a non-zero value for the same reason that the tag in the pointer restored to PRs before a cross-domain return erroneously is non-zero.)

Notice that a call through a cross-domain argument that is an entry variable may not be a cross-domain call, even though it is validated by matching a dynamic access capability. If the call is to an internal entry point then it may be an internal entry point of a procedure invocation executing in an earlier outstanding invocation of the calling domain. If the call is to an external entry point then it may be a gate entry point back into the calling domain, which gate is statically accessible from the source domain of the dynamic access capability that allows the call.*

---

*The processor described in this chapter does not allow procedures executing in some domain to use the dynamic access capabilities of an earlier outstanding invocation of the same domain. This restriction is required to prevent a cross-domain return from jumping several outstanding domain invocations without giving procedures that execute in these domain invocations a chance to place the encapsulated subsystems in a consistent state. A cross-domain call through an entry variable, however, can generate a legitimate need for a procedure executing in some domain to use the dynamic access apabilities (other than the return gate capability) of an earlier outstanding invocation of the same domain. It is straightforward to extend the processor to handle this case properly. The extension requires expanding the tags in pointers and dynamic access capabilities to contain one bit for each domain invocation which can exist at one time. The expanded tags allow the specification to be made that a reference be validated with respect to the dynamic access capabilities of an earlier invocation of the domain that is the domain of execution.

Label variables are very similar to entry variables, and also are normally implemented as a pair of pointers. The first pointer locates the start of the machine code corresponding to some labeled statement of a program and the second is an activation record pointer which identifies the procedure invocation which goes with that label. The goto operation on a label variable involves destroying all stacked activation records between the activation record for the procedure which performs the goto and the activation record indicated in the label variable. Along the way, the appropriate condition handlers defined in the activation records being destroyed will be invoked. When the stack is properly unwound, then a transfer to the indicated procedure segment location occurs. Because of its complexity, this non-local goto operation through a label variable is normally implemented by compiler generated code which calls upon a system-provided stack unwinding program. In the case of the processor design described here, the same approach can be used. This unwinding program is easiest to construct if it executes in domain 0 where it will have access to all activation record stack segments and the DAS. The processor, however, can be expanded a little to make the job of the unwinding program a little easier.

If a label variable is passed as an input argument on a cross-domain call then a goto operation on that label by a procedure executing in the called domain will unwind the activation record stack and the DAS back through unreturned cross-domain calls. The transfer of control to the label may change the domain of execution. By adding a label variable identification flag to argument list entries, and extending the processor to create goto access capabilities for label variables on cross-domain

calls, the processor can help the unwinding procedure to determine that an attempted cross-domain goto is allowed. The unwinding procedure merely need look in the most recent DAS frame for a matching goto access capability to determine if the attempted cross-domain goto should be performed. The goto access capability is analogous to the call access capability in the way it is constructed and the information it must contain.

Bit Addressing

For simplicity of explanation, the processor presented in this chapter is word addressed. This allows the problem of associating an explicit length with each reference to be ignored, for all references have an implied length of one or two words. When passing arguments with cross-domain calls and referencing arguments of cross-domain calls, however, the ultimate precision of specifying references by bit address and bit length can be very useful.

To see the usefulness of full bit addressing, consider passing a bit string of length 7 as an input argument on a cross-domain call. With the word addressed machine the corresponding argument list entry and dynamic access capability will specify the one or two word subsegment necessary to cover the bit string. Additional information may have to be communicated to the called procedure, either in the argument list or another argument, to specify the location of the bit string within the larger subsegment. Procedures executing in the called domain will have read access to the entire subsegment, although only a bit string of length 7 is the argument. If the string is embedded in

other information which the calling procedure wants to be certain does not become accessible to the called procedure, then the bit string must be copied into a word that contains no other valuable information before the call is made, and the copy passed as the argument. With a bit addressed machine the exact seven bits holding the string could be specified as an argument, and a dynamic access capability created giving read access to that subsegment of seven bits. No copy would have to be made to avoid giving the called procedure access to adjacent bits.

The protection techniques presented in this chapter apply equally well to a bit addressed machine. The essential changes required are extending the length of the displacement portion of a pointer to hold a bit number rather than a word number, and devising means to associate the proper length with all references. Working through the hardware mechanisms required to generate the proper length information for each reference appears to be the most difficult problem encountered when applying the techniques presented in this chapter to a bit addressed machine.

Summary

This chapter described a hardware processor that supports a multi-domain computation implemented as a single execution point in a segmented virtual memory and that automatically performs cross-domain calls. The processor recognizes static access capabilities allowing read, write, and/or execute access to segments and allowing procedures executing in one domain to call gate entry points into another domain. Dynamic access capabilities allowing read and/or write access to subsegments

representing arguments are automatically created in a processor managed dynamic access stack when a cross-domain call occurs, along with a dynamic access capability allowing access to the return gate for the call. The standard argument list of an interprocedure call is used by the processor as the specification of what capabilities to create. These capabilities are automatically destroyed when the corresponding cross-domain return occurs.

The processor is organized so that all addresses related to cross-domain arguments are automatically tagged with the number of the source domain invocation for the argument. This tag causes validation of references through such addresses to be with respect to the dynamic access capabilities currently part of the domain of execution. As long as addresses are manipulated using the EPP<u>n</u> and SPP<u>n</u> instructions, the relationship of an address to a cross-domain argument is never lost. Recursive invocation of domains and the passing of arguments through multiple consecutive cross-domain calls do not confuse the address tagging mechanism. The failure of a machine code procedure to utilize properly this and other processor protection mechanisms, however, can only compromise the integrity of the domains in which that procedure executes.

While the protection mechanisms described in the chapter may seem complex when viewed at the level of the hardware, this complexity is not apparent to the users. The users control the cross-domain call mechanisms directly from high-level programming languages. The call statement of a language like PL/I, with the simple extension that arguments be declared as input or output arguments, and the special built-in functions "window"

and "neutralize" are all that the programmer needs to direct the cross-domain call mechanisms and properly control access to arguments.

The processor developed in this chapter provides a hardware basis for a computer utility in which users may encapsulate independently compiled programs and associated data bases as protected subsystems, and then, without compromising the protection of the individual subsystems, combine protected subsystems of different users to perform various computations. The software system required to produce such a facility based on this processor is discussed in the next chapter.

CHAPTER 4

SOME COMMENTS ON THE SUPPORTING SOFTWARE

In this chapter a software second layer for the processor protection mechanisms developed in Chapters 2 and 3 is briefly discussed. The purpose of the material in this chapter is not to provide a detailed development of the complex issues involved in providing a proper user interface for controlling sharing in a computer utility. Rather, it is simply to demonstrate one way of harnessing the processor protection mechanisms so that users can define and control the sharing of protected subsystems.

As stated in Chapter 1, the goal of the research reported in this thesis is to develop a computer utility in which mutually suspicious subsystems can cooperate in a single computation in an efficient and natural way. A specific objective derived from this goal is that a programming environment be provided in which procedure segments, data segments, and protected subsystems of all users can be treated as building blocks and combined in a controlled way into different computations. The starting point of the research was Multics, which to a large extent already provides the building block facility with respect to procedure and data segments, but which allows protected subsystems to be defined and manipulated only in restricted ways. The primary obstacle to providing a general facility for manipulating protected subsystems in Multics is the restriction that the domains of a process be totally ordered with respect to contained access capabilities. The

processor discussed in Chapters 2 and 3 removes this restriction.

With the primary obstacle out of the way, only a few aspects of the Multics software system need to be extended to produce a first approximation to a software system which can effectively apply the hardware protection mechanisms to generate an environment in which protected subsystems may be manipulated in a natural way by all users. The most significant changes required are the expansion of the file system to catalogue protected subsystems as well as segments and the recasting of control lists in terms of protected subsystem names rather than domain numbers. These extensions are outlined in this chapter.

Because this chapter is intended only to provide an example of how the processor protection mechanisms can be harnessed by a software system, little justification or detail is provided for those aspects of the software that are derived directly from Multics. The reader is referred to the book by Organick [34] for a detailed discussion of the Multics facilities outlined here.

A Distributed Supervisor

The distributed supervisor organization used in Multics is easily implemented on the new processor. When a user logs in to the system a process with a new virtual memory is created for him. Initially two domains of the process are occupied. Domain 0 encapsulates the supervisor.* Thus, the supervisor is a protected subsystem that is part of

---

*Domain 0 is chosen because of the special treatment afforded it by the processor. In Multics, the supervisor is encapsulated in the largest of the linearly nested domains of a process.

every process. Another domain encapsulates a "home" protected subsystem for the user. When the user first gains control of the process it is executing in this home domain in some procedure which interprets as commands characters input from his terminal.

Because the supervisor is a protected subsystem that is part of the process, the supervisor can be invoked by standard interprocedure calls to the gate entry points into domain 0 to perform various services for the process. (It may also be invoked by generating an exception, for an exception forces control into domain 0.) Among the services of interest here are adding various segments stored on-line to the virtual memory of the process and associating various protected subsystems whose definitions are stored on-line with unoccupied domains of the process.

The File System

In Multics all segments stored on-line are catalogued by a file system which is implemented as a set of procedure and data segments in the supervisor. The single, system wide catalogue is arranged as a tree-structured directory hierarchy. Each directory contains entries giving the attributes of all directly inferior directories and segments. The attributes in each entry include a name, a length, a beginning absolute address,* and an access control list. The name in a directory entry uniquely identifies the entry in that directory. Each directory

---

*Directories are implemented as data segments which can be read and written only from the supervisor domain.

or segment can be uniquely located in the tree-structured directory hierarchy by a multicomponent tree name consisting of an ordered list of directory entry names defining a path from the base of the tree -- the "root" directory -- to the item of interest. Figure 4-1 illustrates a directory hierarchy and gives the tree names of the items shown. The symbol ">" is used as a separator for the components of a tree name.

The directory hierarchy allows a content-related organization to be imposed on all information stored on-line in the system. The structure of the initial layers of the hierarchy usually will be quite static and will be known by most system users. Figure 4-2 illustrates a structure similar to that used in Multics to organize on-line storage. This structure reflects the fact that the people who use the system are grouped for administrative purposes into various projects. (A user name in this case is the concatenation of a project name and a person name.) Each user has a home directory in which to create an arbitrary substructure of segments and directories. Each existing process has a directory in which to place all its temporary data segments such as its descriptor segment, stack segments, etc. Other areas of the hierarchy catalogue library and system segments.

There are a variety of operations which may be performed on the directory hierarchy by calling the appropriate supervisor gate entry point. Examples are creating or deleting an object in some directory, changing the attributes of an existing object, listing the entry names in a directory, and adding a segment to the virtual memory of the

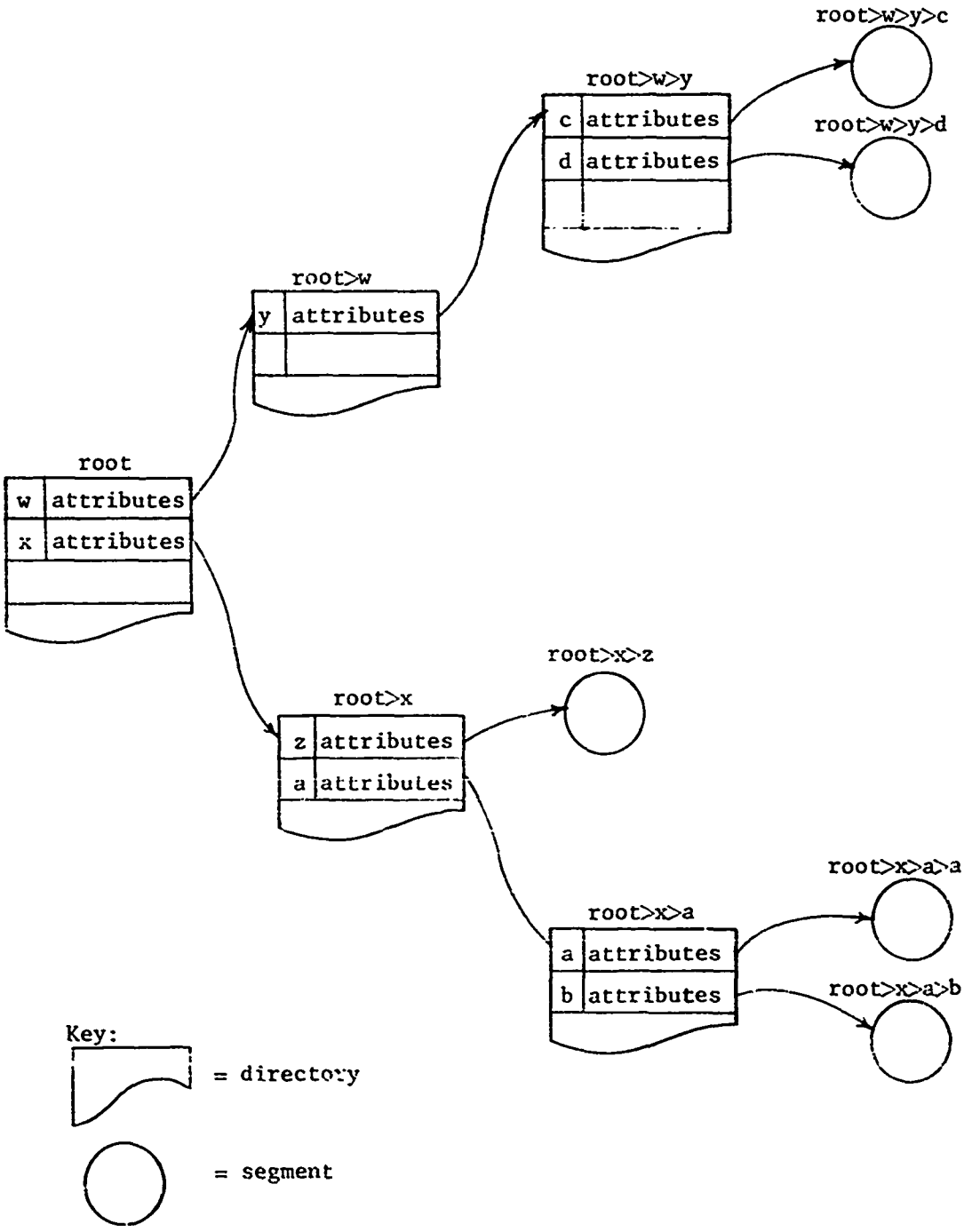


Figure 4-1: A representative directory hierarchy.

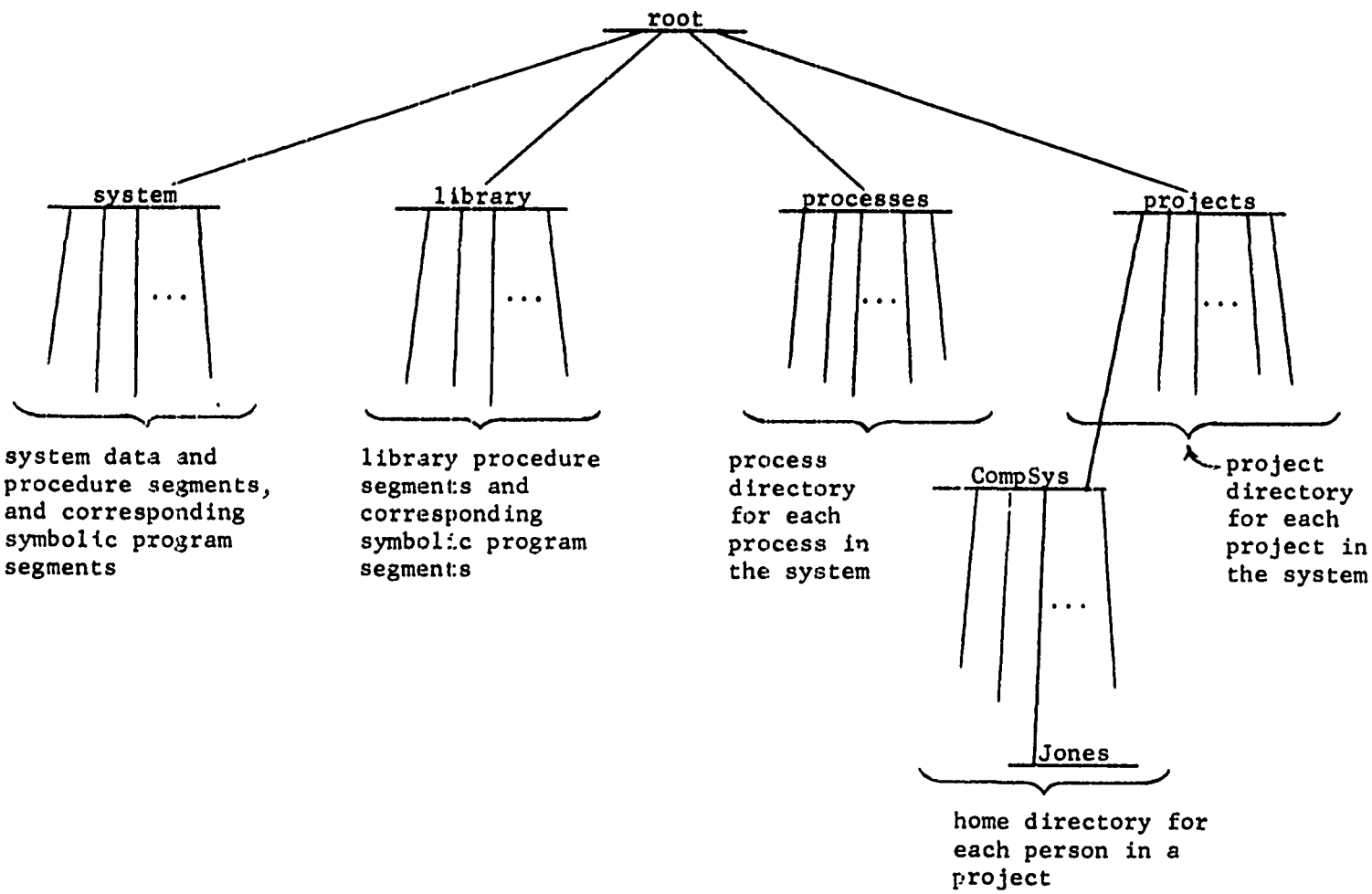


Figure 4-2: Use of the directory hierarchy to organize information stored on-line.

process.* Whether or not a particular requested operation on some object will be performed is controlled by the access control list (ACL) of the object. The supervisor compares the name of the user associated with the process being executed and the number of the domain from which the request is made with the ACL of the object. If some ACL element containing a matching name and number is found, then the various permission flags set in that element determine whether or not the requested operation will be performed. If no matching element is found then the requested operation will not be performed. Each ACL entry essentially defines some capabilities included in the specified domain of a process of the specified user. The permission flags present determine the particular kinds of access allowed by the capabilities. The flags that may appear in an element of a directory ACL provide independent control on operations like listing the entries in that directory, creating new objects in that directory, and changing certain attributes recorded in entries in that directory. The last case includes changing the ACL's in these entries. The permission flags that may appear in an element of a segment ACL can indicate read, write, execute, and gate access to that segment.

The flaw in this access specification scheme used by Multics, when applied to controlling the sharing of protected subsystems with the new processor, is that actual domain numbers appear in ACL elements. The implication of this fact is that a particular domain of a process of a

---

*In Multics this last operation is normally caused by a dynamic linking exception rather than an explicit call to the supervisor.

particular user always protects the same protected subsystem. An extension that eliminates this flaw is to make protected subsystems named objects that are catalogued in the directory hierarchy and to replace the domain number in an ACL element with the tree name of a protected subsystem. Then any protected subsystem may be associated with any domain of a process as the need arises. The numbered domains of a process are allocated to protected subsystems just as segment numbers in the virtual memory of the process are allocated to segments stored on-line. With this revised scheme the access of procedures executing in some domain of a process is determined by the name of the protected subsystem associated with the domain and the name of the user associated with the process, rather than the domain number and the user name. This approach is now explored in a little more detail.

Protected Subsystems

A protected subsystem is a third type of object which may be defined by a directory entry. The attributes of a protected subsystem are a name and an ACL. A protected subsystem has no content, so attributes such as length and address are not required. The ability to create a new protected subsystem in some directory or delete a protected subsystem from a directory is controlled by the ACL of the directory, as it is for the other two types of objects which may be defined by a directory entry.

The ACL for a directory, segment, or protected subsystem is a list of two-component elements of the following form:

<protected subsystem tree name>:<user name> <mode>

The first component identifies an instance of use of a particular protected subsystem in a process of a particular user. The second component is some set of permission flags appropriate for the object type.

By means to be discussed in a moment it is possible to associate a protected subsystem with one of the numbered domains of a process, thereby generating a protected subsystem instance. The supervisor maintains for each process a table of associations of domain numbers and protected subsystem tree names. This domain table can also record the name of the user associated with the process. When a process is first created, the domain table might appear as illustrated in Figure 4-3. Domain 0 is occupied by an instance of the supervisor protected subsystem and domain 1 is occupied by the home protected subsystem instance for the user. (A directory hierarchy similar to that shown in Figure 4-2 is assumed.)

When a procedure executing in some domain of a process calls a gate entry point into the file system, the requested operation is validated by comparing the protected subsystem name associated with the requesting domain and the user name associated with the process against the appropriate ACL. The mode in the first matching ACL element determines if the request will be performed. If no ACL element matches, then the request is not performed.

Permission Flags for Segments

To understand how a protected subsystem can become associated with a domain of a process, it is necessary to investigate in more detail the use of the modes which may be specified in an element of an ACL for a

domain number	occupied	protected subsystem tree name
0	yes	root>system>supervisor
1	yes	root>projects>CompSys>Jones>home
2	no	-
3	no	-
4	no	-
5	no	-
6	no	-
7	no	-
User name is "CompSys.Jones"		

Figure 4-3: Initial state of the domain table for a process.

segment. As indicated earlier, these modes may specify read, write, execute, and gate permission. When a procedure executing in some domain explicitly or implicitly requests of the file system that a segment be made accessible from that domain, the matching ACL element determines which permission bits in the corresponding descriptor segment entry (DSE) will be set on. For example, if the segment is or was previously assigned segment number s in the virtual memory of the process, if the request is from domain n, and if the permission flags in the matching ACL element indicate read and execute access, then DSEs.Rn and DSEs.En are set on by the supervisor, and DSEs.Wn and DSEs.Gn are set off.

The case when the matching ACL entry indicates gate permission is a little more complex. Continuing with the previous example, in this case only DSEs.Gn is set on by the supervisor. However, DSEs.GATES and DSEs.DOMAIN must also be filled in. Recall that these fields indicate the number of gate entry points in the segment and the number of the domain for which they are gates, respectively. As the information source for these fields, two new attributes for a segment are introduced. The first attribute, which may or may not apply to a given segment, says: "this segment is a gate into protected subsystem <protected subsystem tree name>", and is called the gate attribute. The second attribute, which is only meaningful when the first is applied, says: "there are m gate entry points in this segment". How these attritubes are used is obvious. If gate permission applies, then the supervisor determines if the protected subsystem named by the gate attribute is already associated with some domain of the process. If so, then the corresponding domain number derived from the domain table for the

process is used to set DSEs.DOMAIN. If not, then that protected subsystem is assigned an unoccupied domain of the process, and the number of this domain is used to set DSEs.DOMAIN. DSEs.GATES is set directly from the second attribute. Thus, a protected subsystem is associated with a domain of a process when the first attempt is made to get at a segment containing gates into that protected subsystem.

If all eight of the domains of a process are occupied at the time when a new protected subsystem is to be associated with the process, then it is necessary to free an occupied domain to make room. The choice of which domain to free, i.e., which protected subsystem instance to remove from the process, is best made by the user associated with the process or a program provided by him. With the reasonable restriction that the domain to be freed have no outstanding invocations, the designated domain is easily freed by turning off all the permission bits in the corresponding column of the descriptor segment, turning off all gate permission bits in other columns that correspond to gates into the freed domain, and appropriately updating various supervisor data bases. The execution environment of the freed domain must also be returned to the unused state. Reusing domains is much easier than reusing DSE's. The reason is that segment numbers diffuse into non-supervisor-controlled data segments as part of stored pointers, making it very difficult to locate all instances of pointers referring to the segment previously associated with a reused DSE. Domain numbers are not part of stored pointers, however, and appear only in supervisor-controlled data segments.

Controlling the Gate Attribute

In general, given that a particular protected subsystem instance has permission to modify the ACL of an object at all, there need be no finer control on the particular modification. Any element may be deleted from the ACL and any element may be added. There need be no constraint on the protected subsystem instance named or the mode specified in an added element. (To keep things consistent the supervisor might enforce the rule that the specification of gate permission in an ACL element for a segment can be made only if the gate attribute is present for the segment.) The ability to set the gate attribute for a segment, however, needs to be carefully controlled. In particular, if a procedure executing in a protected subsystem instance which has proper permission to change the attributes of some segment could specify a gate attribute for that segment naming an arbitrary protected subsystem, then no protected subsystem would be secure. Any user could create, say from his home protected subsystem instance, a gate into any other protected subsystem and thereby cause a procedure he constructed to execute in any other protected subsystem. The ability to create gates into a protected subsystem needs to be restricted to those responsible for the protected subsystem (or their designates). The ACL on a protected subsystem provides this control.

There need be only one kind of permission flag that can appear in an ACL element of a protected subsystem: the "define gates" permission flag. In order for some protected subsystem instance to set the gate attribute for some segment to specify that the segment contains gates into a protected subsystem with tree name T, a matching element on the

ACL of T must give the requesting protected subsystem instance "define gates" permission. Thus, in addition to providing a tree name for a protected subsystem, a protected subsystem object in the directory hierarchy controls the creation of gates into the protected subsystem.

Examples of Access Control Lists

A few examples will help to sharpen the meaning and use of ACL's on protected subsystems and segments. To start with, imagine that each system user has a two-part project/person name, that a home directory with the tree name "root>projects>PROJECT_NAME>PERSON_NAME" is provided for each user, and that a home protected subsystem with the name "root>projects>PROJECT_NAME>PERSON_NAME>home" is provided for each user. Only the home protected subsystem instance of the user and the home protected subsystem instance of his project administrator have permission to perform operations on the user's home directory, including modifying the ACL's of directly inferior segments, directories, and protected subsystems.

The most frequently used ACL elements by a user will be those giving access to his home protected subsystem instance. Following the assumptions just outlined, such ACL elements for the user "CompSys.Jones" would be expressed as:

root>projects>CompSys>Jones>home:CompSys.Jones <mode>

This specification will be so frequently used (even in the examples in this section) and is so awkward that a shorthand notation is adopted to express the home protected subsystem instance of a user. It is just the user's name preceeded by a colon. Thus the ACL element:

:CompSys.Jones <mode>

is equivalent to the previous expression.* A data segment that this user creates for his own use, then, would bear the ACL element:

:CompSys.Jones read, write

while a procedure segment created for his own use would bear the ACL element:

:CompSys.Jones read, execute

This procedure segment could be shared with his friend "Smith" in the same project by adding to its ACL the element:

:CompSys.Smith read, execute

Now imagine that user "CompSys.Jones" wishes to create a protected subsystem that encapsulates one procedure segment containing gate entry points and one data segment. He will name this protected subsystem "ex_ps" and put it in his home directory. Thus, its tree name is "root>projects>CompSys>Jones>ex_ps". The ACL of this protected subsystem is set with the single entry:

:CompSys.Jones define gates

which gives user "CompSys.Jones" permission to define gates for the protected subsystem when executing in his home protected subsystem instance. The procedure segment is then given a gate attribute for the protected subsystem "root>projects>CompSys>Jones>ex_ps". Initially, the ACL of the procedure segment has two elements. They are:

---

*In a real implementation conventions also would be required to shorten the expression of other commonly used protected subsystem instances in ACL elements. No further conventions will be introduced here, however.

:CompSys.Jones read, execute

root>projects>CompSys>Jones>ex_ps:*.* read, execute

These elements allow execution of the procedure segment in his home protected subsystem instance and in any* instance of the "ex_ps" protected subsystem. Likewise, the data segment gets the ACL:

:CompSys.Jones read, write

root>projects>CompSys>Jones>ex_ps:*.* read, write

allowing read/write access to the segment from the home protected subsystem instance of the user and from any instance of the "ex_ps" protected subsystem. So far, no user can borrow this protected subsystem.

Suppose now that user "CompSys.Smith" wants to borrow this protected subsystem. User "CompSys.Jones" can permit this by adding the element:

:CompSys.Smith gate

to the ACL of the procedure segment containing gates into the protected subsystem. This gives user "CompSys.Smith" permission to call these gate entry points from his home protected subsystem instance and establish an instance of the protected subsystem "ex_ps" in his process.

This simple example illustrates the general pattern for defining and controlling the sharing of a protected subsystem. All the procedure and data segments of the protected subsystem are made directly accessible in appropriate ways from any instance of the subsystem. The sharing of the protected subsystem is then controlled by giving other protected subsystem instances gate access to the gate segments.

The user name ".*" means any project and any user. A match anything character such as "*" is very useful for constructing ACL elements which match a group of protected subsystem instances.

The pattern for controlling the sharing of protected subsystems allows the access of one protected subsystem instance to another to be revoked in a particularly smooth way by removing the ACL entries giving the former gate access to the gate segments of the latter. If the latter happens to be executing at the time this gate access is revoked, execution is not stopped instantaneously. Rather, execution continues until the cross-domain return terminates the activities of the protected subsystem instance in an orderly fashion. Thus, the protected subsystem instance is not stopped executing at an awkward moment leaving component data bases in an inconsistent state. Because of the revoked gate access, however, further cross-domain calls back into the protected subsystem instance by the former will not be allowed.

Other Issues

The decision to provide general support for user-defined protected subsystems in a computer utility impacts other areas of the software system besides the file system. One of the most important is the execution environment provided by the system for procedures. A procedure segment that can be used as a building block is of necessity pure and may contain ambiguous unresolved references to other procedure and data segments. In order to execute, the procedure segment must be provided with an impure appendage in which to store linkage information and static variables, a stack segment in which to allocate activation records, and information such as search rules [4] with which to resolve the ambiguous references to other procedure and data segments. It is

essential that all these aspects of the execution environment which can affect the operation of a procedure be provided by the system on a per domain basis, and that a standard initial execution environment be defined which is guaranteed by the supervisor to exist in any domain of a process. Thus, when a protected subsystem instance is associated with a domain of some process, the procedures of that protected subsystem can know what execution environment will be provided initially and therefore can be constructed to operate correctly in that environment.

Multics already meets this requirement. I/O stream name definitions and attachments, reference name definitions, and search rule definitions are limited in scope to a single domain of a process and always given a system-defined initial state. A stack segment and a segment for linkage information are provided for each domain of a process and are given system-defined initial states.

Other system functions which affect the secure operation of a multidomain computation include error detection and handling (for example, the PL/I condition mechanisms), process interruption via the attention key of the user's terminal, input/output, and accounting. The mechanisms in Multics for performing these and other functions, with minor revisions, appear to be adequate for supporting multidomain computations.

Finally, in order for the procedures of a protected subsystem to make meaningful decisions on whether or not to fulfill requests received in the form of cross-domain calls, it must be possible for these procedures to determine the identity of the protected subsystem instance

making the request. For this purpose a supervisor gate should be provided which allows a procedure executing in some domain to determine the tree name of the protected subsystem instance from which the last incoming cross-domain call was made. It may be appropriate to provide a supervisor gate which essentially makes the contents of the domain table of a process available to procedures executing in any domain of a process.

Summary

This chapter has presented a brief description of a software second layer for the processor protection mechanisms described in Chapters 2 and 3. The purpose of this material is to demonstrate one way to harness the processor protection mechanisms so that users can define and control the sharing of protected subsystems. There are many degrees of freedom in constructing a software system to harness the hardware protection mechanisms. Further research may produce a software system organization that provides a more simple, natural, and flexible user interface for controlling sharing in a computer utility than the interface developed in this chapter. The software system outlined here, however, demonstrates that a practical protection facility meeting the objectives presented in Chapter 1 can be constructed based on the hardware protection mechanisms developed in this thesis.

CHAPTER 5

CONCLUSIONS

This thesis has described protection mechanisms for a computer utility which, in addition to allowing controlled sharing of programs and data among users, allow controlled sharing of user-defined protected subsystems. These mechanisms are practical to implement using the hardware and software technology available today, and if implemented would provide a practical and sophisticated facility for controlling information sharing in an operational computer utility.

The unique aspect of these protection mechanisms is the efficient, natural, and general manner in which they support the controlled sharing of user-defined protected subsystems. A protected subsystem is a collection of programs and data bases encapsulated so that other executing programs can invoke certain component programs within the protected subsystem, but are prevented from reading or writing component programs and data bases, and are prevented from disrupting the intended operation of the component programs. The ability to define and share protected subsystems provides users with a flexible and powerful means for controlling access to stored information. By defining appropriate protected subsystems, for example, users can implement complex controls on access to data bases, share proprietary algorithms without divulging the structure of the algorithms, or limit the damage that borrowed programs can do to the programs and data bases of the borrower.

In the first example, by encapsulating various data bases in a protected subsystem with programs that he provides, a user can force all references to the data bases attempted by programs outside the protected subsystem to be made by invoking these caretaker programs. The user can specify arbitrarily complex controls on access to the encapsulated data bases since he writes the programs that judge, perform, and record requested references. Providing users with the ability to write programs for controlling access to data bases has an important impact on the structure of the computer utility. It means that the system need not provide protection mechanisms that directly implement complex, content-dependent controls on access to stored information. Rather, the system only need implement the general mechanisms required for users to define the encapsulation of programs and data bases in protected subsystems and for users to control the sharing of protected subsystems. More complex controls on access to stored information than are directly provided by the system can be implemented by the users themselves by encapsulating caretaker programs with data bases in protected subsystems. The result is that arbitrarily complex control on access to stored programs and data can be implemented based on a general, fixed, fairly simple set of system-provided protection mechanisms.

The second example use for protected subsystems, sharing proprietary algorithms without divulging the structure of the algorithms, takes advantage of the fact that programs in a protected subsystem may be invoked by outside programs but not read by them. In addition, none of the temporary or permanent data associated with the programs in the protected subsystem may be read or written by outside programs. Thus, the programs

in a protected subsystem may be used by other programs, but no aspect of their structure examined.

The third example use for protected subsystems, limiting the damage that borrowed programs can do to the programs and data of the borrower, is based on the fact that programs inside a protected subsystem can invoke programs outside without compromising the protected subsystem. Thus, a borrower only need arrange that his programs and data form a protected subsystem and that a borrowed program not be part of that protected subsystem. Then, if by malice or error the borrowed program behaves in an unexpected way, it will not be able to damage the programs and data of the borrower.

The protection mechanisms described in this thesis for supporting user-defined protected subsystems allow multiple, independent protected subsystems to be combined in a single computation without compromising the protection of the individual subsystems. The mechanisms are based on the division of a computation into independent domains of access privilege, each of which may encapsulate a protected subsystem. The central component of these mechanisms is a hardware processor that fully supports a multidomain computation implemented as a single process whose address space is a segmented virtual memory. This processor allows a standard interprocedure call with arguments to change the domain of execution of the process. All the semantic implications of an interprocedure call as defined in a high-level programming language like PL/I are accomodated on a cross-domain call without compromising the protection of the subsystems encapsulated by either of the domains involved. The processor design represents the first time that hardware mechanisms for fully

automating cross-domain calls between independent domains of a computation have been described.

The processor recognizes and enforces two kinds of access capabilities: static and dynamic. Static access capabilities represent the encapsulation of segments of procedure and data in protected subsystems and control the ability of procedures executing in one protected subsystem to invoke procedures of another via a cross-domain call. Dynamic access capabilities correspond to the arguments of cross-domain calls. The processor automatically adds to the called domain dynamic access capabilities to reference whatever arguments accompany a cross-domain call and automatically removes these capabilities whenever the subsequent return occurs. The distinction between static and dynamic access capabilities corresponds to an apparently fundamental division of the access privilege associated with a process that executes in a multidomain environment into the capabilities associated with the address space component of the process (a segmented virtual memory in this case) and the capabilities associated with the execution point component of the process.

The processor is the key to both the efficiency and the naturalness of the protection mechanisms. Because the processor automatically validates each reference made by an executing program against the static or dynamic access capabilities in the domain of execution and automatically performs cross-domain calls, little intervention by supervisor software is required for the enforcement of the access constraints associated with a multidomain computation as that computation executes. Because on a cross-domain call the processor is able to interpret the standard

argument list and create in the called domain dynamic access capabilities that exactly correspond to the arguments, the user has the natural interface to the cross-domain call mechanism of the call and return statements in a high-level programming language.

In addition to the hardware processor, supporting software was discussed. This software, when used in conjunction with the hardware protection mechanisms, provides an environment in which users of the computer utility can construct and share procedure segments, data segments, and protected subsystems. This protection facility extends the idea inherent in programming generality of using programs as building blocks to allow protected subsystems to be used as building blocks as well. Independently written and compiled procedure segments may be combined in different ways along with the associated data segments to form protected subsystems without recompiling the procedure segments. Independently defined protected subsystems may be combined in different ways as cooperative partners in various computations without either modifying the protected subsystems or altering the component procedure segments. The ease with which procedure segments, data segments, and protected subsystems provided by different users can be combined to perform different computations will encourage the users of the computer utility to communicate, cooperate, and build upon one another's work.

Taken together, the hardware and software mechanisms described in this thesis constitute an existence proof of the feasibility of building protection mechanisms for a computer utility that allow multiple user-defined protected subsystems, mutually suspicious of one another, to cooperate in a single computation in an efficient and natural way.

Areas for Further Research

The topic of protection mechanisms for computer systems is by no means exhausted as a viable research topic. In this final section of the thesis two of the many areas for further research are discussed briefly. The first is the user interface for controlling sharing in a computer utility. As indicated in Chapter 4, further research may produce a software system organization that provides a more simple, flexible, and natural user interface for controlling sharing in a computer utility than that described in this thesis. Making the user interface as simple and natural as possible without sacrificing needed flexibility is very important. Experience with Multics suggests that users not accustomed to dealing with sophisticated protection mechanisms frequently make errors when specifying to the system the access to be allowed to some object. Even after some skill is developed in applying the protection mechanisms, users find it hard to be sure that specifications given to the system to control the access to some object result in the intended pattern of access being allowed. For example, it is very hard to detect the accidental granting of too much access to some object, because no obvious errors result from too much access being granted. When dealing with user-defined protected subsystems the problem probably becomes worse, for access specifications become embedded in programs that users construct.

To make more progress on this important problem of human engineering, it may be necessary to employ an actual system with a creative user community as a tool. Only by applying a set of protection mechanisms to the protection needs of real users can it be seen where user mistakes

commonly are made, and can it be seen what aspects of the flexibility provided are really needed and what aspects complicate the mechanisms with unneeded flexibility. It would be feasible to construct a prototype of the system proposed in this thesis to use as such a tool. Because of the close relationship of the overall architecture of the processor described here to the architecture of the new Multics processor, only a few of the major functional modules of that processor would have to be altered to produce a processor that implements most of the protection mechanisms described in Chapter 3. Only a few changes are required to the existing Multics software to produce the software system described in Chapter 4. In addition to testing user reaction to the protection facilities provided, this prototype system would provide an environment in which to test improved interfaces and to develop tools that allow users to debug the access specifications embodied in protected subsystems or given to the system-provided protection mechanisms.

The second area for further research is that of certification. Certification means guaranteeing that the protection mechanisms in a system actually implement the intended protection facility, i.e., that no mistakes were made during the design or construction of system protection mechanisms that allow these mechanisms to be circumvented. The essential problem is that there exist no methodical techniques for designing or constructing systems so that this guarantee can be made. In addition, the protection mechanisms devised to date, especially their software components, are much too complex to certify by inspection once constructed.

Protection hardware for supporting multidomain computations, such as that described in this thesis, offers some help in attacking the problem of certifying the software components of the protection mechanisms in a system. The hardware can be applied to isolate by encapsulation in protected subsystems those software components that are critical to the correct function of the protection mechanisms. The key to success, and the area where further research is required, is making small and simple those software components which, if improperly designed or implemented, or if violated, would compromise the protection mechanisms of the system. Once made small and simple, and their correct operation certified, the hardware can provide enforced isolation for these components, thus insulating them from errors introduced into the system by modifying other, less critical components of the software.

The two areas for further research mentioned above are only a small sample of the work that remains to be done on the topic of protection mechanisms for computer systems. One of the basic problems to be overcome is that there exists such a lack of formal knowledge on the topic that it is difficult to see the problems that remain unsolved. As computer systems play larger and more important roles in society, protection mechanisms will become increasingly important as the guardians of privacy. The design and implementation of protection mechanisms must become so well understood and methodical that, as a matter of course, commerically available computer-based information systems contain protection mechanisms which prevent unauthorized access to the information stored in them while at the same time make it easy for users to communicate when appropriate by sharing information. We have a long way to go to reach this goal.

BIBLIOGRAPHY

References, in alphabetic order:

[1] Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics Virtual Memory: Concepts and Design", Comm. ACM 15, 5 (May 1972), 308-318.

[2] Brinch-Hansen, P., "RC-4000 Software Multiprogramming System", A/S Regnecentralen, Copenhagen, April 1969.

[3] Burroughs Corporation, "The Descriptor - a definition of the B5000 Information Processing System", Detroit, Mich., February 1961.

[4] Clingen, C.T., "Program Naming Problems in a Shared Tree-Structured Hierarchy", In working papers distributed at NATO Science Comm. Conf. on Techniques in Software Eng., Rome, Italy, October 1969.

[5] Corbató, F.J., Clingen, C.T., and Saltzer, J.H., "Multics: The First Seven Years", Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., 571-583.

[6] Corbató, F.J., "PL/I as a Tool for System Programming", Datamation 15, 6 (May 1969), 68-76.

[7] Dennis, J.B., and Van Horn, E.C., "Programming Semantics for Multiprogrammed Computations", Comm. ACM 9, 3 (March 1966), 143-155.

[8] Dennis, J.B., "Programming Generality, Parallelism, and Computer Architecture", Information Processing 68, North-Holland Publishing Co., Amsterdam, 1969, 484-492.

[9] Dennis, J.B., "Future Trends in Time-Sharing Systems", Time-Sharing Innovation for Operations Research and Decision Making, Washington Operations Research Council, Rockville, Maryland, 1969, 229-235.

[10] Dennis, J.B., "Modularity", Computation Structures Group Memo 70, M.I.T. Project MAC, June 1972.

[11] 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.

[12] Fabry, R.S., "Preliminary Description of a Supervisor for a Computer Organized around Capabilities", Quarterly Progress Rep. No. 18, Institute of Comp. Research, Univ. of Chicago, I-B 1-97.

[13] Graham, G.S., and Denning, P.J., "Protection - Principles and Practice", Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., 417-429.

[14] Graham, R.M., "Protection in an Information Processing Utility", Comm. ACM 11, 5 (May 1968), 365-369.

[15] Gray, J., et al, "The Control Structure of an Operating System", to be published.

[16] Honeywell Information Systems, Inc., "The Multics PL/I Language", Waltham, Mass., July 1972.

[17] Honeywell Information Systems, Inc., "Model 645 Processor Reference Manual", Cambridge Information Systems Laboratory, Cambridge, Mass., April 1971.

[18] Lampson, B.W., "On Reliable and Extendable Operating Systems", Techniques in Software Engineering, NATO Science Committee Workshop Material, Vol. II, September 1969.

[19] Lampson, B.W., "Dynamic Protection Structures", Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., 27-38.

[20] Lampson, B.W., "Protection", Proc. Fifth Annual Princeton Conf. on Information Sciences and Systems, Dept. of Elec. Eng., Princeton Univ., Princeton, N.J., March 1971, 437-443.

[21] M.I.T. Project MAC, "Multics Programmers' Manual", 1969.

[22] Motobayashi, S., Masuda, T., and Takahashi, N., "The Hitac 5020 Time-Sharing System", Proc ACM 24th Nat. Conf., ACM, New York, 1969, 419-429.

[23] Organick, E.I., The Multics System: An Examination of its Structure, M.I.T. Press, Cambridge, Mass., 1972.

[24] Schroeder, M.D., "Performance of the GE-645 Associative Memory while Multics is in Operation", ACM Workshop on System Performance Evaluation, ACM, New York, April 1971, 227-245.

[25] Schroeder, M.D., and Saltzer, J.H., "A Hardware Architecture for Implementing Protection Rings", Comm. ACM 15, 3 (March 1972), 157-170.

[26] Univ. of California, "CAL-TSS Internals Manual", Computation Center, Berkeley, Calif., November 1969.

[27] Univ. of California, "CAL Time-Sharing System Users Guide", Computation Center, Berkeley, Calif., November 1969.

[28] Vanderbilt, D.H., "Controlled Information Sharing in a Computer Utility", M.I.T. Project MAC, MAC-TR-67, 1969.

[29] Wilkes, M.V., Time-Sharing Computer Systems, American Elsevier, 1968.

BIOGRAPHICAL NOTE

Michael David Schroeder was born in Richland, Washington on January 5, 1945. He attended public school there, graduating from Columbia High School in June, 1963. He entered Washington State University in September, 1963 where he studied Mathematics and Computer Science, receiving the degree of B.A. with highest honors (February, 1967). In September, 1967 he entered the Massachusetts Institute of Technology where he studied Computer Science, receiving the degrees of S.M. (February, 1969) and E.E. (June, 1969).

Mr. Schroeder joined the staff of the M.I.T. Electrical Engineering Department in June, 1968 as a Teaching Assistant; in June, 1970 he became an Instructor. He has been involved in developing and teaching a course on the structure of computer-based information systems. In July, 1967 he became associated with Project MAC where he has been involved in the Multics development effort. Research conducted at Project MAC in protection mechanisms for computer systems was the subject of his doctoral dissertation.

Mr. Schroeder is a member of Phi Beta Kappa, Phi Sigma Phi, Sigma Xi, the IEEE, and the ACM.

Publications

"The Classroom Information and Computing Service", Project MAC Technical Report MAC-TR-80, January, 1971 (with D.D. Clark, R.M. Graham, and J.H. Saltzer).

"Performance of the GE-645 Associative Memory while Multics is in Operation", ACM Workshop on System Performance Evaluation, ACM, New York, April 1971, 227-245.

"A Hardware Architecture for Implementing Protection Rings", Communications of the ACM 15, 3 (March 1972), 157-170 (with J.H. Saltzer).