A SECURE AND FLEXIBLE MODEL OF PROCESS INITIATION FOR A COMPUTER UTILITY

Warren Alan Montgomery

June 1976

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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

Acknowledgements

I would like to thank some of the people who helped in this research. I would like to thank professor Saltzer, my thesis supervisor, for help in defining the topic and guidance throughout the project. Many of the members of the Computer Systems Research group gave helpful suggestions as the ideas for this thesis were beginning to form. Ken Pogran and Doug Wells of the Computer Systems Research group, and Paul Green of Honeywell Information Systems provided great assistence in the design and debugging of the test implementation. Professor M. D. Schroeder, and Dr. D. Clark provided many valuable suggestions on drafts of some of the sections of the thesis. Most of all, I would like to thank my wife, Carla, for inspiration throughout the project, for help in preparing the thesis, and for patience during three long years of graduate study.

I would also like to thank the National Science Foundation (NSF) for funding for graduate study under the NSF Graduate Fellowship Program.

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

A SECURE AND FLEXIBLE MODEL OF PROCESS INITIATION FOR A COMPUTER UTILITY*

by

Warren Alan Montgomery

ABSTRACT

This thesis demonstrates that the amount of protected, privileged code related to process initiation in a computer utility can be greatly reduced by making process creation unprivileged. The creation of processes can be controlled by the standard mechanism for controlling entry to a domain, which forces a new process to begin execution at a controlled location. Login of users can thus be accomplished by an unprivileged creation of a process in the potential user's domain, followed by authentication of the user by an unprivileged initial procedure in that domain.

The thesis divides the security constraints provided by a computer utility into three classes: Access control, prevention of unauthorized denial of service, and confinement. We develop a model that divides process initiation into five independent functions: Process creation, domain changing, resource control, authentication, and environment initialization. We show which classes of security constraints depend on each of these functions and show how to implement the functions such that these are the only dependencies present.

The thesis discusses an implementation of process initiation for the Multics computer utility based on the model. The major problems encountered in this implementation are presented and discussed. We show that this implementation is substantially simpler and more flexible than that used in the current Multics system.

---

*This report is based upon a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, on May 13, 1976 in partial fulfillment of the requirements for the degrees of Master of Science and Electrical Engineer.

Table of Contents

SECTION	PAGE
ABSTRACT	3
TABLE OF CONTENTS	4
LIST OF FIGURES	6
CHAPTER 1. INTRODUCTION	7
1.1 The Problem	7
1.2 Method of Attack	8
1.3 Results	10
1.4 Thesis Plan	12
1.5 Related Work	13
CHAPTER 2. A MODEL FOR PROCESS INITIATION	15
2.1 Security Goals	15
2.2 A Layered Security Kernel	20
2.3 A Model for Process Initiation	22
CHAPTER 3. AUTHENTICATION	27
3.1 Properties of Authentication Mechanisms	27
3.2 Authentication Forwarding	33
3.3 Example of our Authentication Scheme	36
CHAPTER 4. RESOURCE CONTROL	39
4.1 Issues of Resource Control	39
4.2 Primitive Operations for Resource Control	42
4.3 Limitations on Resource Control Policy	46

4.4 Limitations on Security Constraints 46

CHAPTER 5. MECHANISMS FOR AUTHORIZING DOMAIN CHANGES 48

5.1 Introduction to Domain Changing 48

5.2 Four Mechanisms for Authorizing Domain Changes 49

5.2.1 Exact Specification 50

5.2.2 Partial Specification 54

5.2.3 Last Component Specification 58

5.2.4 Appending Specification 60

5.3 Domain Changing and Confinement 63

5.4 Evaluation of Domain Changing Mechanisms 68

CHAPTER 6. THE TEST IMPLEMENTATION 70

6.1 Brief Introduction to Multics 70

6.2 The Implementation 75

6.3 Evaluation of the Test Implementation 88

CHAPTER 7. EVALUATION AND CONCLUSIONS 93

7.1 Comparison of the Model to Other Schemes 93

7.2 Summary of Conclusions 99

7.3 Areas for Future Research 102

APPENDIX A. DETAILS OF THE IMPLEMENTATION 103

REFERENCES 123

FIGURES AND TABLES

Number		Page
Table 2.1	Process Initiation Functions in the Security Kernel	26
Figure 5.1	Domain and Domain Gate Objects in a Hierarchical File System	56
Table 5.1	Examples of ACL Term Matching	61
Figure 6.1	A Typical Process Initiation	77
Table 6.1	Impact of the Model on the Number of Lines of PL/I Code in the Kernel	90
Table 6.2	Impact of the Model on the Number of Programs in the Kernel	90
Figure 7.1	Hierarchical Process Creation for Mutually Suspicious Subsystems	98

CHAPTER 1

INTRODUCTION

1.1 The Problem.

This thesis is concerned with process initiation in a computer utility. Process initiation consists of all those functions that are performed to support the creation of processes. In the Multics computer utility, these functions are:

1) Process Creation: The addition of a new process to the set of processes being managed by the system.
2) Resource Control: The assignment of resources (CPU cycles, memory pages, and the use of I/O devices) to a new process.
3) Authentication: The identification of the user who will control the new process.
4) Domain Changing: The assignment of a Principal ID, which will be used in determining the process's access to objects in the file system, to a new process.
5) Environment Initialization: The initialization of mechanisms needed to support the computation performed by the new process.

As can be seen from the above list, process initiation includes a wide variety of functions. Some of these functions must enforce security constraints, while others are unrelated to security. In the Multics computer

utility, and in many others, the mechanisms that implement the functions that we include in process initiation are poorly organized and heavily interdependent. This interdependence not only makes all of these mechanisms more difficult to prove correct, but also makes the security of the computer utility dependent on a larger set of mechanisms than the minimum set that is necessary to implement the desired security constraints.

The primary goal of this thesis is to devise an organization for the mechanisms that implement process initiation that is simple and minimizes unnecessary dependencies. New mechanisms will be developed to perform some of the functions listed above in that organization.

A second goal of the thesis is to produce an organization for process initiation that can easily be used for any situation in which processes must be created for users. Processes are a powerful tool for structuring computation and a process initiation mechanism that is simple and inexpensive encourages the use of processes. An implementation of process initiation for the Multics computer utility will be used to test the proposed organization.

1.2 Method of Attack.

We will be most interested in reducing the number of mechanisms on which the security of the computer utility depends, and in reducing the complexity of those mechanisms. We extend the notion of a security kernel [Sc75] to a kernel with several layers. Each layer is responsible for enforcing a different set of security constraints. All of the mechanisms that must function correctly to enforce a particular set of constraints are inside of the kernel layer for that set.

The principle of least privilege [Sa75] is used as a guide to determine the functions that are implemented in each kernel layer. This principle states that each mechanism should be given only those privileges needed to perform its function. Thus, each kernel layer should contain only those mechanisms needed to enforce the security constraints for which that layer is responsible. The principle of least privilege reduces unnecessary dependencies.

Another important structuring technique used in this thesis is to implement each function with a small program module, and to minimize the interactions between modules. By clearly defining the function performed by each such module, we make each module easy to verify. By minimizing the interactions between modules, we make the structure of the system simple and thus easy to verify.

An important goal of this thesis is the minimization of common mechanism. By this we mean making the set of mechanisms on which all users must depend as small as possible by removing mechanisms that don't need to be shared and by simplifying those that remain. Such common mechanisms must be included in the security kernel. Any mechanism that a user need not depend on need not be certified, as a user who is not satisfied that such a mechanism is correctly implemented can avoid using it. The structure presented for process initiation in this thesis has very little mechanism on which all users must depend.

1.3 Results.

This thesis demonstrates that the security kernel of a computer utility can be simplified by making process creation unprivileged. The authorization for process creation is provided by the domain changing mechanism, which forces a new process to begin execution at a controlled location. An unprivileged process can thus be used to create a process for a potential user in that user's domain. Authentication of the user is performed by an unprivileged initial procedure in that domain. The remainder of this section describes these results in somewhat greater detail.

A security kernel with three layers is used in the thesis. The layers provide:

1) Access Control: Restrictions on the operations that processes can perform on objects.
2) Prevention of Unauthorized Denial of Service: A guarantee that each user receives a fair share of the available resources.
3) Confinement. A guarantee that information stored in the computer utility is released only to users who are authorized to see that information.

The thesis partitions process initiation into the five functions mentioned above: Process creation, resource control, authentication, domain changing, and environment initialization. Each function is implemented in the kernel layer that provides the least privilege required to perform that function. The thesis considers three of the functions (domain changing, authentication, and resource control) in detail.

The domain changing mechanism for process initiation, which controls a newly created process's access to objects, must perform a similar function to that of mechanisms used to control the calling of protected subsystems. The desired characteristics for a domain changing mechanism that will serve both purposes in an access control list oriented system, such as Multics, are presented and discussed. We present several domain changing mechanisms that can be used for both purposes.

The thesis shows that authentication can be removed from the access control and denial of service layers of the kernel. This removal can be accomplished by allowing each user to select his own authentication procedures. The thesis also shows how authentication can be removed from the confinement layer by allowing different authentication mechanisms to guard the release of different pieces of confined information.

The thesis also presents the concept of authentication forwarding, which allows information obtained through authentication to be shared in a secure way. Authentication forwarding is a natural model for dealing with authentication information. Authentication forwarding allows processes to make use of authentication procedures performed by the system without forcing every user to be dependent on the correctness of such procedures.

The test implementation of process initiation done for the Multics computer utility demonstrates that the functionality of process initiation provided by Multics can be achieved with a much simpler structure than that currently used. The implementation also makes the set of programs that must function correctly in order to enforce a particular security constraint much easier to distinguish.

1.4 Thesis Plan.

The first three sections of this chapter have provided a brief overview of the work done in this thesis. The remainder of this chapter discusses previous work in the areas of computer security and process initiation.

The second chapter presents the model for computer protection mechanisms that is used in this thesis. This model is used to define more precisely the notion of a layered security kernel, and to define clearly the layers used in this thesis. The five functions of process initiation are described, and each function is assigned to a layer of the kernel according to the privileges required to perform that function.

Chapter three considers the problem of authentication. We show that authentication falls outside the access control and denial service layers of the kernel in our protection model, and show how to remove authentication from the confinement layer. We present the concept of authentication forwarding, and discuss the functions that must be performed by an authentication forwarding mechanism.

Chapter four considers the problem of resource control. We discuss the issues involved in performing resource control, and show how many policies of resource control can be implemented by programs executing in an environment that does not provide privileges that would allow those programs to violate the constraints provided by the access control layer.

Chapter five presents four mechanisms for authorizing domain changing. Properties of domain changing mechanisms desirable for process initiation and protected subsystem calling are discussed. The advantages and disadvantages of each of these mechanisms are evaluated, before choosing the mechanism used in the test implementation.

Chapter six discusses an implementation of process initiation for the Multics computer utility. A brief description of Multics is presented, with special emphasis on the properties of the current process initiation scheme. We describe an implementation of process initiation for Multics based on our model, and show that that implementation is substantially simpler than the one currently used. A more detailed description of the implementation appears in Appendix A.

Chapter seven evaluates the usefulness of the model in structuring process initiation. The model is compared with two common process initiation schemes in three situations in which a process is created. The chapter summarizes our conclusions about the model and discusses topics for further research in process initiation.

1.5 Related Work.

This thesis draws heavily on previous work on computer protection mechanisms. The concept of protection domains introduced by Lampson [La74] forms the basis for the access control scheme used by this thesis. The design of a confinement mechanism for the thesis was influenced by much previous work on the confinement problem [An74,Be73,La73,Ro74,Sc75,We69]. The domain changing mechanisms of Jones [Jo72] and Schroeder [Sc72] strongly influenced the design of the mechanisms for authorizing domain changes in the thesis. A study of these two theses first lead to the idea that process creation could be made unprivileged.

This thesis is part of a research effort described by Schroeder [Sc75] by the Computer Systems Research group of the M.I.T. Laboratory for Computer Science to simplify the security kernel of the Multics computer utility. The

Multics system [Or72] is ideal for such study because it contains sophisticated hardware and software protection mechanisms. Some recent theses [Br75,Ja74] have shown that various functions could be removed from the security kernel. Other work [Be73,Re76,Hu76] has demonstrated that the security kernel can be substantially simplified by structuring the functions that it performs. This thesis shows that some of the functions of process initiation can be removed from the kernel, and presents a structure that simplifies those that remain.

CHAPTER 2

A MODEL FOR PROCESS INITIATION

In this chapter, we show how to perform process initiation in a secure computer utility. First, we define more precisely what is meant in this thesis by "secure", by defining three security goals. We then examine briefly the mechanisms used to enforce those security goals to see how they interact with process initiation. We show that the security goals can be enforced by a security kernel with three layers. Finally, we examine each of the five process initiation functions and show in which layer of the kernel each function should be implemented.

2.1 Security Goals.

In this section, we define three security goals for a computer utility:

1) Access Control - The control of the operations that can be performed on objects in the computer utility.
2) Prevention of Unauthorized Denial of Service - A guarantee that authorized operations can actually be performed.
3) Confinement - The prevention of the release of information stored in a computer utility to users not authorized to see that information.

Access Control.

As stated above, the goal of access control is to provide control of the operations that can be performed on objects. Such control allows the user or users responsible for an object to protect the integrity of that object. To provide access control, we use the concept of protection domains [La74].

Each process in the computer utility is associated with a protection domain by a process-domain binding, a binding made in a system-wide context. The domain of a process determines the operations that that process can perform on objects in the computer utility. The domain of a process represents the authority responsible for the activities of that process.

The details of how the operations that a process can perform are determined from the domain of the process are not important in this chapter. We can imagine that there is a two-dimensional matrix, which for each domain and object specifies the operations that a process in that domain can perform on that object. In chapter five, we consider access control mechanisms in greater detail.

In order for such an access control mechanism to provide protection for objects, the association of a process with a domain must be controlled. If a user could obtain control of a process in any domain, then the access control mechanism could not deny that user the use of any object. This thesis refers to the problem of authorizing changes in the process-domain binding as domain changing. Domain changing is described in greater detail in a later section of this chapter and in chapter five.

Prevention of Unauthorized Denial of Service.

The goal of prevention of unauthorized denial of service is to keep one user from interfering with the use of the computer utility by other users. One common example of denial of service occurs when a user can exploit a flaw in the operating system of the computer utility to cause the computer utility to fail. Such a failure denies service to all users while the system is restarted, and may cause work in progress at the time of the failure to be lost.

Many less severe examples of denial of service exist. In some computer utilities, one user can capture a sufficiently large percentage of the available computing power or memory, that the use of the system by other users is impaired. In this thesis, denial of service generally refers to the denial of the right to use a process.

Confinement.

Simply stated, the goal of confinement is to provide control over the set of users who are allowed to observe a piece of information in the computer utility. (1) Confinement has been used to prevent the release of classified military information [We69]. Confinement has also been used to protect proprietary information that must be read by an uncertified program [Ro74].

There are two definitions of the confinement problem: message confinement and total confinement. Message confinement [An74] consists of preventing the transfer of confined information to unauthorized users through

---

(1) The term "piece of information" can represent a wide variety of things. It can mean the contents of an object such as a file, or the name of an object, or even just the presence of an object. Any of these may convey information that may need to be concealed from some set of users.

the operations performed on objects. Total confinement consists of preventing the transfer of confined information to unauthorized users through any means, however slow or obscure. (This includes the covert channels of Lampson [La73], which transfer information through the observation of the use of shared resources.) The mechanisms discussed in the next section are intended to provide message confinement. In order to provide total confinement, the use of shared resources must be controlled so as to block information transfer through covert channels. Several researchers have proposed mechanisms to achieve confinement in a computer utility. [An74,Be73,Ro74,We69] These mechanisms all tag the objects in the computer utility with some indication of the confined information that they represent, and use the tags to restrict the distribution of information to users. There are two ways in which the tags have been used to provide confinement:

1) The high water mark. [Ro74] In these mechanisms, each operation that modifies an object and may add confined information to that object, changes the tag of that object to reflect the confined information that could have been transfered.
2) The *-property. [Be73] In these mechanisms, an operation that modifies an object is not allowed unless that object is already tagged as containing any confined information that the operation could add.

For this thesis, the second type of mechanism is chosen. Rotenberg [Ro74] shows how the changing of the tags that occurs with the high water mark mechanism can itself be used to convey confined information. It therefore seems extremely difficult to achieve total confinement with a high water mark mechanism.

The model of confinement used in this thesis tags each object, process, and user with a confinement set. A confinement set is a set of confinement attributes. Each confinement attrigute is used to represent some class of information, such as a military security classification, or a proprietary project. The confinement set of an object identifies the confined information that that object contains. The confinement set of a process indicates the confined information that that process is allowed to observe. The confinement set of a user represents the information that the user may observe. Three rules are used to enforce confinement:

1) A process is allowed to perform an operation that observes an object (i.e. one whose outcome depends on the contents of the object) only if the confinement set of the object is a subset of that of the process.
2) A process is allowed to perform an operation that modifies an object only if the confinement set of the object contains that of the process.
3) A process can direct the output of an object to a user only if the confinement set of the user contains the confinement set of the object, and that of the process.

These rules taken together enforce what is referred to elsewhere as the *-property. (1)

Process initiation interacts with confinement in several ways. The process initiation mechanism must assign a confinement set to each newly

---

(1) Some mechanisms use a level and category set, similar to a military classification, to objects, processes, and users. [We69]. By using one confinement attribute for each level and each category, the mechanism presented above can be made to enforce the same constraints as a level and category mechanism. The above mechanism was chosen because the rules (the *-property) are significantly simpler with this approach.

created process. This assignment must be done in such a way that confined information is not released. The process initiation mechanism must also prevent the use of process creation as a signal to transmit information to a user who is not authorized to see that information.

2.2 A Layered Security Kernel.

The set of mechanisms that must function correctly in order to provide security is known as the *security kernel*. One design goal for a secure computer utility is to make the set of mechanisms in the kernel small and simple, thus making the kernel easier to verify. The notion of a security kernel can be extended to a kernel with several layers. Each layer of such a kernel includes all of the programs needed to enforce a different set of security constraints.

A kernel with multiple layers is useful because it indicates clearly the mechanisms capable of violating each security constraint. The specifications for each layer of the kernel need not include any indication that that layer does not violate the security constraints provided by lower layers. This reduction in specification simplifies the task of verifying the kernel.

In this thesis, we choose three kernel layers corresponding to the three security goals described above. The innermost layer of the kernel provides access control, the second layer prevents denial of service, and the outer layer provides confinement. The layers were chosen to minimize the number of mechanisms that fall in each layer.

The access control layer is placed below the denial of service layer because the denial of service layer can make better use of the functions provided by the access control layer than vice versa. The denial of service

layer must provide some form of access control in order to keep the actions of users from interfering with each other. The access control layer need not prevent denial of service. (1) Thus if the access control layer is placed below the denial of service layer the denial of service layer can be simplified, as it can make use of the access control provided by the lower layer. For this reason, we place the access control layer below the denial of service layer.

The confinement layer is placed above the denial of service layer for a similar reason. The confinement layer must prevent some types of denial of service. A denial of service cannot be allowed to convey confined information in violation of the *-property. For this reason, we place the denial of service layer below the confinement layer.

The layers chosen in this thesis are by no means the only choice possible. Other researchers [Be73] have chosen to place at the core of the kernel a layer that contains a simple access control mechanism that enforces the *-property for operations performed on objects (message confinement). This layer does not enforce total confinement, as actions such as denial of service can still be used to convey confined information in violation of the *-property. These so-called covert channels [La73] can be used very effectively in many computer systems.

---

(1) Interruptions of the processing done by the access control layer (either through denial of service or through failure of the hardware) must not result in the failure of that layer.

2.3 A Model for Process Initiation.

We now describe a model for process initiation mechanisms. Such mechanisms change the set of processes, the set of domains, and the process-domain binding. We want the model to be as general as possible, so that it can easily be used for any situation in which processes must be created.

Our model separates process initiation into five functions: process creation, domain changing, resource control, authentication, and environment initialization. In this chapter, we discuss briefly what each of these functions does, and in which of the kernel layers previously discussed each mechanism lies. Later chapters consider some of these mechanisms in greater detail.

Process Creation.

Process creation consists of creating an initial process state. A process state describes the characteristics of a process. A process state contains the domain of the process, the confinement set of the process, the execution point of the process, the machine registers of the process, and a description of the address space of the process.

Because process creation alters the process-domain binding, it must be performed within the kernel layer that provides access control. A second reason for including process creation in the kernel layer for access control is that each process may at some point in its lifetime execute functions inside the access control layer. If the process state of such a process is not correctly initialized by process creation, then that process may not be able to perform those functions properly.

Domain Changing.

Domain changing in this thesis really means the authorization of domain changes. The process creation mechanism actually makes the domain changes by altering the process-domain binding according to instructions received from the domain changing mechanism. The problem of authorizing domain changes has been extensively studied. Schroeder [Sc72], among others, concludes that a domain changing mechanism must insure that the first procedure executed by a process that enters a given domain is an acceptable initial procedure for that domain. This is the only function that the domain changing mechanism must perform in order to provide access control. (1) Chapter five dicusses the details of controlling domain changing.

The domain changing function must be performed in the kernel layer that provides access control. The domain changing function needs to alter the process-domain binding, and thus could violate access control constraints if not correctly implemented.

Resource Control.

The resource control function assigns the resources necessary to begin the execution of a process. In the Multics computer utility, these resources consist of CPU cycles and memory pages, as well as the choice of whether or not to allow a process to be created at all. The assignment of resources to processes is made according to a resource control policy that attempts to insure that each user receives a fair share.

---

(1) The initial procedure can control the computation performed by the process, and thus prevent misuse of access rights or resources available to the domain.

Resource control clearly lies within the kernel layer for prevention of unauthorized denial of service. The resource control mechanism can deny a user the right to create a process by refusing to allocate the needed resources. In the design of many current systems, the resource control mechanism also lies within the kernel layer that provides access control. In chapter four, we show how the resource control function can be implemented outside of the access control layer, thus simplifying that layer.

Authentication.

An authentication mechanism is responsible for determining the identity of a user. If a user can control the operations performed by a process (by communicating with a command interpreter executing in that process), then the user must be identified to insure that he is authorized to use the domain of that process. In the Multics computer utility, a process that is created to serve a user has an initial procedure that calls a command processor to give the user control of the process. The identity of the user is determined through authentication before the process is created.

In chapter three, we show how to remove authentication from all three layers of our security kernel. This removal is accomplished by allowing each user to choose his own authentication mechanism. An error in one user's authentication mechanism is no more serious than an error in any other program that that user chooses to call. Each user can protect himself from failures of the authentication mechanisms of other users. Chapter three describes how the three sets of security constraints can be provided without depending on authentication.

Environment Initialization.

Environment initialization consists of the initialization of mechanisms that support the execution of a process. In the Multics system, environment initialization includes the creation of certain working storage segments for the process, the initialization of error handling for the process, and the initialization of stream I/O for that process. Environment initialization is performed by the initial procedure of a process, and the procedures that it calls.

Environment initialization requires no special privileges because all of the functions that it performs are local to the process being created. Environment initialization need not be included in the security kernel.

Summary.

This chapter has presented a brief description of the five functions that are included in process initiation. Each function has been assigned to a layer of our security kernel based on the privileges required to accomplish that function. Table 2.1 summarizes these assignments.

Table 2.1

Process Initiation Functions in the Security Kernel

Function:	Kernel Layer:
Process Creation	Access Control
Domain Changing	Access Control
Resource Control	Denial of Service
Authentication	(none)
Environment Initialization	(none)

These assignments were made only on the basis of least privilege. The implementation described in chapter six shows that each of the functions can actually be implemented in the layer shown above, without undue complexity. Such an implementation insures that each kernel layer contains the minimum number of process initiation functions.

The next three chapters of this thesis explore three of these functions (Authentication, Resource Control, and Domain Changing) in greater detail. These chapters describe mechanisms that can be used to provide those functions in the kernel layers shown above.

CHAPTER 3

AUTHENTICATION

This chapter discusses how authentication is related to process initiation. The chapter begins with a discussion of the properties of authentication mechanisms. These properties shape the attitude toward authentication that is taken by this thesis. We show that authentication need not be performed by the security kernel. We also present the concept of authentication forwarding, which can be used to allow the sharing of information obtained through authentication. Authentication forwarding can reduce the number of times that a user must undergo authentication, by allowing the information obtained from the user's first authentication to be shared among the processes with which he must communicate.

In order to discuss authentication, a model of how users communicate with a computer utility is needed. For this purpose, we adopt the concept of a stream. We use a stream to represent a two-way communication channel. We refer to the user who communicates with the computer utility through a stream as the source of that stream. The time during which a user is communicating with the computer utility will be refered to as a session.

3.1 Properties of Authentication Mechanisms.

An authentication mechanism is a mechanism designed to determine the identity of an unknown user. Such mechanisms usually require the user to

produce some piece of data (password, encryption key, etc.) that must match a value kept by the computer utility. Protection mechanisms enforce security constraints within a computer utility, while an authentication mechanism can be used to identify users for the processes executing on the computer utility.

Three important properties of authentication mechanisms are:

1) No authentication mechanism is perfectly reliable. An authentication mechanism identifies a user by a sequence of bits (password or encryption key) supposedly known only to that user. Because any user can produce such a sequence, any such mechanism can be fooled into misidentifying a user.

2) A security conscious user can always devise an authentication mechanism that is more reliable than a system provided authentication mechanism. The probability that a user will be able to fool an authentication mechanism by guessing the password or key decreases as the length of the password or key is increased. Thus a security conscious user can obtain greater reliability by using a longer password or key, at the expense of having to remember more information.

3) Each use of an authentication mechanism releases information that aids an imposter in determining the password or key. In general, the stream through which a user communicates with the computer utility passes through some insecure channel (such as a telephone line) that an intruder may be able to monitor. Encryption based schemes are less vulnerable to such monitoring than password schemes.

These three properties influence the way in which this thesis deals with authentication. Points one and two suggest that it is not necessarily

desirable for all users to rely on one system-wide authentication mechanism. Such a mechanism cannot be guaranteed always to make correct identifications, and no matter what mechanism is used, a better one always can be found.

Point two suggests that different users might want to use different authentication mechanisms. Different users have different security requirements and thus some users might be willing to spend a great deal (in terms of extra communication, extra computation, and the overhead of remembering more information) to insure that they cannot be impersonated. All of the users of the computer utility might not want to pay the cost of the security requirements of these few.

Point three suggests that authentication should be performed only when necessary. Thus the results of authentication should be remembered, so that each new process or domain that encounters a stream does not necessarily have to perform authentication. Authentication Forwarding is introduced to provide this memory.

Authentication and Security.

In this section, we examine how authentication must be used to enforce the security constraints of our three kernel layers.

1) Access Control.

The innermost layer of our kernel is responsible for providing protection for objects in the computer utility. The definition of the security provided by this layer of the kernel was carefully chosen to avoid the notion of a user. This layer of the kernel insures that objects can be accessed only by authorized domains. This constraint can be enforced without using authentication to identify users.

By ensuring that a process can enter a domain only through a controlled initial procedure, we allow the initial procedure to guard the domain. The initial procedure can authenticate a user before allowing that user to control the process.

In many computer utilities, each user is authenticated soon after he contacts the utility. An authenticated user is then allowed to change the authentication procedure to be used for future sessions (by changing his password,) and to specify from his terminal the operations that the computer utility will perform for him during the current session. In the organization used in this thesis, a user who contacts a computer must choose an initial domain. He then must satisfy whatever authentication mechanism is used by the initial procedure of that domain. Even after successful authentication, the initial procedure may impose limits on the operations that will be performed for the user.

The organization used in this thesis allows a user who requires a high degree of security to specify his own authentication procedure in the initial procedure for the domain that he will use (as will be shown in chapter 5). It also allows for limited service users, a concept that has proved useful in current computer utilities.

2) Denial of Service.

Whether or not authentication is required to prevent unauthorized denial of service depends on whether the utility guarantees service to users, or whether it guarantees service to domains. If a computer utility guarantees each user a fair share of the available resources, users must be authenticated to insure that one user cannot monopolize the resources of the computer

utility by requesting services from many terminals simultaneously. Domains can be guaranteed a fair share of the available resources by imposing restrictions on the resource use of processes. The resource controller need not be aware of the fact that some of the processes are performing operations on behalf of the users of the computer utility.

The initial procedure of a domain can be used to allocate the resources guaranteed to that domain to users, much the same as the initial procedure is used to insure that the access rights granted to that domain are not abused. The Multics computer utility uses a resource control scheme that assigns resources to processes based on their principal ID. As we show in chapter six, this resource control scheme can be implemented without relying on authentication.

3) Confinement.

Authentication is required in some form in order to achieve confinement. This is because the purpose of confinement is to prevent a user from obtaining information that he is not entitled to. There are several ways in which authentication can be incorporated into the mechanism that enforces confinement.

One way to provide confinement is to authenticate each user who contacts the computer utility and to insure that each process with which the user communicates has a confinement set that is smaller than that of the user. This scheme has the disadvantage of system-wide authentication schemes mentioned before, namely that it does not allow different authentication mechanisms to be used for different users with different security needs. Because different confinement attributes protect different information, it is

likely that some of that information is more valuable than the rest and therefore a user should be forced to pass a more rigorous authentication before gaining access to such information. The following scheme allows different authentication mechanisms to be used to obtain different confinement attributes.

Each terminal that contacts the computer utility is initialy assigned an empty confinement set. A process that wishes to communicate with a terminal may discover that it cannot do so because the confinement set of the terminal does not contain the confinement set of the process. The process must call on an authentication mechanism to identify the user at the terminal. After the authentication mechanism has identified the user, it changes the confinement set of the terminal to include the confinement set of the authenticated user. Each authentication mechanism is only authorized to supply some of the possible confinement attributes, so that different authentication mechanisms can be used to grant different confinement attributes.

This scheme also has the advantage that the responsibility for devising and maintaining the authentication mechanisms can be distributed among the users who wish their information to be protected by confinement. The computer utility need only provide some means of allocating the confinement set attributes and establishing the authorized authentication mechanisms.

The major disadvantage of the above scheme is that a user with a large confinement sets may have to be authenticated several times during the same session in order to obtain access to all of the information that he needs. Current applications of confinement mechanisms do not tend to have users with large confinement sets. Also, a user rarely needs access to all of the information that he is potentially entitled to in any one session. Making it

awkward or costly for a user to obtain access to all of the information that he could potentially see may have the beneficial effect of encouraging each user to obtain only the privileges that he needs for his current task.

Encryption.

Much recent work on authentication has been devoted to the developement of authentication mechanisms based on encryption. Such schemes have the advantage over passwords that the sensitive identifying information (password or encryption key) is not sent through the stream, and thus is less vulnerable to being stolen. Some of the protocols require that each process that talks to a stream know the encryption key for that stream. The scheme developed by Kent [Ke76] uses one key for authentication and one key to provide secure communication through the stream once authentication has been performed. The second key must be known by each process that communicates with the stream. The authentication forwarding mechanism described below is well suited for the distribution of such keys.

3.2 Authentication Forwarding.

We say that a process that relies on a previously performed authentication to determine the identity of the source of a stream is using a forwarded authentication. Thus in most computer systems, where a system-wide mechanism authenticates users when they first contact the system, each process relies on a forwarded authentication (from the system-wide mechanism) for the source of the stream from which it draws commands.

Forwarded authentications are a very common phenomenon outside of the computer utility. Identification cards represent forwarded authentications. Anyone who determines the identity of a person from an identification card (or

driver's license or credit card) is actually relying on the authentication performed by the issuer of the card. Unfortunately, identification cards can be lost, stolen, or forged. Forwarded authentications maintained inside a computer utility can be protected, making them unforgeable and unstealable.

There are two facts that any process using a forwarded authentication must know: The claimed identity of the user, and the authentication procedure used. Both of these facts can be provided by allowing a process that performs authentication to record securely the identity determined for the user. In order to allow the authentication mechanism used to be determined, sufficient information to identify the author of each forwarded authentication must also be recorded. With our model, the necessary information is the process, domain, and procedure that recorded the result of an authentication, and the time of recording. This information allows a process that uses a forwarded authentication to identify the authentication mechanism used, just as the distinctive format of an identification card allows the issuer of the card to be identified.

Identification cards sometimes become invalid, due to changes in the information that they contain. In the computer utility, a change in the source of a stream invalidates previous authentications for that stream. The computer utility cannot always detect each case in which the source of a stream changes. (1) In the case of streams with finite lifetimes, such as telephone or other network connections, the computer utility can detect when a user's stream has been disconnected, and should forget any authentications

---

(1) One case in which it's difficult to detect a change in the source of a stream occurs when a user walks away from a terminal and a second user takes over without either one informing the computer utility of the change.

performed for such a stream. The authentication forwarding mechanism should delete the forwarded authentications for a stream when that stream is disconnected. A stream can be disconnected and reconnected between the time when a process performs an authentication and the time when that process records the authentication, leading to an incorrect forwarded authentication.

One solution to this problem is to have the computer utility maintain a count of the number of times that a stream has been connected. The process performing authentication can then obtain this connection count before performing authentication and present the connection count to the authentication forwarding mechanism along with the forwarded authentication. The authentication forwarding mechanism can then obtain the current connection count in order to determine whether or not the forwarded authentication is valid. The connection count is used as the eventcounts of Kanodia and Reed [Ka76].

A forwarded authentication for a stream is useful only to the processes that can read from or write to that stream. It therefore seems desirable to allow only those processes that can read or write a stream to read the forwarded authentications for a stream. We also allow only those processes that can read from a stream to record forwarded authentications for that stream. These restrictions allow the computer utility to limit the resources expended in keeping forwarded authentications, by limiting the number of authentications kept for each stream, without allowing one process to monopolize these resources by recording forwarded authentications for streams that it cannot use. The above restrictions are not necessary for security reasons, because the information recorded with a forwarded authentication identifies the author of that authentication and prevents forgery.

We must, however, keep authentication forwarding from becoming a covert channel for confined information. This can be done by assigning a confinement set to each forwarded authentication and forcing the reading of forwarded authentications to obey the *-property. Each forwarded authentication is given the confinement set of its author. (1)

3.3 Example.

The following section shows how processes are created for users of a computer utility using the ideas on authentication of this chapter. The scheme described is compared with a more commonly used scheme for incorporating authentication into process creation.

A user who contacts a computer utility for service informs the computer utility of his identity. Based on this identity, the computer utility selects a domain in which to create a process to serve the user. The computer utility may or may not authenticate the user to verify his right to use the requested domain, perhaps by demanding a password. If authentication is performed, then the result of that authentication is recorded as a forwarded authentication for the stream that represents the user's terminal. A process is then created for the user, beginning execution in the chosen domain in one of the valid initial procedures for that domain. It is the responsibility of the initial procedure to determine whether or not to serve the user. This decision could be based on the forwarded authentications recorded for the user's stream.

---

(1) An alternate scheme would be to give each forwarded authentication the confinement set of the corresponding stream. This scheme would not work well for a system in which the confinement sets of streams changed, such as the authentication scheme described above where a stream gains confinement attributes after its source is authenticated.

If the user desires access to confined information, then he must make contact with a process with the desired confinement set (either by specifying that his initial process be created with a non-null confinement set, or by asking his initial process to try to change its confinement set or give his stream to some process with the desired confinement set). Such a process will discover that it cannot communicate with the user, and must select one or more authentication mechanisms to call on to identify the user, depending on the attributes that the confinement set of the user's stream is missing. Each of these authentication mechanisms in turn records forwarded authentications for the user's stream, and some of these mechanisms may rely on authentications forwarded from others.

We contrast this scheme with the authentication scheme used in most computer systems today, which uses a system-wide authentication mechanism to identify each user who contacts the system. An authenticated user can then create and control processes in any domain that he is authorized to use.

Notice that the scheme presented in this chapter can be made to behave like the more common scheme (by performing authentication for all users who contact the computer utility, and having all initial procedures make use of the forwarded authentication from the system-wide mechanism). Thus a user who does not require a high degree of security need not generate his own authentication mechanism and can instead rely on the system-wide mechanism. A highly privileged domain, however, can be guarded by an arbitrarily secure authentication mechanism.

One of the most important differences between our scheme and the more commonly used one is that the process that responds to a user who contacts a computer utility (called the listener, logger or monitor, in some computer

systems), needs no special privileges in order to create processes for users. We therefore can remove this process from the security kernel. This process generally executes complex programs, because it must be capable of dealing with several users concurrently, and work with a large variety of ports on the computer.

Notice also that several processes can be used to wait for users to contact the computer utility. Different processes can be used to respond to different types of streams (telephone connections versus network connections), and thus the complexities of dealing with a particular type of stream can be isolated in one process. A utility with parallel processing capability may also want to make use of multiple processes to increase the rate at which new users can be handled.

In chapter six, we show how this authentication scheme can be implemented for the Multics computer utility. Chapters six and seven summarize the advantages and disadvantages of this scheme.

CHAPTER 4

RESOURCE CONTROL

This chapter discusses how resource control is related to process initiation. We begin with a discussion of the issues involved in controlling resource use in a computer utility. We then present a set of operations through which the use of resources in the computer utility can be controlled, and show that the use of these operations can not violate access control constraints. The chapter concludes with a discussion of the kinds of resource control policies that can be implemented using our set of operations, and the security constraints that can be violated through the use of these operations.

4.1 Issues of Resource Control.

A resource is a service provided by the computer utility. Thus resources can include physical devices (like printers, card readers etc.), abstract devices (virtual processors, memory pages, etc.), or even programs (matrix inverters, etc.). This chapter is most concerned with the resources needed to initiate a process. In Multics, these resources are the process itself, and the CPU cycles and memory pages needed to execute the initial procedure. Resource control consists of the distribution of resources to processes, and recording the use of resources by processes for accounting.

In this section, we present some of the issues involved in the control of resource use in a computer utility. These issues guide the way in which resource control is included in the model of process initiation. We consider

two issues: The distinction between mechanism and policy and the general scheme of resource control used (hierarchical or central).

Policy and Mechanism.

Recent research [Jo72,An74] has stressed the importance of distinguishing policies from the mechanisms used to implement those policies inside a computer utility. The separation of mechanism and policy is particularly important in the area of resource control, since different resource control policies may be appropriate for different resources of the same system. Different policies may also be needed for different users. A flexible resource control mechanism can implement a wide variety of policies.

This chapter is most concerned with the interface between mechanism and policy. The interface should be chosen so that the mechanism can be implemented with a small, simple, and easily verifiable set of program modules. At the same time, the interface should support a wide variety of resource control policies, without allowing the violation of access control constraints through the use of the operations provided by the interface. Such an interface allows the removal of the most complicated and variable portion of resource control (the policy) from the access control layer of the security kernel.

Resource Control Philosophy.

Two common approaches to resource control are the hierarchical and centralized systems of control. In the centralized system, there is a central authority known as the resource controller that is responsible for the assignment of resources to all processes. In the hierarchical scheme, each process is responsible for fulfilling the resource needs of the processes that

it creates. Thus each process acts as resource controller for its descendents.

The hierarchical system has the advantage that the creator of a process has more knowledge of the anticipated resource needs of that process than a centralized resource controller, and thus can make a better decision of the resources to assign. The hierarchical system is also quite flexible because each process can implement its own policy of resource control.

However, the hierarchical scheme requires that each process that creates processes perform resource control. This duplication makes it difficult to add a new type of resource, because several algorithms may need to be modified to deal with the new resource. In the central scheme, only the central resource controller need be modified to add a new type of resource. Duplication of mechanisms also increases the chance of error.

The hierarchical scheme does not respond well to processes with erratic, time-varying resource requirements. Resources assigned to meet a sudden demand by such a process may have to pass through resource control algorithms in several processes. These algorithms may be unwilling or unable to meet such a demand.

Another disadvantage of the hierarchical scheme is that it does not provide for a process and its creator to be mutually suspicious. Each process must trust its creator to assign the resources that that process needs. In turn, each process must trust its descendents not to waste their assigned resources by not performing the desired task. The centralized scheme does not share this difficulty, as each process is dependent only on the central resource controller for its resources. A process and its creator can be mutually suspicious, because neither must depend on the other for resources.

A fourth problem with the hierarchical scheme is that it does not interact well with confinement. In a computer utility with hierarchical control, the resources that a process assigns to its descendents can be used as a covert communication channel to pass confined information. In addition, each process can signal information to its creator through its use of the assigned resources. Both of these channels are difficult to block with the hierarchical resource control. If neither of the channels is blocked, then each process must be assigned the same confinement set as its creator, so that neither channel can be used to violate confinement. Such an assignment of confinement sets would force all processes to have the same confinement set.

Because mutual suspicion and confinement are both considered important in this thesis, we choose centralized control.

4.2 Primitive Operations for Resource Control

In this section, we present and discuss a set of primitive operations that enable a centralized authority to perform resource control. These operations form the interface between mechanism and policy discussed above. We show that the operations do not allow the resource controller to violate access control constraints, but do allow the resource controller to implement a wide variety of resource control policies.

We use the following set of primitive operations for resource control:

1) The resource controller will be allowed to control the distribution of resources to all processes.

2) The resource controller will be allowed to monitor the use of all resources by all processes.

3) The resource controller will be allowed to observe a fixed set of parameters of a proposed process initiation (such as the initial procedure or domain), and veto the creation of a process.
4) The resource controller will be allowed to destroy any process.

The first of these operations is clearly needed to implement a resource control policy. Different types of control are needed for different resources. Some resources, such as line printers or card readers, are assigned to a process for a relatively long time period (minutes at least). Primitive operations that allow the resource controller to assign such resources to processes should be provided. Some resources, such as the use of the CPU or memory, must be rapidly switched among processes in order to provide rapid response to requests from users. A small, simple, and fast control mechanism is generally provided for such resources. The resource controller controls the distribution of such resources by specifying to this control mechanism the set of processes in contention for the resource and the priority of each process.

The second operation allows the resource controller to observe the resource use of each process, even if the actual assignment of resources is made by a lower level mechanism (as in the assignment of CPU cycles and memory pages described above). This primitive allows the resource controller to record resource use for accounting.

The last two operations allow the resource controller to control the total number of processes. Each process may consume space in tables that contain the state of that process, and the amount of such space may be limited. The performance of algorithms for multiplexing the available

processors and memory among processes degrades as the number of processes increases. The resource control policy of the computer utility may therefore dictate that the number of processes be limited. Another reason for limiting the number of processes is to provide good response to sudden changes in the resource requirements of processes. If the resources are divided among too many processes, it may be difficult for the resource controller to gather all the resources needed to meet a large demand by one process. The resource controller is allowed to observe certain characteristics of each process that is created, so as to have some basis for deciding whether or not to allow the creation of that process.

We now show that none of the four operations allows the resource controller to violate the access control constraints of the kernel. This property allows a resource controller that depends only on the above operations in order to perform control to be implemented outside of the access control layer of the kernel.

There are three ways in which one of our primitives might violate the constraints of the access control layer:

1) It might perform an operation not authorized by the access control mechanism.
2) It might alter the process-domain binding.
3) It might change the relationship that determines the operations that each domain can perform on each object. (In the case of Multics, the access control lists.)

The first of the primitives controls the assignment of resources to processes. The primitive does not alter the process-domain binding, nor does it alter the set of operations that each domain is allowed to perform. It therefore does not violate the constraints of the access control layer (1)

The observation of resource use clearly cannot alter access control information. It may, however, allow the resource controller to observe the objects being used by a process even if the domain of the resource controller does not authorize the resource controller to see those objects. This does not violate access control, as no process can be compelled to give away information in this manner. It does, however, allow the resource controller to violate confinement, which is one reason that the resource controller is included in the kernel layer that enforces confinement.

The resource controller can change the process-domain binding by rejecting a process creation request, or by destroying a process. The change does not, however, allow the resource controller to gain unauthorized access to objects.

Thus the four operations do not allow the resource controller to violate the access control constraints of the kernel. They do, however, give the resource controller knowledge of the resource use of all processes, and total control of all resource allocation. These abilities allow a wide variety of resource control policies to be implemented.

---

(1) We must be very careful, however, that resource assignments do not affect the functioning of the access control layer. In a system with a distributed supervisor, the withdrawal of resources may stop a process that is modifying access control information, and may leave that information inconsistent.

4.3 Limitations on Resource Control Policy.

There are limitations on the resource control policies that can be implemented with these primitives. As noted before, the resource controller does not know the identity of the users who control the processes of the computer utility. Thus the resource controller cannot base resource allocation decisions on the knowledge of which user will control the process that receives the resources. We suggested earlier that the resource controller use the initial domain of a process to determine the resources that the process will receive. This seems a satisfactory substitute in most cases.

We have also made no provision for the resource controller to find out the details of the computation being performed by a process. Allowing the resource controller to observe more about the execution of a process makes it more difficult for a process to conceal the contents of the objects that it uses from the resource controller. Such observation may be needed in order to implement some resource control policies, such as a policy that grants higher priority to a process when that process is performing certain tasks. The parameters that the resource controller is allowed to observe when the process is created may help the resource controller to determine the task that a process performs, but they do not allow the resource controller to distinguish among several tasks performed in the same process.

4.4 Security Limitations.

There are also limitations on the security constraints that can be enforced without certifying the resource controller. Although we have shown that we can remove the resource controller from the kernel layer that implements access control, it is clear that the four operations give the

resource controller the power to deny service, and thus must be in the kernel layer that prevents denial of service. We also saw that the operations allow each process to transmit information to the resource controller, and that the resource controller can transmit information to any process through the resources that it allocates. Because of these information channels, the resource controller must be certified not to violate confinement.

A less obvious problem is that of revocation. The ability to revoke access to objects may be very important to the functioning of a computer utility. A denial of service can prevent a process from revoking access. Although this does not violate the access control constraints (the right to revoke access is not guaranteed), it may cause inconvenience to the users of the system.

Summary

We have shown how a centralized scheme of resource control can be implemented with four primitive operations. These operations allow a wide variety of resource control policies to be implemented. The primitive operations do not allow the resource controller, which implements the resource control policy, to violate access control constraints. Chapter six shows how the complicated resource control policy of the Multics computer utility can be implemented in this manner. This implementation substantially simplifies the access control layer of the kernel.

CHAPTER 5

MECHANISMS FOR AUTHORIZING DOMAIN CHANGES

This chapter considers mechanisms to authorize domain changes in a computer utility. The chapter assumes a list-oriented implementation of access control, such as that of Multics [Or72]. The mechanisms discussed use the access control mechanism of the computer utility to authorize domain changes. Each mechanism is evaluated for use in authorizing process initiation and for use in the calling of protected subsystems.

5.1 Introduction.

The domain changing mechanism needed in process initiation performs similar functions to the mechanisms needed to authorize the calling of a protected subsystem. We therefore desire to have one mechanism that will serve for both purposes.

The mechanisms to be described all make use of two special types of objects in the computer utility, domain objects and domain gate objects. Access to a domain gate object is required in order to create a process or call a protected subsystem, while access to a domain object is required for the creation of domain gate objects. These special objects are used because the access control mechanism of the computer utility can be used to authorize domain changes, just as it is used to authorize operations performed on other types of objects. There is a unique identifier for each domain that we refer to as a Domain Identifier (Domain ID). A Domain ID is used to designate a

domain in the same way that Saltzer uses a Principal ID to designate a domain (Sa75). Each Access Control List consists of a list of terms (ACL terms) that specify a Domain ID and a set of access rights. A process's access rights for an object are determined by the term of the ACL for the object that matches the Domain ID of the domain of the process. The matching algorithm used depends on the particular domain changing mechanism used.

The remainder of this chapter describes four mechanisms to control domain changing. These mechanisms represent a number of ways to control domain changing using the access control mechanisms of the computer utility. They include mechanisms designed for process initiation and those designed for protected subsystem calls. Included in this set of mechanisms are mechanisms similar to those used by Jones [Jo72] and Schroeder [Sc72] to authorize domain changes.

5.2 Four Mechanisms for Authorizing Domain Changes.

I have named the four mechanisms to be presented Exact Specification, Partial Specification, Last Component Specification, and Appending Specification. Exact Specification is the simplest of the four mechanisms. Partial Specification is slightly more complicated, but can be used to implement authorization schemes that allow several authorities to share responsibility for a domain, such as the scheme used in the Multics computer utility [Or72]. Last Component Specification is similar to the mechanism presented in Schroeder's thesis to control the creation and calling of protected subsystems. [Sc72] Appending Specification is a much more general mechanism that allows the entire call history of a process to be used in determining the access rights of that process.

5.2.1 Exact Specification.

The first mechanism for domain entry control to be discussed will be referred to as Exact Specification. Each domain change is authorized by a domain gate object. A domain gate object specifies a Domain ID and an initial procedure. A process makes a call to a procedure in another domain by calling the "domain call" primitive (an operation provided by the security kernel) and passing it the name of a domain gate object. (1) If the process has "call" access to the domain gate object, the domain of the process is changed by the kernel to that specified by the domain gate and the process executes the specified initial procedure. To create a process, one must call the process creation primitive passing it the name of a domain gate object to which the caller has "create" access.

The "call" and "create" accesses described above are determined from the ACL of the domain gate. (2)

The creation of new domain gates is controlled by the domain objects. Each domain object specifies a Domain ID. A process may create a domain gate by calling the "create_gate" primitive, passing it the name of a domain object and the name of an initial procedure. The process must have "create_gates" access to the specified domain object.

---

(1) If an attempt to call the gate directly resulted in an error condition, then the computer utility could detect attempts to call domain gates and invoke the domain call primitive automatically. This scheme is similar to dynamic linking. The calling procedure could then call the gate just as it would call any procedure in the same domain.

(2) As noted before, the initial procedure for a domain can be used to guard the access rights and resources of that domain. Therefore, the "call" and "create" access rights are unnecesssary, and only serve as a convenience. The important function of the domain gate object is to bind together an initial procedure and a domain.

The creation of domain objects must be controlled, since any process with access to a domain object can create new gates for the domain that is specified by that objects. This control can be accomplished by allowing the creation of a domain object only if the Domain ID specified by that domain object has not been previously used.

It is important to understand the system of control being employed in this mechanism as it is common to all the mechanisms discussed in this chapter. This system of control is very similar to that used by Schroeder [Sc72] to control the creation and calling of protected subsystems. The creation of new domains is an unprivileged operation, as any process is allowed to create new domain objects, while the creation of gates into a particular domain is under the control of the domain object for that domain.

Notice that access to a domain gate object is sufficient to use a domain gate. Access to a domain object is not required. Thus we cannot, through the ACL of a domain object, revoke the right to use domain gates that were created using that domain object. Adding to the ACL of a domain object is in some sense non-revokable. This non-revokability is true of all of the domain changing mechanisms discussed by this chapter. We could provide some mechanism to destroy all of the domain gates created from a particular domain object. Because domain gates cannot be freely transferred or duplicated, as can capabilities, it is easy for the computer utility to locate all of the domain gates that were created using a particular domain object.

Exact Specification could be used for both calling and process initiation, as it is capable of authorizing a domain change between any two domains. It also seems relatively easy to implement. There are, however, two disadvantages to this mechanism that make it less suitable.

Using Exact Specification, a process that has "create_gates" access to a domain object can use the corresponding domain by creating gates into that domain. Thus in the case that there is a single authority responsible for a domain, that authority can use the domain object to control the use of the domain. Several computer systems, including Multics, allow two or more independent authorities to share responsibility for a domain. The use of a domain in such a system requires the independent approval of all of the authorities that share responsibility for that domain. An example from the Multics computer utility should help illustrate the use of such a system of control.

In the Multics computer utility, Principal IDs (Domain IDs in our terminology) have Person and Project components. The creation of a process with a particular Principal ID requires the independent approval of both the user who corresponds to the Person component and the project administrator of the project that corresponds to the Project component of that Principal ID. The Principal IDs that appear in Access Control Control (ACL) terms are allowed to contain "*" components that match any value of the corresponding component in a Principal ID of a process. Thus the term "Jones.*.* read" grants read access to any process that has a Principal ID with a Person component of "Jones".

Such ACL terms are frequently used to allow all of the users of a given project to use a particular program or data base, or to allow a user to have access to his private data while working on any project. In order to preserve the meaning of such terms while using Exact Specification to control domain changing, we must carefully control the creation of a domain object with a Domain ID that matches a previously created Domain ID in any component. For

example, we could not allow the creation of a domain object with a Domain ID of "Jones.new" if the Domain ID of "Jones.old" had already been used. This is because the domain "Jones.new" can gain access to objects through ACL terms with a Domain ID of "Jones.*" and therefore the use of that domain must be authorized by the person corresponding to "Jones".

The above problem can be solved by allowing only a trusted system administrator to create a domain object that specifies a Domain ID that matches a previously existing Domain ID in some component. This solution, however, overly restricts the way in which users may create and use domains, and forces all users to trust the system administrators. The Partial Specification mechanism to be discussed later provides a better way to allow several authorities to share responsibility for a Domain.

A second difficulty with the Exact Specification mechanism is that it does not provide the proper control for the calling of protected subsystems. When a process makes a call that changes its domain of execution, the called domain must have access to the arguments of the call in order perform the desired function. This access should be revoked when the called domain returns, so that the caller can be assured that the callee will not read or modify the arguments at some later time. In addition, the callee should have some way of verifying that the caller has access to the arguments of the call, so that the caller cannot trick the callee into reading or modifying some object to which only the callee has access.

A domain changing mechanism intended for the calling of protected subsystems should require that the callee and caller share some access rights, thus providing some means to pass arguments. Exact Specification and Partial Specification do not enforce such a requirement. Several researchers

[Jo72,Ro74,Sc72] present mechanisms designed specifically to deal with the problem of passing arguments between domains. Any of these mechanisms could be combined with Exact Specification or Partial Specification to form a domain changing mechanism, by using the argument passing mechanism to control access to arguments of cross-domain calls, and using the ACL mechanism to control access to other objects. The Last Component Specification and Appending Specification mechanisms discussed later in this chapter both provide partial solutions to the problem of argument passing that may be significantly easier to implement than the mechanisms of Schroeder and Jones.

5.2.2 Partial Specification.

The second mechanism for authorizing domain changes will be termed here Partial Specification. Domain IDs for this mechanism have a fixed number of components with implied meanings, just as did the Principal IDs of the Multics computer utility described above. These components represent the independent authorities responsible for each domain. A domain object in this mechanism specifies one component of a Domain ID. A Domain gate specifies a complete Domain ID and an initial procedure as before. Domain gates are created by passing to a kernel primitive the name of a procedure and a list of names of domain objects. Each of these domain objects must specify a different component of a Domain ID, and all of them taken together specify the Domain ID of the gate to be created. Domain gates are used in creating processes and calling subsystems as before. New domain objects that specify previously unused Domain ID components can be created by calling the "create_domain" primitive.

Figures 5.1a and 5.1b show one way to use this mechanism to implement the pattern of authorization used in the Multics computer utility as described above. The figures show how the domain and domain gate objects could be maintained in a hierarchical file system, such that each such object is under control of the proper authority. Domain IDs have two components, corresponding to Person and Project. Domain IDs specifying the Person component are of the form Person.*, while those specifying the Project component are of the form *.Project. A Project is created by creating a domain object that specifies component of a Domain ID. A new user can be registered by creating a domain object that specifies the Person component. The ACL's on these objects determine who may use them. The following abbreviations are used for access rights in the figures:

s - (status) Allows a process to obtain information about the objects contained in a directory.

a - (append) Allows a process to create more objects in a directory.

m - (modify) Allows a process to modify information in a directory (including the access control lists for the objects in that directory.)

Notice that the domain Locksmith.SysAdmin is given modify access to the directory ">Users". This access allows a process executing in that domain to obtain access to any of the objects shown in both figures (by modifying ACLs). The Locksmith.SysAdmin domain will have special uses, as shown later.

Figure 5.1a

Domain and Domain Gate Objects in a Hierarchical File System

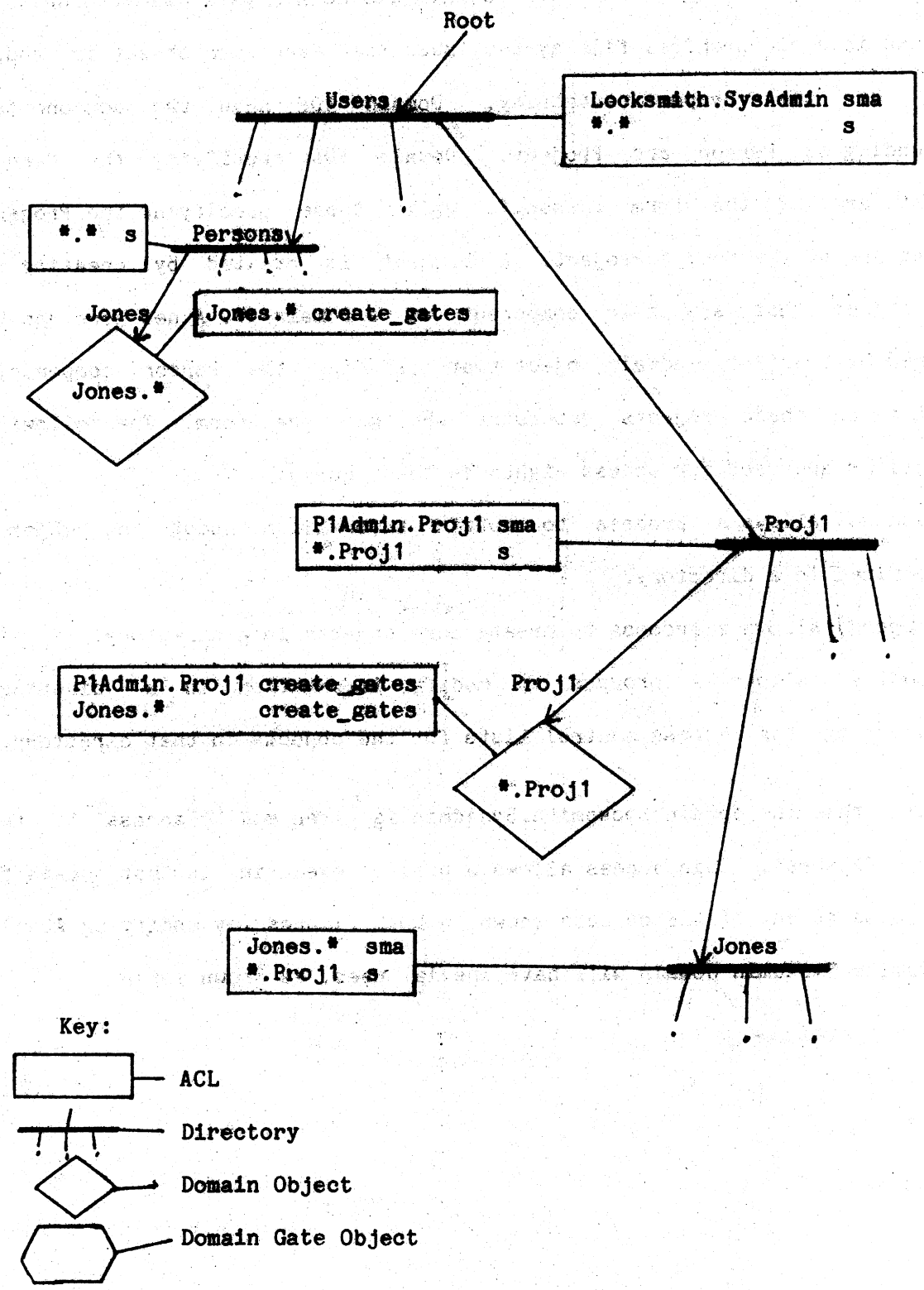
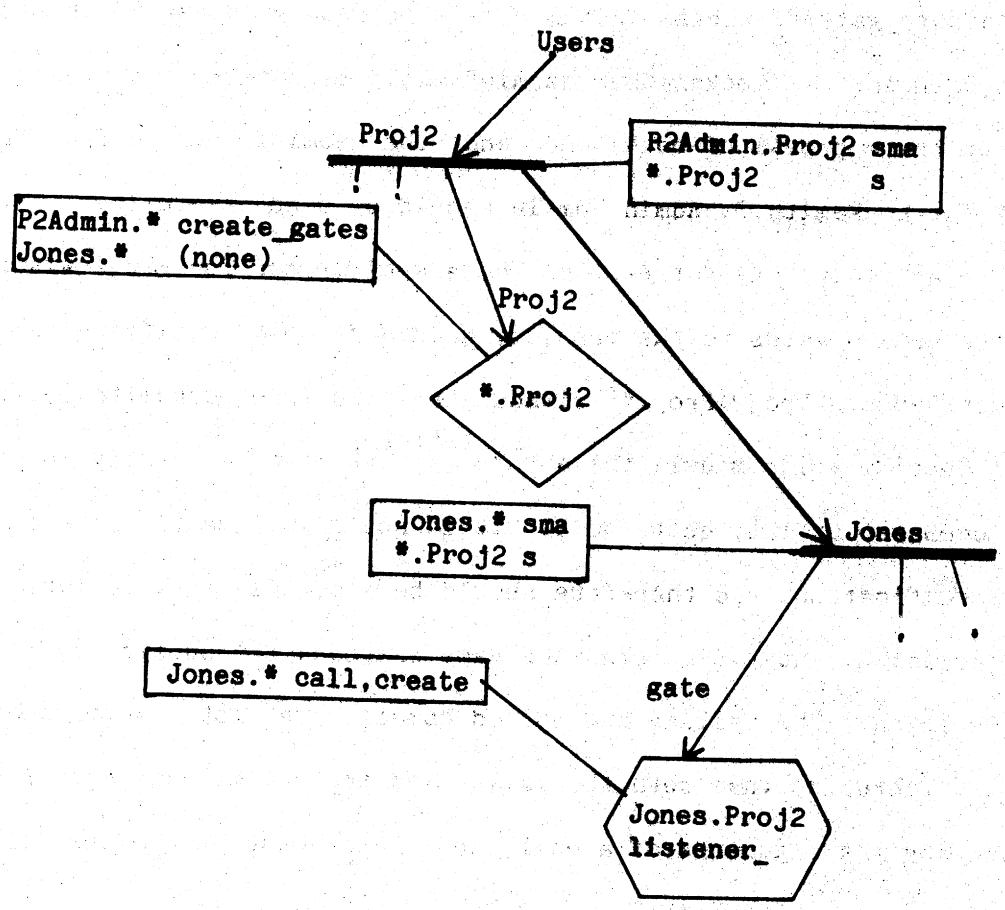


Figure 5.1b

Domain and Domain Gate Objects in a Hierarchical File System



In Figure 5.1a, Jones has been given free access to project Proj1, as he may create new gates into it from any domain with a Domain ID with his name as Person component. These gates can be created by passing the object ">Users>Persons>Jones" and the object ">Users>Proj1>Proj1" to the create gate primitive.

Figure 5.1b shows the hierarchy below the Proj2 directory. Although Jones cannot create new gates into Proj2, he may enter the domain "Jones.Proj2" by using the gate ">Users>Proj2>Jones>gate". This gate had to be created from the domain "Locksmith.SysAdmin", as this is the only domain that has "create_gates" access to the domain objects required to create the gate. The procedures of "Locksmith.SysAdmin" would presumably not create such a gate without the approval of both Jones and the administrator for Proj2. The power of the Locksmith.SysAdmin domain should be used carefully.

Notice that if at any future time the administrator for Proj2 wishes to allow Jones to create gates to the project, he can do so by modifying the ACL on the object ">Users>Proj2>Proj2", without any help from Locksmith.SysAdmin.

Partial Specification models the authorization scheme currently used in the Multics computer utility quite well. It is not significantly more complex than Exact Specification, and therefore should be almost as easy to implement.

This mechanism, however, has the same drawback for subsystem calls as Exact Specification. The calling and called domain are not constrained to share access rights, so that both the caller and the callee must take special action in passing the arguments of a call, and both must be aware of the domain change produced by the call.

5.2.3 Last Component Specification.

The third mechanism to be discussed I will call Last Component Specification. This mechanism cannot be used to authorize domain changes between any two domains, and therefore is not suitable for use in authorizing process initiation. The restrictions made on domain changing by Last Component Specification do, however, make it a more attractive mechanism for

authorizing protected subsystem calls than the first two mechanisms considered in this chapter. As before, Domain IDs have a fixed number of components. Domain and domain gate objects specify only the last of these. (1) A call to a particular gate causes the domain of the calling process to be changed. The Domain ID of the process following the call is formed by replacing the last component of the Domain ID of the calling domain with the component specified by the gate. Thus if a process executing in the domain "Jones.Proj1.home" made a call to a ga as its component, the process would begin to execute the initial procedure of that gate in the domain "Jones.Proj1.editor". New domain objects can be created as before as long as they do not specify the same last component as previously created domain objects.

This mechanism is very similar to that proposed in Schroeder's thesis [Sc72] for controlling the calling of protected subsystems. The last component of a Domain ID can be used to specify a protected subsystem that could be changed by calls during the life of a process. The other components of a Domain ID can be used to specify attributes that remain constant throughout the life of a process, such as the Person and Project components of Multics. All of the subsystems called in a single process are executed in domains that share some access rights (all access rights that can be obtained by the process through ACL terms with "*" as their last component). Although this does not totally solve the argument passing problem discussed before, it does help somewhat by guaranteeing that all of the subsystems in one process share some access rights.

---

(1) We could allow them to specify any one component. The specification of only the last component will, however, be adequate for the intended use of the mechanism and simplifies the description.

5.2.4 Appending Specification.

\The last mechanism I will refer to as Appending Specification. This mechanism is not well suited to process initiation, as it cannot authorize a domain change between any two domains. The domain and domain gate objects specify only one component of a Domain ID, as in Last Component Specification. The Domain ID of the target domain of a call is formed by appending the component specified by the gate to the Domain ID of the calling domain. A return causes the last component of the Domain ID to be dropped. Thus if a process in the domain "Jones.Proj1.home" made a call to a gate specifying "editor" as its Domain ID, the domain of the process would become "Jones.Proj1.home.editor".

We can see that Domain IDs can have different numbers of components with this scheme. We therefore need to augment the rules for matching of Domain IDs and ACL terms to specify what happens when the Domain IDs being matched are of different lengths.

The component "**" has special significance in our matching algorithm, and is used to allow an ACL term to match Domain IDs of various lengths. Before comparing the Domain IDs of the process requesting access and the ACL term, the matching algorithm checks to see if the Domain ID of the ACL has a component of "**". If so, and if the Domain ID of the process has at least as many components as that of the ACL term, then the "**" component is replaced by one or more "*" components so that the Domain ID of the term and that of the process have the same number of components. If the Domain ID of the ACL term has more components than that of the process, the the "**" component is

deleted. We allow each ACL term to contain at most one "**" component. (1) If the Domain ID of the ACL term does not have a "**" component, or if it has more components than that of the process, then the following two rules may apply.

1) If the Domain ID of the Process is longer than that of the ACL term, then they do not match.
2) If the Domain ID of the ACL term is longer than that of the process, then they match only if all of the "extra" components of the ACL term are "*".

Table 5.1 illustrates these matching rules.

Table 5.1

Examples of ACL Term Matching

	Process Domain ID			
ACL term ID	a.b.c.d	a.b.c	a.b.d	c
a.**	match	match	match	no
**.c	no	match	no	match
a.b.*	no	match	match	no
a.b.c.d.**	match	no	no	no

A process can grant access to an object about to be passed by a call by putting a term with the Domain ID of the domain about to be called followed by

(1) Allowing more than one "**" component makes the matching algorithm much more complicated, and makes it difficult for a user to see which Domain IDs match a given term.

".**" on the ACL of the object. In this way, the object will be accessible to the subsystem to be called and any subsystems that it calls. The ACL term need not be removed following the call, as all of the domains that it matches can only be reached by calling the same subsystem again. Thus in a sense the Appending Specification mechanism automatically revokes access following a call.

This control of access to arguments is made possible by the way in which Appending Specification assigns a protected subsystem to a domain. Using Exact Specification or Partial Specification, each protected subsystem is assigned to one domain. Any call to a particular subsystem always enters the same domain independent of the domain of the caller or the process in which the call is made. Thus using either of these mechanisms, the caller must grant access to the callee prior to the call and must later revoke that access. With Last Component Specification, the domain that a particular subsystem enters depends on that process it is called in, but not on the subsystem that makes the call. Thus some objects remain accessible to a process throughout the life of the process, and can be used as arguments to a call with no special handling. With Appending Specification, the domain in which a protected subsystem executes depends on the subsystem that called it. This allows very precise specification of the access rights to be given to each invocation of a protected subsystem.

There are, however, some undesirable effects of not assigning a particular subsystem to the same domain at each call. As each subsystem can be invoked in several domains in each process, Appending Specification will tend to use more domains than the other mechanisms. Each domain requires a certain amount of local storage for local variables. In addition, in a system

that performs dynamic linking, such as the Multics computer utility, the processor time required to link a subsystem in each domain may become expensive.

In addition to the economic objections to not assigning a subsystem to one domain always, one might argue that the environment that is provided by Appending Specification is more difficult to program in. One can have objects that are accessible only to one subsystem (by using ACL terms of the form **.subsystem), only to one person or project (Person.**, or *.Project.**), or only to one invocation (by specifying the exact domain of that invocation in the ACL term). A user must be very careful in deciding the access that he desires for the working storage of the subsystem. Current programming languages do not provide an easy way to specify all of the possible storage classes. For these reasons, while Appending Specification is the most natural of the four mechanisms to use for the calling of protected subsystems, it might not be suitable for all computer utilities.

5.3 Domain Changing and Confinement.

In this section, we discuss two aspects of domain changing in a computer utility that provides confinement. We first consider how to use the domain changing mechanisms of the computer utility to control the assignment of confinement sets to processes. We desire to control the confinement set that a process receives because that confinement set partially determines the objects that the process can read. In some applications of confinement mechanisms to military security, the confinement set of the process may be the only form of access control.

To control the confinement set received by a newly created process, or newly called protected subsystem, we include in the domain gate object the specification of a confinement set. The confinement set assigned to a newly created process or newly called protected subsystem must be contained in the confinement set specified by the gate that was used for process initiation or calling. In addition, we require that the confinement set specified by a gate be a subset of that of the creator of that gate. These two rules insure that the assignment of a confinement set to a process is properly authorized. They do not, however, prevent the domain changing mechanism from releasing confined information.

We now consider how to keep our domain changing mechanisms from being used to release confined information. Lampson [La73] suggests that the channels that can be used to transfer confined information be enumerated, so that they can be individually closed. In this section we enumerate the channels provided by our four domain changing mechanisms, and suggest ways to prevent these channels from being used to release confined information.

With each of the four mechanisms, there are six operations that could be used to release confined information:

1) Domain object creation.
2) Domain gate object creation.
3) Process initiation.
4) Calling of protected subsystems.
5) Deletion of domain objects, or domain gate objects.
6) Modification of access control information for domain objects or domain gate objects.

We now enumerate the channels produced by these six operations.

Domain creation can be used to transmit information in two ways:

1a) The domain object created could carry confined information.

1b) The Domain ID used could carry confined information, and could be observed by other processes attempting to create domain objects.

The first of these channels can be effectively blocked by forcing the creation and use of domain objects to follow the *-property. We assign to each domain object the confinement set of the creator of that domain object, and require that a process have a confinement set that contains that of the domain object in order to use that domain object to create gates. (1)

The second channel is more difficult to close, as all of our mechanisms depend on the fact that the Domain ID in a particular domain object is different from the Domain IDs in all other domain objects. One possible solution is to partition the space of possible Domain IDs among the possible confinement sets. We require that the Domain ID given to a new domain object be a member of the set of Domain IDs assigned to the confinement set of the creator of that domain object. This can be done by including some designation of the confinement set of the creator in the Domain ID. Partitioning the Domain ID space among confinement sets in this manner prevents the observation of the use of a Domain ID by a process with a confinement set not equal to that of the user. Thus the use of a Domain ID cannot release confined information.

---

(1) If the domain and domain gate objects are kept in a hierarchical file system, then the confinement set of the directory containing a domain or domain gate can be used to provide this control.

Gate creation presents one channel for the release of confined information.

2a) The gate that is created could carry confined information.

This channel can be closed in the same manner as the channel described in 1a above was: by enforcing the *-property for the creation and the use of domain gates. (1)

Process initiation presents an additional channel for the release of confined information:

3a) The gate chosen for process initiation can convey information, even if the created process has no means of communicating with its creator.

To block this channel, we must require that the created process have a confinement set that contains that of the creator. There is no way to prevent the gate chosen for process initiation from conveying information. On the other hand, our mechanisms provide no way for the creator to obtain information about the created process. Therefore, there is no reason to force the confinement sets of the creator and created process to be equal.

---

(1) Note that the confinement set associated with a gate in order to enforce the *-property is different from the confinement set specified by the gate. The confinement set specified by a gate was introduced earlier to control the assignment of a confinement set to a process created with that gate. The confinement set introduced above controls the use of the gate, and prevents the use of a gate as a covert channel.

The calling of protected subsystems presents two possible communication channels:

4a) The caller can pass information to the callee by the choice of a gate for the call.

4b) There are a number of ways in which the callee might be able to pass information to the caller.

The first of these channels can be blocked in the same manner as channel 3a above. This means that performing a call to a protected subsystem will never cause the confinement set of a process to decrease.

The problem of keeping a subsystem from releasing information to its caller is shared by all calling mechanisms. Lampson [La73] shows some subtle ways in which information can be released in this way. Rotenberg [Ro74] studied this problem in detail and proposed a partial solution. This thesis does not discuss the problem further.

The deletion of domain objects and domain gate objects, and the manipulation of the ACLs of these objects are operations that modify the directory that contains the object being deleted or the ACL being manipulated. Thus the confinement set of that directory is used to control those operations. [Be73]

From the above discussion, we see that our mechanisms for authorizing domain changes do not violate confinement. An examination of the methods used to prevent the release of confined information reveals, however, that it is impossible to create a gate that crosses confinement sets (i.e. one that is accessible to a process with a confinement set that is different from that specified by the gate). As with other types of objects in a computer utility,

the confinement sets of domain objects and domain gate objects may need to be changed by some trusted authority in order to make the system usable. Such "declassification" is needed with existing confinement mechanisms [Ro74,Be73] as well. The intervention of a trusted authority (person) is needed because programs lack the judgement needed to decide whether or not the object being declassified conveys confined information.

5.4 Choosing Domain Changing Mechanisms.

Of the four domain changing mechanisms that have been presented, we see that none serves well both for authorizing process initiation and protected subsystem calls. We have already suggested one method of obtaining a domain changing mechanism that performs both functions: by combining Partial Specification with an argument passing mechanism similar to those of Jones and Schroeder. Such mechanisms, however, are not easily implemented in existing computer systems.

A second way to obtain a domain changing mechanism is to combine two of our four mechanisms. Using Partial Specification for process initiation, and Last Component Specification for calls, we obtain a mechanism that performs well for process initiation, and provides some help in passing arguments. These two mechanisms can easily be combined. Such a combination does not provide the argument passing capabilities of the mechanisms of Jones and Schroeder, but is significantly easier to implement.

Another combination of domain changing mechanisms that is particularly attractive is that of Exact Specification for process initiation, and Appending Specification for calls. With this combination, all processes are initiated in a domain with a one component Domain ID. Additional components

are acquired by making calls to gates specifying those components. This scheme allows each authority responsible for a particular domain to validate attempts to enter that domain with the initial procedure for the gate that is used to obtain the component corresponding to that authority. With Partial Specification, all authorities must agree on a single initial procedure to be used in validating attempts to enter a domain. This scheme, however, has all of the above mentioned problems of the Appending Specification mechanism.

The variable length Domain IDs (which cause substantial complexity in the implementation of Appending Specification) could be eliminated by restricting the depth of calls, and thus the number of components that a process can accumulate. The current Multics implementation of ACLs allows only three components, and would require substantial modification to increase that number. Three components are not enough to implement the Person and Project authorization of Multics, and allow the coexistence of mutually suspicious subsystems in a single process. At least four components (Person, Project, and one for each subsystem) would be required. Any change in the number of components would also require the modification of the ACLs on objects currently stored by Multics.

Because of the problems mentioned above for Appending Specification, and because Appending Specification would be very difficult to implement for the Multics computer utility, we have chosen to use the combination of Partial Specification and Last Component Specification for the test implementation. This choice was made primarily based on the characteristics of the Multics computer utility, and should not be taken as an indication that this choice is inherently superior.

CHAPTER 6

THE TEST IMPLEMENTATION

6.1 The Multics System.

In this chapter, I describe a test implementation of process initiation for the Multics computer utility, based on the model of this thesis. The chapter begins with a brief discussion of the functions performed by the present implementation of process initiation for Multics, continues with a description of the test implementation, and concludes with an evaluation of the test implementation. For this discussion, It is assumed that the reader has some familiarity with access-control-list based protection schemes, segmented virtual memory systems, and multi-level security systems. No detailed knowledge of Multics is assumed.

The Multics process is implemented as an execution point in a segmented virtual address space. The segments are organized in a hierarchical file system. Each reference of a process to a segment is validated by three access control mechanisms: the Access Control List (ACL) mechanism, the Ring mechanism, and the Access Isolation Mechanism (AIM).

The ACL mechanism implements a list oriented protection scheme with multi-component Principal IDs. The two currently used components stand for Person and Project, two independent authorities that must authorize the creation of a process. The ACL mechanism is hierarchical, in that modification of an ACL for a segment or directory is controlled by the ACL on the directory that contains that segment or directory.

The ring mechanism provides 8 protection rings within each process. The sets of segments that can be read or written in these rings are linearly nested, with ring 0 being the largest set. The ring mechanism is used primarily to protect the Multics operating system.

The AIM mechanism implements a multi-level security system that attempts to prevent the flow of information from a high classification to a lower security classification. The technique used is to prevent operations that spread information, as in our model of confinement mechanisms. The security classifications used are a combination of a level and a compartment within a level.

Process Initiation in Multics.

There are three types of processes created by Multics:

1) Interactive processes, which are created to serve a user at a terminal.
2) Absentee processes, which perform a series of operations for a user from a previously generated script.
3) Daemon processes, which perform system functions and communicate with the operator.

All of these processes are created by a privileged process known as the Initializer. (The Initializer is one of the Daemon processes and is itself created when the system is initialized.) I will now discuss briefly how each of the five functions of process initiation are performed by Multics.

Process Creation.

Processes in Multics are created by the Initializer process executing in ring 0. A process is created with the Principal ID and initial procedure

specified by the Initializer. A directory for the process in which temporary segments for the process will be kept, and several segments in that directory that will be needed to support the process are created at the time that the process is created.

Resource Control.

The following resource control activities take place during process initiation in the current Multics implementation:

1) An account to fund the activities of the new process is located.
2) The Initializer determines whether or not the new process will overload the system and degrade service to other processes.
3) The scheduling parameters, which determine the rate at which a process consumes CPU and memory resources, are determined for the new process.
4) The mechanism that monitors the CPU and memory usage of all processes is informed of the newly created process.

All of these activities take place in the Initializer process in the current implementation. Additional resources may be given to a process after it has been created, but such resource allocations will not be considered here as they are not part of process initiation.

Domain Changing.

The concept of a domain corresponds most closely with the access rights defined by one Principal ID on Multics. There is no single mechanism on Multics that controls the Principal ID given to a new process. This control is accomplished by a complicated set of programs in the Initializer process that decide the initial procedure and Principal ID of the process to be

created. An interactive process can be created with a given Principal ID only if a user who is authorized to use that Principal ID and has satisfied an authentication performed by the Initializer requests such a process. An Absentee process can be created with a given Principal ID only if an Absentee request is received by the Initializer from a process with that Principal ID. A Daemon process with a given Principal ID can be created at the request of the operator.

Authentication.

As noted above, the Initializer must authenticate interactive users in order to determine which Principal ID to assign to the processes that are created for interactive users. This authentication is accomplished by a password check. Presentation of a correct password entitles a user to obtain a process with any Principal ID with the Person component that is authenticated by that password. Each project has a project administrator who is responsible for controlling access to that project. The project administrator maintains a list of users who may use his project. This list provides the authorization for the project component.

Environment Initialization.

The standard initial procedures for Interactive, Absentee, and Daemon processes perform the following environment initialization functions:

1) Initialization of the error condition handling for the process.
2) Attachment of the terminal channel or Absentee script to a command processor.

The proposed removal of the dynamic linking and name space management algorithms from the security kernel of Multics would add the initialization of these mechanisms to environment initialization. [Ja75,Br75] In addition to these activities, one function of environment initialization is currently performed by the Initializer, before a process is actually created. The Initializer creates a home directory for a process if such a directory does not already exist. The Initializer creates the directory, because the process itself does not in general have sufficient access rights to do so.

Summary.

As can be seen from the descriptions above, the mechanisms of process initiation for Multics are highly interdependent. Resource control, domain changing, and authentication are all performed by the same set of programs in the Initializer process, and all use the same data bases (a list of authorized users and their attributes, a list of authorized projects and their attributes, and the lists of authorized users for each project.) At least one part of environment initialization is also performed by the Initializer process and makes use of the same data bases.

In redesigning process initiation according to our model, we attempted to keep these mechanisms separate, while maintaining the functionality of the current implementation wherever possible. We were particularly interested in showing that process initiation for Multics can be implemented in a multi-layered security kernel as argued in the earlier chapters of this thesis.

6.2 An Implementation of Process Initiation for Multics.

In the test implementation, each of the five functions of process initiation is provided by a small program module that executes independently of the modules that provide the other four functions. A sixth module is used to coordinate the activity of the other five. We begin with an overview of the functions performed by each module, and a brief description of how the modules interact to perform process initiation. Later sections of this chapter discuss the implementation issues in each of the modules. Appendix A contains a more detailed description of the programs in each module.

The process creation function in the new implementation is the same as that of the current implementation. Process creation is performed by the Initializer process in ring 0 as before.

Resource control in the test implementation is also very similar to that in the current Multics implementation. The four resource control functions described before are performed in the Initializer process. The programs providing resource control in the test implementation have been simplified by the removal of code that interpreted input from user terminals.

The partial specification mechanism described in chapter five is used to control domain changing. It is implemented as a type manager for domain and domain gate objects, and provides functions that create and interpret these objects. Domain and domain gate objects are implemented as segments that are accessible only in rings 0 and 1. (These will be referred to as ring 1 segments).

In the test implementation, authentication is the responsibility of the initial procedure for a domain. The logger, which initiates processes for interactive users, authenticates each user who contacts the computer utility

for service and records the result as a forwarded authentication. The standard initial procedure for interactive processes uses the forwarded authentication to determine whether or not the user is authorized to use the process. A security conscious user can write his own initial procedure, with whatever authentication mechanism he desires.

Forwarded authentications are also stored in ring 1 segments. They are managed by the authentication forwarding mechanism. The authentication forwarding mechanism restricts access to the forwarded authentications for a stream to those processes that can read or write that stream.

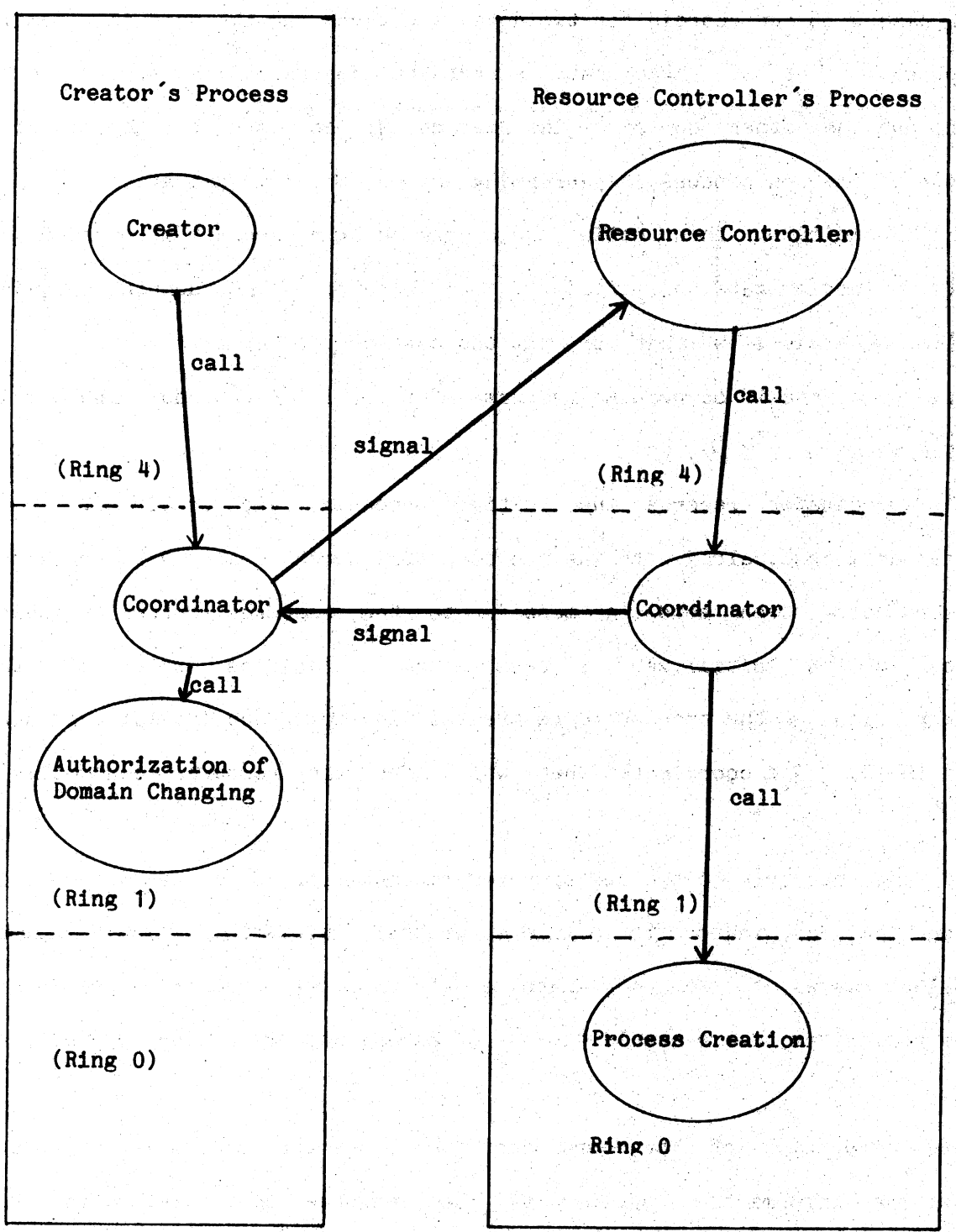
Environment initialization is performed by the initial procedure as before. In addition to the functions described earlier, the standard initial procedure also scans the forwarded authentications as noted above.

In addition to the above modules, there is a coordinator module that coordinates process initiation. The coordinator serves as an interface between modules, which allows the modules to function independently. The coordinator gathers information from the resource controller, the partial specification mechanism, and the process that requests process initiation (the creator). The coordinator distributes this information to the process creation module and the initial procedure for the new process. The information is held in a protected data base while process initiation is in progress.

Figure 6.1 illustrates a typical process initiation.

Figure 6.1

A Typical Process Initiation



Process initiation begins when a process that wishes to create a process (labeled the creator in the figure) calls on the coordinator module. The creator passes to the coordinator two data structures and the name of a domain gate object. One of these data structures describes the process to be created, and the other contains information to be used by the initial procedure of the new process in performing environment initialization.

The coordinator then calls the domain changing mechanism, passing the name of the domain gate specified by the creator. The domain changing mechanism determines whether or not the creator has "create" access to the specified gate, and if so returns the name of the initial procedure and Domain ID of the gate.

The coordinator records the initial procedure and Domain ID in a protected data base, along with the two data structures passed by the creator. The coordinator then sends a message to the resource controller (which executes in the Initializer process) that specifies some of the characteristics of the process to be created (including the initial procedure and Domain ID). The coordinator then waits for the resource controller's reply.

If the resource controller approves the creation of the new process, it calls on the coordinator to complete process initiation. The resource controller passes to the coordinator a data structure containing parameters for the mechanisms that schedule the use of memory and CPU cycles by the new process.

The invocation of the coordinator in the resource controller's process combines the information supplied by the resource controller with that obtained from the creator and the domain changing mechanism, to form a

description of the process to be created. This description is passed to the process creation mechanism. The invocation of the coordinator in the resource controller's process signals the completion of process creation to the invocation of the coordinator in the creator's process.

The above overview leaves many unanswered questions about the functioning of the modules. Later sections of this chapter describe each module in greater detail, and consider the implementation issues in each module.

Process Creation.

The process creation module for the test implementation was taken directly from the current Multics implementation. The set of functions performed by the process creation module of the current implementation was exactly the desired set.

Domain Changing.

As noted before, the current Multics implementation does not contain a mechanism to authorize the use of a domain. The Partial Specification mechanism described in chapter five was used for this purpose in the test implementation. Partial Specification was chosen because it models the two authority authorization scheme used in Multics very well. It also required no changes to the existing ACL mechanism, as Appending Specification would have, nor did it require that the ACLs of objects already in the Multics hierarchy be modified. The domain changing mechanism of the test implementation adopted the strategies discussed in chapter five to prevent the release of confined information by domain changing.

The module that authorizes domain changes is small and simple, and relies on the Multics ACL mechanism in order to perform the authorization.

Domain and domain gate objects are represented by ring 1 segments in the Multics hierarchy. These segments are similar to those used to implement other extended type objects, such as mailboxes and message segments. The Access Control List associated with a ring 1 segment determines which processes can read or write that segment while executing in ring 1. Thus, the ACL mechanism can be used to control the availability of domain and domain gate objects to processes, just as it was in our description of Partial Specification in chapter five.

The domain changing mechanism thus provides operations to create or delete domain and domain gate objects, while access control for these objects is performed by the access control mechanism for segments. Choosing to implement domain and domain gate objects has the disadvantage that each domain or domain gate object must be allocated at least one page (36864 bits) of storage, while in fact each domain object requires only 720 bits and each domain gate requires 1260 bits. The inefficient use of storage was tolerable for the test implementation, but may be a severe problem in a system that supports a large number of domains.

A second responsibility of the domain changing mechanism is to insure the uniqueness of the Domain IDs in the domain objects. For this purpose, the domain changing mechanism maintains a data base that contains all of the Domain IDs in use (contained in domain objects). The data base is protected by a lock to prevent simultaneous updates that could cause duplication. The data base is implemented as a linear list of partially specified Domain IDs, corresponding to the partially specified Domain IDs that are used in the domain objects. The linear list representation was chosen because searches of the data base are infrequent (because domain creation is infrequent) and

because the linear search is much simpler and presumably easier to verify correct than more efficient searching procedures.

Domain IDs are never deleted from this data base, so that they cannot be re-used. This means that the Domain ID data base is constantly growing as more domains are created. The growth was not a severe problem in the test implementation, because the amount of space required for each Domain ID is small (56 characters), and the creation or deletion of domain objects is infrequent.

We need not maintain in the Domain ID data base any Domain ID that does not appear in a domain object, a domain gate object, or an ACL term. The assignment of such unused Domain IDs to new domain objects cannot cause confusion. Thus the file system could be periodically scanned to determine which of the Domain IDs in the Domain ID data base were actually in use. Such a check could be incorporated in the program that scans the file system to verify the integrety of the file system.

In order to implement the multiple authority authorization scheme of Multics, domain objects specifying only the Person component or only the Project component are used. A project domain object by convention is kept in the project directory for that project. Thus the project administrator for a project can control the use of the project by modifying the ACL of the domain object for that project. The person domain objects present a more difficult problem, because the hierarchical access control of Multics makes it difficult to give each user exclusive control over the ACL of his domain object. In our implementation, the person domain objects are all kept in a single directory (>udd>persons). Each has an ACL that allows only the corresponding user's processes to create gates. Modification of the ACL of a person domain object

requires administrative action. This use of the domain changing mechanism is illustrated by figures 5.1a and 5.1b

Authentication.

The test implementation provides authentication forwarding as described in chapter three. and connections made through the Arpa Network.

Chapter three notes that each forwarded authentication should be accompanied by identifying information, so that the user of a forwarded authentication can identify its author. Our implementation of authentication forwarding records the Principal ID, ring number, and process ID of the author and the time of recording for each forwarded authentication. The Principal ID and ring number identify the domain of the author, while the process ID and time form a unique index for the forwarded authentication. Although it would be desirable to record the procedure that produced each forwarded authentication, this information cannot be obtained. (A Multics procedure cannot reliably identify its caller.)

The forwarded authentications are stored in ring 1 segments, so that access to forwarded authentications can be controlled.One such segment is used for each Arpa Network socket or local terminal channel that actually has forwarded authentications.

The use of one segment for each channel allows the forwarded authentications for each channel to be managed independently of those for other channels. Thus a process cannot interfere with the use of forwarded authentications for any channel that that process can not use. Each forwarded authentication requires approximately 2000 bits of storage. Thus, up to 5000 forwarded authentications can stored for each channel.

As noted in chapter three, only those processes that may use a stream should be allowed to read or record forwarded authentications for that stream. Control of forwarded authentications is accomplished in the test implementation by checking the accessibility of the stream before recording or reading forwarded authentications. The accessibility of a stream is checked by requesting the connection status of that stream. The Multics implementation denies status information about a stream to processes that do not have access to the stream.

Three strategies were adopted to insure that forwarded authentications always refer to the current connection of a stream:

1) Each process that has access to a stream may delete the forwarded authentications for that stream.
2) The forwarded authentications for a stream are automatically deleted when that stream is disconnected.
3) A scheme similar to the connection count scheme described in chapter three was implemented.

Any process that believes that the forwarded authentications for a stream that the process has been using are no longer valid can thus delete those forwarded authentications. The second strategy above insures that a forwarded authentication never refers to a previous connection of a stream.

The connection count is not implemented exactly as described in chapter three. This is because we do not want to maintain connection counts for channels not in use, as there are many such channels. Instead, the time at which the last call to connect a channel was made is used as the connection count of that channel. The time is expressed with sufficient precision that

two connections cannot be made to the same channel at the same time. The use of the time of connection as the connection count avoids the necessity of maintaining information for channels that are not connected.

The implementation of forwarded authentications very closely follows the description of chapter three. The programs that implement forwarded authentications are all small and simple.

Authentication Forwarding is used to allow the initial procedure of an interactive process to make use of the standard system authentication mechanism. The logger process authenticates each user who contacts Multics, and records the result as a forwarded authentication. The initial procedure of an interactive process chooses whether or not to believe the forwarded authentication.

Resource Control.

The resource controller for the test implementation was adapted from current Multics implementation of process initiation. The Multics resource controller was adapted to communicate with the coordinator module (described later) rather than with a terminal channel, Absentee request, or the operator. This change did not affect the function performed by the resource controller, but merely changed its source of information.

A second series of changes was made to make the resource controller reject a process creation request that contained unacceptable parameters, rather than attempting to correct those parameters. This change was made primarily because the resource controller cannot alter some parameters, such as the initial procedure and domain of a new process. This change does not alter the resource control constraints enforced by the resource controller.

The resource controller makes use of three privileged operations in order to implement resource control constraints.

1) The resource controller is allowed to monitor the CPU and memory usage of all processes.
2) The resource controller can destroy any process.
3) The resource controller determines the scheduling parameters, which partially determine the rate at which processes consume resources.

These operations do not allow the resource controller to violate access control constraints, as shown in chapter 4.

The Multics resource controller implements a very complex set of resource control constraints, which are designed to give each user a fair share of the computing resources of Multics. The fact that this complex set of constraints can be implemented with only the above three operations suggests that our model can be used for many resource control policies.

The resource controller is a very complex set of programs. Some of this complexity arises from the fact that the resource controller has been adapted from the current Multics implementation, which had other responsibilities in addition to resource control. A great deal of the complexity, however, is inherent in the nature of the constraints being implemented. It is clear that removing this complexity from the access control layer of the security kernel will result in a simpler certification of that layer.

Environment Initialization.

In our model, each domain is responsible for initializing its environment. Environment initialization for a domain is performed by the

initial procedures for that domain, and therefore is under control of the authority responsible for that domain. An initial procedure for interactive processes that performed environment initialization was written for the test implementation. This initial procedure is intended as a demonstration of environment initialization in our model.

The initial procedure performs all of the environment initialization functions mentioned above (initialization of error handling and attachment of the terminal stream to the command processor). In addition, it checks the forwarded authentications for the source of the stream that represents the terminal channel. The forwarded authentications are checked to insure that the identity of the source of that stream had been verified by a trusted authentication procedure, and that the authenticated user corresponds to the Person component of the Principal ID of the new process. The procedure that was implemented trusted any process with the same Principal ID as that of the new process, and also trusted the logger process.

The environment initialization performed by this initial procedure is very simple and straight forwarded. Notice that any desired authentication check could have been made, rather than relying on the forwarded authentications.

The Coordinator.

The coordinator gathers information from the domain changing mechanism, the resource controller, and the process that requests process initiation (the creator). This information is combined to form the parameters given to the process creation module, and to the initial procedure of the new process. The coordinator allows the creator, the domain changing mechanism, the resource

controller, and the new process all to function independently. Several strategies are adopted by the coordinator in order to insure this independence.

Each parameter produced by the coordinator is derived from the information presented to the coordinator in a well defined manner. Thus the domain changing mechanism is given control of the Principal ID, ring number, and initial procedure for the new process, the resource controller is given control of the parameters that determine the rate at which the new process can use CPU and memory resources, and the creator is allowed to pass additional parameters to the new process such as information about the task that that process is to perform.

As can be seen from figure 6.1, the coordinator gathers information in both the creator's process and the resource controller's process. The creator's and the domain changing mechanism's inputs to process initiation are copied into a ring 1 data base before the resource controller is notified of a process initiation attempt. Thus process initiation can be completed even if the creator's process is destroyed before the resource controller acts on the request.

The resource controller is given a limited time to act on each request before the request will be aborted and the information related to it purged from the ring 1 data base. The time limit insures that the coordinator will not have to keep a request indefinitely. It also insures that the resource controller cannot cause confusion by delaying a process initiation attempt until the task that that process was to perform is no longer relevant.

A unique index is given to each process initiation request so that the resource controller and the coordinator do not become confused if two requests

are made for processes with similar characteristics or if the resource controller attempts to respond to a request that the coordinator has given up on and aborted.

The coordinator is a large program, but is simple in structure. The size of the coordinator is primarily due to the number of parameters that must be generated from the available information.

6.3 Conclusions on the Test Implementation.

This chapter has shown how process initiation was implemented for the Multics computer utility. In this section, we compare this new implementation with the current implementation of process initiation for Multics, to see the advantages and disadvantages of our model.

Three advantages of the model are immediately apparent. The first of these is the reduction of the amount and complexity of the programs in each kernel layer. In the current Multics system, any program executing in the Initializer process could potentially create a process with any desired initial procedure and Principal ID. Thus all of the programs that execute in the Initializer process must be considered to be in the innermost layer of the kernel. These programs include not only all of the process initiation mechanism, but also other complicated programs such as those that handle the scheduling of Absentee requests and those that implement the Telnet and FTP protocols of the Arpa Network. Also included in the programs executed in the Initializer process are numerous programs that had been removed from ring 0 with the intent of removing them from the security kernel. In our implementation, the set of programs in each layer of the kernel is well defined and in each case smaller than the set of programs that are in the

Initializer process in the current implementation.

Tables 6.1 and 6.2 show the impact of the model on the size of the Multics security kernel, both in terms of lines of PL/I code, and in terms of the number of modules. The tables include all of the modules related to process initiation, and all other programs that are only included in the kernel because they execute in the Initializer process. The figures for the kernel layers are cumulative. (i.e. The figures for the Denial of Service layer include those for the Access Control layer, and the figures for the Confinement layer include both the other layers.)

The first line of each table shows the current size of the kernel. Because Multics currently has a single kernel layer that implements all of the security constraints, only one number is shown. The second line represents the size of the kernel layers as measured in the test implementation. These figures show a great reduction in the access conrol layer, because many of the programs in the Initializer process need not be included in that layer.

The test implementation did not take full advantage of the simplification that could be achieved by making process initiation unprivileged. Many of the functions performed by the Initializer process in the test implementation do not need to be performed there. The third line of Tables 6.1 and 6.2 estimates the size of each kernel layer in an implementation that took full advantage of the model of this thesis, by removing all unnecessary programs from the Initializer process, and by recoding those that remain to remove functions not related to resource control.

Table 6.1

The Impact of the Model on the Number of Lines of PL/I Code in the Kernel

	Unprivileged	Access Control	Denial of Service	Confinement
Current Multics Implementation	150	<---------- 12000 ----------		>
The Test Implementation	1150	825	10050	10050
A Full Implementation of the Ideas of this Thesis	6600	825	3500	3900

Table 6.2

The Impact of the Model on the Number of Programs in the Kernel

	Unprivileged	Access Control	Denial of Service	Confinement
Current Multics Implementation	3	<---------- 47 ----------		>
The Test Implementation	5	8	43	43
A Full Implementation of the Ideas of this Thesis	17	8	23	27

A second advantage of the model is that every process can request the creation of a new process, whereas only the Initializer can create new

processes in the current implementation. This limitation is the reason that functions such as the Absentee system and the Telnet and FTP protocols of the Arpa Network must be implemented in the Initializer process. This can result in a substantial reduction of the kernel, as approximately 3000 lines of PL/I code are used in the current implementation to provide these functions. These functions, and any new function requiring the creation of processes, need not be performed in the security kernel in an implementation of process initiation based on our model.

A third advantage of the model is that the authority responsible for a domain can control the use of that domain through the initial procedure of the domain. The mechanisms for such control are less apparent in the current implementation.

The test implementation does, however, have several disadvantages. We have already noted that the implementation of domain and domain gate objects is very wasteful of storage. At the time of this investigation the M.I.T. Multics system had approximately 2000 users and 250 projects, and would require a total of perhaps 5000 domain and domain gate objects. These objects would ocupy about 5% of the available permanent storage space. The storage requirement could be substantially reduced if the domain and domain gate objects were supported by the mechanism that implements directories. The data contained in a domain or domain gate object could be placed in the directory containing that object, thus eliminating the need to have a whole segment to hold the representation of such objects. Such an implementation would add some complexity to the programs that implement directories. due to the problems of maintaining the large central data base.

The implementation of forwarded authentications also makes poor use of storage if each stream has only a small number of forwarded authentications. This inefficiency is tolerable, because few streams are connected to Multics at any one time, and forwarded authentications need be maintained only for connected streams.

The implementation based on the model is slightly slower than the current Multics implementation of process initiation. Each process initiation requires about .1 CPU seconds more in our implementation. The extra time is due to the time required to merge the data structures and the time required to format and transmit the message to the resource controller. The total time required for process initiation on Multics is approximately 4 seconds. (Most of this is spent by the resource controller.) The test implementation is thus not significantly slower than the current Multics implementation of process initiation.

The hierarchical access control structure of Multics is in some ways inconsistent with the access control needs for domain and domain gate objects. This inconsistency leads to difficulty in modelling exactly the authorization scheme used in Multics.

Overall, the model has substantially simplified the layers of the security kernel and provided some additional functionality at the cost of using more storage and CPU time, and of forcing users to be careful of the effects of hierarchical access control. Because security is an important goal of the Multics system, this cost can be justified. The following chapter will evaluate the model in the more general context of its use for any computer utility.

CHAPTER 7

EVALUATION AND CONCLUSIONS

In this chapter, we evaulate our model as a whole and draw some conclusions about its usefulness in structuring process initiation. We begin with a comparison of the model with two other process initiation schemes. Following this comparison, we summarize the conclusions about the model. Finally, we discuss topics for further research in the area of process initiation.

7.1 Comparison.

In this section, we compare our model with two common schemes for process initiation: A hierarchical scheme, such as that used in the CAP system [Wa73], and a scheme with central control such as the current Multics implementation of process initiation. These are the most commonly used schemes in current computer systems. We compare the ease with which these three schemes can be used to create processes in the following situations:

1) Creating a process to act for an interactive user at a terminal.
2) Creating one or more processes to carry out some parallel processing algorithm.
3) Creating a process to execute a subsystem that is mutually suspicious with its caller.

In the hierarchical scheme, each process assigns a subset of its resources and a subset of its access rights to each process that it creates. Each process is totally dependent on its creator for resources and access rights. Each process is destroyed when its creator is destroyed. In the centrally controlled scheme, only one process is allowed to create processes. This privileged process controls completely the access rights and resources granted to all processes. The privileged process never terminates.

Process Creation for Interactive Users.

The creation of processes for interactive users was extensively studied in chapter three. Both the model and the centrally controlled scheme handle this situation well. The model, however, offers more flexibility than the centrally controlled scheme. With the model, different processes can be used to create processes for users of different terminals. This capability is useful if the protocols used to talk to different terminals are different. These logger processes need not be certified correct in order to achieve the security goals of the computer utility. The model also allows a security conscious user to protect himself against malfunctions of most of the process initiation mechanism.

The hierarchical scheme of process initiation can also easily be used to create processes for interactive users. The process that responds to requests for processes from interactive users (the logger process) must, however, manage all of the resources required by those users and must be given access to all objects needed by those users. The hierarchical scheme is not readily extended to allow more than one process to create processes for users, as is our model. The hierarchical scheme does not allow the security conscious user

to protect himself from the logger process, because the logger has complete control of the resources and access rights of user processes.

Parallel Processing.

The hierarchical scheme of process creation handles the creation of processes to perform parallel processing for a single user very well. Once an initial process has been created for an interactive user, that process can create additional processes for the user to perform parallel processing. The resources and access rights assigned to the user's first process can be distributed among these processes as needed.

The central scheme requires that each process be created by the privileged process. The privileged process may not provide the resources or access rights needed by the user, as it has less knowledge of the task to be performed than does the user's initial process. The central scheme does, however, provide a better opportunity to control the total number of process in the computer utility. As noted in chapter four, such control is needed to insure that the resource controller can respond rapidly to demands for resources. Most current computer systems impose limits on the total number of processes.

The model shares some of the drawbacks of the central scheme, but provides somewhat more flexibility than that scheme. Like the central scheme, our model has one central resource controller that is responsible for all resource allocation. As before, the central resource allocator must participate in each process creation, and may not provide exactly the desired resources. The resource controller can, however, control the number of processes in the computer utility, as in the central scheme.

Access rights in our model, however, are not under control of a central authority. The domain changing mechanism provides precise control over the creation of processes, and over the assignment of access rights to processes. Thus the use of parallel processes by a user can be controlled by controlling access to the domain and domain gate objects for that user's domain. The availability of parallel processing to a user may also depend on the task to be performed, as the initial procedures specified by the gates into the user's domain may restrict the tasks that the user can perform.

Mutually Suspicious Subsystems.

The protection of mutually suspicious subsystems is one of the most interesting and difficult computer protection problems. Schroeder presents a mechanism that allows mutually suspicious subsystems to cooperate in a shared process. This mechanism does not guarantee each subsystem a fair share of the resources of the process, and thus one subsystem may deny service to others in the same process. By providing separate processes for such subsystems, we can eliminate the problem of denial of service.

The model of process initiation of this thesis is ideal for the creation of processes to execute mutually suspicious subsystems. The domain changing mechanism allows the owner of a subsystem to control the calling of that subsystem, while the central resource control mechanism allows the resources of the caller and callee to be separately managed. Thus neither the caller nor callee need trust the other.

In the central scheme, all processes are created by the privileged process. Thus each creation of a process for a protected subsystem involves communication with the privileged process. The privileged process must

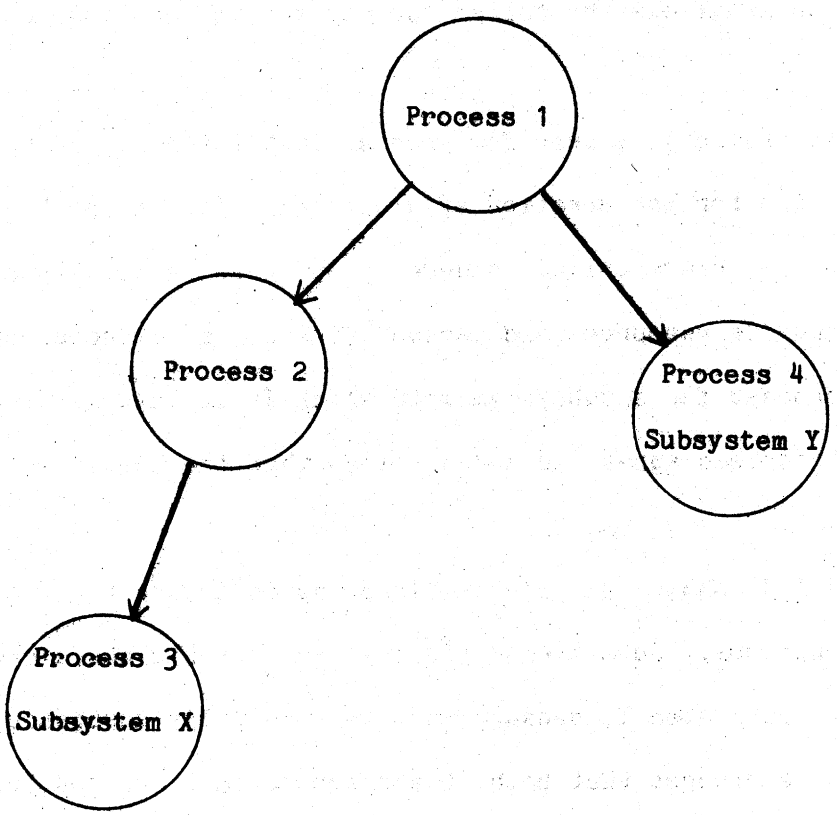
implement some control over the creation of processes for protected subsystems similar to that of our domain changing mechanism. There must also be a secure communication mechanism that allows each process to communicate requests for processes to the privileged process. All protected subsystems must trust the privileged process to provide the correct access rights and resources. The central mechanism allows the caller and callee to be independent, as does model.

The hierarchical scheme for process initiation is the most difficult of the three to use for the creation of a process for a protected subsystem. Because in the hierarchical scheme a process is totally dependent on its creator to provide resources and access rights, a process cannot directly create a process for a subsystem with which it is mutually suspicious. Each process must instead appeal to some process that the subsystem to be executed trusts.

Figure 7.1 shows a process hierarchy including two processes that are mutually suspicious. Subsystem X (in process 3) could not directly create a process for subsystem Y, because they were mutually suspicious. Subsystem X had to locate a process that both it and subsystem Y trusted (process 1 in the example) to create the process for Y.

Figure 7.1

Hierarchical Process Creation for Mutually Suspicious Subsystems.



As with the central scheme, secure communications are needed, and each process that creates processes for protected subsystems must implement some control scheme. If only the process at the top of the hierarchy creates processes for mutually suspicious subsystems, then this scheme reduces to the centrally controlled scheme. The hierarchical and central schemes for process initiation are both more awkward to use for the creation of processes for mutually suspicious subsystems than the model of this thesis.

7.2 Conclusions About the Model.

In this section we summarize the advantages and disadvantages of our model. Some of these observations have been discussed at length in other sections and are only briefly mentioned here.

As can be seen from the preceding section, the model handles the creation of processes for interactive users and for mutually suspicious subsystems very well. It provides more flexibility than the other two schemes considered, while forcing users to rely on less of the process initiation mechanism of the computer utility. The model performs less well than the hierarchical scheme for the creation of processes for parallel processing. The model does, however, provide control that the hierarchical scheme does not. The resource controller of the model can easily control the total number of processes so that it can respond rapidly to changing resource requirements, and the domain changing mechanism can be used to control the tasks for which each user may use parallel processes.

Another benefit of our model is that it separates the mechanisms that perform the five functions previously identified: Process creation, domain changing, authentication, resource control, and environment initialization. This separation allows each function to be implemented in a small program module, independent of the other functions. The structure achieved by using small independent modules is easy to verify, and easy to modify.

The model also shows the security constraints that can be violated by the programs that implement each function. Thus we can clearly see which of the modules must be certified correct in order to achieve the security goals of a given system. In the test implementation for the Multics computer utility, we

saw that the size and complexity of the programs that must be certified to achieve the security goals of Multics are both reduced in the implementation based on the model.

Another benefit of the modularization of the model is that it allows any process to create processes. Unlike the hierarchical scheme, the sets of resources and access rights of a process are not restricted to be subsets of those of the creator of that process. Thus any application that requires the creation of processes can easily be implemented in a computer utility using our model, without modifying the process creation mechanism, or the security kernel.

One of the primary drawbacks of the model is the problem of maintaining the domain and domain gate objects for the domain changing mechanism in an efficient manner. In our test implementation, we chose to use very simple management techniques that wasted a large amount of storage. Objects with small representations are inefficiently supported by current hardware technology. This forces the implementor to abandon the hardware protection mechanism for small objects if they must be efficiently implemented. Providing equivalent protection in software greatly increases the size and complexity of the programs that manage such objects. Newer hardware organizations, such as that of the CAP processor [Wa73], make better provison for small objects.

A second drawback is that the controls provided by the model over process initiation may be somewhat awkward to use. We saw in the test implementation that the hierarchical access control mechanism of Multics made it difficult to give each user complete control of his home domain. Each user must be very careful in creating domains and gates. The accessibility of all of the

directories above a given object must be considered in determining the accessibility of that object.

The initial procedure of a domain must also be carefully coded to ensure proper use of that domain. The authentication forwarding mechanism allows the initial procedure to trust a central authentication mechanism to ensure proper use of the domain. Our model achieves a smaller and simpler security kernel by allowing the user to protect himself. Thus there is a greater probability that the protection facilities of the computer utility will be misused and not provide the desired security constraints.

Finally, the argument that authentication and environment initialization can be removed from the security kernel in our model is somewhat deceptive. Clearly, in the test implementation the security of the entire system depends on the authentication and environment initialization performed by the initial procedure used to enter the Locksmith domain. The existence of such privileged domains forces all users to depend on the programs that execute in those domains, much as the security of the entire system is dependent on the compilers and editors used to produce the programs of the security kernel. The privileged domains are infrequently used, and auditing the use of privileged domains may be sufficient to provide security.

7.3 Topics for Further Research.

This thesis leaves several problems in the area of process initiation unsolved. In this section, we briefly describe those problems.

Our model identifies five independent functions of process initiation. The test implementation demonstrates one way in which these five functions can be coordinated to perform process initiation. We did not explore extensively other organizations. (One such organization would require that each process begin execution in the domain of its creator. All domain changes would be accomplished by cross-domain calls. Such an organization may provide an implementation of process initiation that is even simpler than that chosen for the thesis.)

This thesis did not consider many of the problems associated with allowing users to create processes. We did not present a resource control scheme to insure that receives a fair share of the available resources, independent of the number of processes that he is using. The resource control mechanism of Multics does not provide this guarantee. Developing such a resource control scheme, and demonstrating that it can be implemented in our process initiation structure would be an interesting research project.

The thesis presents a novel authentication scheme for confinement systems. The test implementation did not test some of the ideas presented. In addition, it is not entirely clear how this scheme interfaces with authentication mechanisms based on encryption. A recent masters thesis [Ke76] investigated the use of encryption in providing secure communication channels. The protocols developed fit well with the authentication scheme of this thesis. Some further work may be needed, however, to bring together all of the ideas about authentication in these two theses.

APPENDIX A
DETAILS OF THE IMPLEMENTATION

This appendix presents a more detailed description of the test implementation than is given in the text. The appendix is organized in sections, each section devoted to one of the functions of process initiation discussed in the text. Each section describes the programs that implement the corresponding function and the data structures that are used by those programs.

Each of the programs described is a PL/1 procedure, possibly with multiple entry points. The function performed by each entry point is briefly described, along with the function of the entire program. The contents of the data structures are described, but not the detailed format.

Process Creation.

Programs:

hphcs_$create_proc:
This is the entry to the programs that actually create processes. As stated in the text, this function of process initiation was taken from the current implementation. This program takes two data structures as arguments, create_info, and pitmsg. The create_info structure describes the process to be created and is described below, while the pitmsg structure is not used during process creation and is passed to the programs that perform environment initialization. The pitmsg structure will therefore be described in the environment initialization section.

Data Structures:

create_info
The create_info structure contains the following information:

Principal ID for the new process,

Initial and highest ring numbers for process,

AIM clearance for process,

Maximum AIM clearance for process (not respecting the limit requested when the process was created),

Audit checking flags,

Process ID for new process (half specified by creator and half filled in by process creation),

Process ID and trouble report channel,

Pointer to and length of the pitmsg structure for this process,

Record quota for storage in the process directory for the new process,

Location and maximum length of the linkage offset table, combined linkage segment, and known segment table for the new process,

Scheduler work class for this process.

Environment Initialization.

Programs:

user_init_admin_:
This is the first program that gets called in the user ring in a newly created interactive process. It is an assembly language program whose only function is to call user_real_init_admin_ and process_overseer_. These calls are performed because the first program called in a process cannot return until the process terminates, and therefore leaves a frame on the stack for the life of the process. As much of the work of environment initialization as is possible is done in programs that can return and thus release their stack frames.

user_real_init_admin_:
This program obtains a pointer to the pitmsg structure for the process. (This structure was placed in the process directory by process creation). The program also initializes the process's communication channel to the user that requested the process, and finds the system process_overseer_ program, or a user specified process overseer. user_real_init_admin_ also establishes error handlers for certain error conditions that are handled by the same programs throughout the life of the process. user_real_init_admin_ makes use of the information in the pitmsg data structure that is described below.

process_overseer_:
This is the standard initial procedure for interactive processes. It first establishes a handler for any error conditions that occur during the life of the process and are not handled by other programs. Then, it scans the list of forwarded authentications for the communication channel of the process. If an authentication that was performed either by a trusted system procedure, or by a process with the same Principal ID as that of the new process can be found, and if that authentication identifies the correct user (the one who matches the first component of the Principal ID of the new process), then execution proceeds. Otherwise, the process is terminated.

If the authentication check is successful, then process_overseer_ prints the system message of the day, and executes the users "start up" commands. process_overseer_ then calls the command listener to wait for commands from the user.

Data Structures:

pitmsg

The pitmsg structure contains the following information:

Process type (interactive, absentee, or daemon),

Home directory,

Process creation time,

Login time (may be different from above if several process are created for a session with one user),

Login line,

Name of terminal channel,

I/O module needed to use terminal channel,

AIM access class of terminal channel,

System control attributes of this process,

Load control information for this process,

Summary of previous usage of the processes account (supplied by the resource controller),

Additional information for absentee processes.

Domain Changing.

Programs:

dm_:

dm_ is a gate used to call the domain and domain gate object managers. Below is a list of the entries to dm_ and the programs that they call.

entry	program called
dm_$create_domain	domain_manager_$create_domain
dm_$create_gate	domain_manager_$create_gate
dm_$interpret_domain	domain_manager_$interpret_domain
dm_$interpret_gate	domain_manager_$interpret_gate
dm_$delete_domain	domain_manager_$delete_domain
dm_$delete_gate	domain_manager_$delete_gate
dm_$add_dom_acl_entries	domain_manager_$add_dom_acl_entries
dm_$add_gate_acl_entries	domain_manager_$add_gate_acl_entries
dm_$delete_dom_acl_entries	domain_manager_$delete_dom_acl_entries
dm_$delete_gate_acl_entries	domain_manager_$delete_gate_acl_entries
dm_$list_dom_acl	domain_manager_$list_dom_acl
dm_$list_gate_acl	domain_manager_$list_dom_acl
dm_$replace_dom_acl	domain_manager_$replace_dom_acl
dm_$replace_gate_acl	domain_manager_$replace_gate_acl
dm_$make_process	initiate_process_$initiate_process_

domain_manager_:

This program is the manager for objects of type domain, and domain gate. The program has several entry points that allow the creation, deletion, and access control list manipulation of these objects. The program uses the domain, domain_gate, and domain_list structures described below.

domain_manager_$create_domain:

This entry point creates a domain object. The entry point takes the directory pathname and entry name desired for the domain object to be created, the desired ring number, and the desired Principal ID. The Principal ID is checked to insure that it does not duplicate a previously specified Principal ID in any component. For this purpose, domain_manager_ maintains a list of all Principal IDs currently in use in the domain_list data base. If the Principal ID is acceptable, then a segment is created in the specified directory with the specified entry name suffixed by ".domain". This segment is accessible only in ring one and contains the domain data structure described below.

domain_manager_$create_gate:

This entry point creates domain_gate objects. It takes as arguments, the directory and entry name for the desired domain gate, a list of domain objects that determine the Principal ID of the gate, a ring number, an AIM authorization for processes created with the gate, and the name of an initial procedure. If the set of domain objects correctly specifies a Principal ID, then a segment is created in the desired location with the desired name suffixed by ".domain_gate". This segment is accessible only

in ring 1 and is used to contain the domain_gate structure described below. The gate specifies the given initial procedure, the maximum of the caller's ring, specified ring, and the ring contained in each of the specified domain objects. The AIM clearance specified by the gate is the minimum of the caller's clearance, the specified clearance, and the clearances of all of the domain objects.

domain_manager_$interpret_gate,
domain_manager_$interpret_domain:
These entry points return the information contained in domain and domain gate objects, provided that the caller has the proper access (p for gates, and c for domains).

domain_manager_$delete_domain, domain_manager_$delete_gate:
These entry points delete domain and domain_gate objects.

domain_manager_$add_dom_acl_entries,
domain_manager_$add_gate_acl_entries,
domain_manager_$delete_dom_acl_entries,
domain_manager_$delete_gate_acl_entries,
domain_manager_$list_dom_acl,
domain_manager_$list_gate_acl,
domain_manager_$replace_gate_acl,
domain_manager_$replace_dom_acl:
These entry points perform ACL manipulation for domain and domain gate objects. They have similar interfaces to the entries in hcs_ that perform ACL manipulation for segments.

create_domain, create_gate, delete_domain, delete_gate, status_domain, status_gate, list_acl_domain, list_acl_gate, set_acl_domain, set_acl_gate:
These are all entry points to a program that implements user commands for manipulating domain and domain gate objects. They will not be described in detail.

Data Structures:

domain:
The domain structure is used to implement a domain object, and contains the following information.

Person component of Principal ID for this domain (* means unspecified),

Project component of Principal ID for this domain (* means unspecified),

Ring number of domain,

Creation time of domain.

domain_gate:
The domain gate structure is used to implement domain gates and contains the following information.

Person component of Principal ID of the domain of the gate,

Project component of the Principal ID of the domain of the gate,

Ring number of the domain of the gate,

AIM authorization specified by the gate,

Initial procedure of the gate,

Flag indicating whether or not the initial procedure should be called before the I/O attachments and static condition handlers of the process are initialized (before user_real_init_admin_ is called).

domain_list:
The domain_list structure is used to keep a record of the Principal IDs currently in use. It has a header that contains a lock and the number of entries. Each entry contains the following information:

Person component of the Principal ID,

Project component of the Principal ID,

Pathname of the domain object that specifies this Principal ID.

Authentication Forwarding.

Programs:

asm_

asm_ is a gate used to access the authentication forwarding mechanism. Below is a list of entries to asm_ and the programs that they call.

entry	program called
asm_$tty_assert	assertion_manager_$tty_assert
asm_$tty_read_assertions	assertion_manager_$tty_read_assertions
asm_$tty_delete_assertions	assertion_manager_$tty_delete_assertions
asm_$ncp_assert	assertion_manager_$ncp_assert
asm_$ncp_read_assertions	assertion_manager_$ncp_read_assertions
asm_$ncp_delete_assertions	assertion_manager_$ncp_delete_assertions
asm_$priv_net_assert	assertion_manager_$priv_net_assert

hcs_, net_, netp_:

These are the gates through which the primitives that manipulate local terminal channels and ARPA network channels are reached. Several entries in these gates were changed to call entries in r1tty_ instead. This is done to maintain the index data bases used by r1tty_, and to notice when these channels are connected and disconnected. The following entries were changed:

entry	program called
hcs_$tty_index	r1tty_$tty_index
hcs_$tty_order	r1tty_$tty_order
net_$ncp_activate	r1tty_$ncp_activate
net_$ncp_connect	r1tty_$ncp_connect
net_$ncp_order	r1tty_$ncp_order
netp_$priv_net_activate	r1tty_$priv_net_activate

assertion_manager_:

This program manages forwarded authentications. It does so by maintaining a segment for each channel connected to the system containing the forwarded authentications for that channel. The format of these segments is described by the assertion_seg data base. These segments are kept in the directories >system_control_1>assertions>tty_seg, and >system_control_1>assertions>ncp_seg, and are accessible only in ring 1. There are three entries to assertion_manager_ for each function, one for local channels, one for network channels, and one for privileged manipulation of network channels.

assertion_manager_$tty_assert,
assertion_manager_$ncp_assert,
assertion_manager_$priv_net_assert:

These entries record forwarded authentications. They take as input the name of a channel, the asserted user name, and an uninterpreted string of "extra" information. They call entries in r1tty_ to translate from the

name of the channel to the index for the channel needed to determine the state of the channel. The state is then obtained in order to insure that the caller has access to the channel and that the channel is still connected. If these conditions are met, the forwarded authentication, along with information identifying its author, is recorded in the assertion_seg for the channel.

assertion_manager_$tty_read_assertions,
assertion_manager_$ncp_read_assertions,
assertion_manager_$priv_net_read_assertions:
These entries extract the forwarded authentications for a channel. They take the name of a channel, and convert and verify it as above. If the channel is accessible, as many forwarded authentications as will fit in a list supplied by the caller of assertion_manager_ are returned, along with a count of the total number of forwarded authentications present. If the verification of the specified channel reveals that the channel is disconnected, the assertion_seg for that channel is deleted, and an error code is returned.

assertion_manager_$tty_delete_assertions,
assertion_manager_$ncp_delete_assertions,
assertion_manager_$priv_net_delete_assertions:
These entry points delete the forwarded authentications for a channel. They are provided to allow any program that detects that such authentications are no longer valid to delete them. The same verification procedure is used as before, and the appropriate assertion_seg is deleted.

r1tty_:
This program serves two purposes. First, it maintains data bases to translate between channel names and channel indices. Second, it notices requests to connect channels and calls assertion_manager_ to delete the assertion_seg for any successful attempt. It maintains two data bases, >system_control_1>ncpxs, and >system_control_1>ttyxs, that are described below.

r1tty_$get_ttyx, r1tty_$get_ncpx:
These entries obtain the index for a channel name. If the named channel is not known to the system, an index of 0, which is invalid, is returned.

r1tty_$get_tty_name, r1tty_$get_socket_num:
These entries return the local channel name or network socket number of a given index. If the index is invalid, an invalid name or socket number is returned.

r1tty_$tty_index,
r1tty_$ncp_activate,
r1tty_$priv_net_activate:
These entries record the index assigned to a channel name. They call the supervisor to obtain the index.

r1tty_$n$tty_order, r1tty_$ncp_order, r1tty_$ncp_connect:
These entries check for orders to connect channels. If such an order is made, the assertion_seg for the channel is deleted by a call to assertion_manager_.

Data Structures:

ncpxs, ttyxs:
These two data bases are used to maintain the index mapping. Each contains a lock, a length, and a list of entries giving the name for each index currently in use.

assertion_seg:
An assertion_seg is maintained for each channel with forwarded authentications. Each assertion_seg contains a lock, the number of forwarded authentications, followed by a list of forwarded authentications. Each forwarded authentication contains the following information.

Time of recording of this authentication,

Principal of the recording process,

Process ID of the recording process,

Ring number of the recording process,

Authenticated user name,

Extra, uninterpreted information supplied by the author of the forwarded authentication.

Resource Control

Programs:

user_process_manager_:
The current implementation of resource control for Multics was adapted to run as the resource controller of the new implementation. user_process_manager_ acts as the resource controller for the new implementation. It calls on the resource control programs of the old implementation to perform specific resource control functions. Some of those programs are briefly described in this section. user_process_manager_, and all of the other programs of the resource controller make use of a data base known as the answer_table. This table contains entries for each process that describe that process's resource limitations and allow the resource controller to obtain the resource usage statistics for that process. In addition, some of the programs make use of the system administrator's table (SAT), person name table (PNT), and project definition table (PDT). These data bases contain resource control parameters for projects, users, and specific user.project combinations.

user_process_manager_$upm_init:
This entry initializes the resource controller. It calls the coordinator for process initiation to establish itself as the resource control process and to abort any process initiation attempts in progress.

user_process_manager_$upm_request:
This entry point responds to a request to create a process. It establishes an entry in the answer_table for the new process, and calls on other resource control programs to verify that there is an account to fund the process and to begin accounting procedures for CPU and memory usage by the new process. Eventually, the coordinator is called to finish the creation of the new process.

user_process_manager_$upm_event:
This entry point responds to events relevant to a process after that process has been created. It is invoked when messages are recieved from a process's trouble report event channel, which are used to report processes that have become damaged or have terminated. It is also invoked when other resource controller programs decide to terminate a process. If the AIM rules allow, user_process_manager_ forwards messages that it recieves for a process to the trouble report channel specified by the creator of that process.

lg_ctl_:
This module locates the entries in the SAT, PNT, and PDT data bases that apply to a particular process. It applies the limits found in these entries to determine whether or not the process under consideration will be created. It also maintains a data base that all processes can read that contains a list of all currently executing processes. lg_ctl_ calls load_ctl_ and act_ctl_ in order to insure that the proposed process will not overload the system and that it has an account to fund its CPU and

memory usage. There are two entries to lg_ctl_: lg_ctl_$upm_in, which is called by user_process_manager_ to check a process before it is created, and lg_ctl_$logout, which records the termination of a process.

load_ctl_:
This program limits the number of processes on the system at any one time.

load_ctl_$load_ctl_:
This entry point is called by lg_ctl_ for each request to create a process. It decides whether or not to allow the new process to be created, and whether or not to preempt existing processes for the proposed new process.

load_ctl_$unload:
This entry point is called to record the termination of a process.

act_ctl_:
This program records the resource usage of all processes. The resource usage information for a particular process is maintained in the PDT entry corresponding to the person and project of that processes Principal ID. There are several entry points to act_ctl_.

act_ctl_$check:
This entry point checks to see that a valid account exists for a proposed process. It also checks that the account for a proposed process is not yet out of funds.

act_ctl_$open_account:
This entry point opens an account for updates. It must be called before account for a process can be initiated.

act_ctl_$cp
This entry point instructs act_ctl_ to begin monitoring the resource usage of a process.

act_ctl_$update:
This entry point updates the resource usage statistics for all processes being monitored. It is called periodically in order to keep records up to date in the event of a system failure.

act_ctl_$dp:
This entry point informs act_ctl_ that a process has terminated and that it should no longer monitor that process.

act_ctl_$close_account:
This entry point closes an account and makes it unavailable for updates until it is re-opened.

cpg_:
This program constructs the create_info and pitmsg structures for a process. It fills in the resource control control items in both

structures from information available in the answer_table, SAT, PNT, and PDT entries for that process.

Data Structures.

answer_table:
The answer_table contains one entry per process, and is used to record information about that process. It also has a header that contains miscellaneous information and will not be described. Each answer_table entry contains the following information:

A state, that indicates whether the entry is free, in use by process initiation, or used by a process that has already been created,

The sizes and locations of the linkage offset table, combined linkage segment, and known segment table for this process,

The trouble report event channel,

The process ID of the process,

The time at which the request for this process was received,

Miscellaneous attributes of this process,

A pointer to the PDT entry for this process,

The scheduler work class for this process,

The person and project components for this process,

The name of the initial procedure for this process,

The time of the last accounting update of this process,

The CPU and memory usage of the process up to the last update,

The time to wait before preempting this process for another.

SAT:
The SAT has a header that contains parameters used by load_ctl_ to determine how many users to allow on the system. In addition, it has one entry per project that contains the following information:

Project name,

Pointer to PDT for that project,

Number of users authorized to use this project,

Maximum number of such users,

Miscellaneous limits on users of that project,

Default load_ctl_ parameters for processes from that project.

PNT:

The PNT is a list of all of the users who may use Multics. It has one entry per user, that contains the following information:

AIM authorization for this user and all his processes,

Audit flags for this user and his processes,

User name.

PDT:

There is one PDT data structure for each project. Each PDT contains entries describing the users who may use that project and charge to its account. Each of these entries has the following format:

Name of user,

Number of processes that the user currently has,

Miscellaneous limits on the user's processes,

Limited initial procedure for user (can be specified by the project administrator to limit the user's resource consumption. This does not force the user to use that initial procedure, but denies him the use of the project unless he does),

Default home directory (used only if process creator doesn't specify a home directory),

AIM authorization for user's processes,

Summary of the resource usage of the user in the project.

Coordination.

Programs:

proc_creat_:

proc_creat_ is a gate used by the resource controller to call the coordinator for process initiation. It is accessible only to valid resource controller processes. Below is a list of the entries to proc_creat_ and the programs that they call.

entry	program called
proc_creat_$initialize	initiate_process_$initialize
proc_creat_$notify	initiate_process_$notify
proc_creat_$create	initiate_process_$create

initiate_process_:

initiate_process_ is the program that provides coordination among the modules of process initiation. This program assembles create_info and pitmsg structures, to be used in creating a process, from data supplied by the domain changing mechanism, the resource controller, and the process that requested process initiation. There are four entry points to initiate_process_ that are described below.

initiate_process_$initiate_process_:

This entry point begins process initiation. It can be called by any process (through the dm_ gate) and takes three arguments: a create_info structure, a pitmsg structure, and the name of a domain_gate object. The entry point dm_$interpret_gate is called to determine whether or not the calling process has "p" access to the gate, and to extract the Principal ID and initial procedure from the gate. The supplied pitmsg and create_info structures are then copied to a protected segment so that they cannot be changed while the resource controller decides whether or not to allow the process to be created. Parameters from these structures needed by the resource controller are then placed in a pr_rq data structure and sent to the resource controller (through the use of the Multics message_segment facility).

initiate_process_ then waits for a message from the resource controller, or a timeout. Because initiate_process_ executes in ring 1, this effectively blocks the creating process until the resource controller is finished. This blocking reduces the chance that the creating process will terminate before process initiation is complete. (The implementation recovers from such an occurance, but it is unpleasant and clearly undesirable.) The signal sent by the resource controller contains an indication of the success or failure of the attempt to create a process. On receipt of the signal, initiate_process_ returns to its caller. If the creation was successful, then the creating process must send a signal to the created process in order to begin its environment initialization. A new process is blocked until it receives such a signal so that the creating process can pass resources (terminal channels in particular) to the new process before environment initialization is attempted. If the creating process does not send such

a signal, the resource controller will do so eventually to prevent the new process from staying blocked indefinitely. initiate_process_ maintains a list of all pending process initiation in the pending_creates data structure described below.

initiate_process_$create:
This entry point is called by the resource controller to finish the creation of an approved process. The arguments to this entry point are a create_info structure, a pitmsg structure, and the index of a process initiation request. The pitmsg and create_info structures supplied by the creating process for the specified request are found and compared with those supplied by the resource controller. All of the entries that represent information supplied to the resource controller in the pr_rq message must match. This matching is done to keep the resource controller from becoming confused when requests are timed out by the creating process, and because some of the resource controller programs replace unacceptable parameters in a process creation request rather than rejecting the request. The resource control attributes are then taken from the resource controller's pitmsg and create_info data structures and placed in the structures copied from those supplied by the creating process. hphcs_$create_proc is then called to create the specified process. If the creation is successful, then a signal is sent to the creating process.

initiate_process_$notify:
This entry point is used by the resource controller to abort an unsatisfactory request for process initiation. It takes as arguments an error code and a request index. The error code is signalled to the creating process for that request.

initiate_process_$initialize:
This entry is used by the resource controller to initialize process initiation. It aborts all pending requests for processes and establishes the calling process as the resource controller (so that the signals will be sent to the proper process).

Data Structures:

pending_creates:

The pending creates data base is used by initiate_process_ to keep track of process creation requests that have been signalled to the resource controller and are awaiting approval. It has a header that contains the following information:

A lock to prevent simultaneous access,

The process ID of the resource controller for signalling,

The next index to use for a process creation request,

The location of a directory in which to keep pitmsg structures.

pending_creates also has one entry per pending request. These entries contain the following information:

A flag indicating whether or not this entry is in use,

The time at which this request was made,

The index of this request,

An event channel to be used for signalling from the resource controller to the creating process,

The process ID of the creating process,

A copy of the create_info structure supplied by the creating process with attributes obtained from the domain_gate replacing the corresponding attributes supplied by the creating process.

pr_rq:

This data structure is used to pass a request for process creation from the creating process to the resource controller. It contains the following information:

The index of this request,

The trouble report channel specified by the creator (the resource controller forwards trouble reports to this channel),

The process ID of the creator,

Principal ID desired for the process,

Home directory for the process,

Initial procedure for the process,

Initial and highest ring numbers for the process,

Requested AIM authorization (minimum of authorization in the domain gate and the authorization requested by the creating process.

Terminal Handling.

Programs:

dialup_:
This program creates processes for users using the TELNET protocol of the ARPA network to use Multics. It is included in this description of process initiation as an example of how the process initiation mechanism can be used.

dialup_$attach:
This entry causes dialup_ to use a network virtual terminal channel. The number of such channels in use at once determines the number of simultaneous TELNET connections that can be supported. When a new TELNET connection is made to Multics, one of the unused virtual terminal channels is selected to be used for that connection.

dialup_$dialup_:
This entry point is called whenever a significant event occurs for a terminal channel. dialup_ sends a greeting message to newly connected channels, and waits for a response. The response is parsed as a login line and the name of a gate to be used to create a process is determined from that line. Additional information in the login line is used to fill in create_info and pitmsg structures for a process. dm_$make_process is called to create a process, and if successful, control of the virtual terminal is granted to the new process before the new process is awakened.

dialup_$process_event:
This entry point is called when a message is received from the trouble report channel of a process created by dialup_. One of four possible actions is taken, depending on the contents of that message. The terminal channel can be hung up (if the process terminated voluntarily). Another process can be created for that terminal (if the message indicates that the previous process was damaged). A new greeting message can be printed and a new login line accepted. Or, an error message can be sent to the virtual terminal, if the trouble report message indicates some error, or is invalid.

Data Structures:

ntbl:

This structure is used internally by dialup_ to keep track of the virtual terminal channels currently in use. It has one entry for each such channel which contains the following information:

Terminal name (of the form netxxx),

Terminal state (dialup expected, login line expected, or hangup expected),

Process state (no process, process being created, process executing),

Event channel for terminal channel events,

Trouble report channel for process,

Error code for operations performed for this channel,

Index for this channel,

Person and Project for this channel,

Home directory (taken from login line),

Gate name.

References

[An74] Andrews, G. R., "COPS - A Protection Mechanism for Computer Systems," Computer Sci. Teaching Lab., Univ. of Wash., Technical Report 74-07-12, July 1974.

[Be73] Bell, D.E., and L.J. LaPadula, "Secure Computer Systems: A Mathematical Model," The MITRE Corporation, MTR-2547, Vol. II, November, 1973.

[Br75] Bratt, R. G., "Minimizing the Naming Facilities Requiring Protection in a Computer Utility," M.I.T. Project MAC Technical Report, TR-156, 1975.

[BH70] Brinch Hansen, P., "The Nucleus of a Multiprogramming System," Communications of the ACM 13, 4 (April 1970) pp. 238-241.

[Di68] Dijkstra, E. W., "The Structure of the 'THE' Multiprogramming System," Communications of the ACM 11, 5 (May 1968), pp. 341-346.

[Hu76] Huber, A. R, "A Multi-process Implementation of a Paging System," S.M. Thesis, M.I.T. Department of Electrical Engineering and Computer Science, June 1976.

[Ja74] Janson, P. A., "Removing the Dynamic Linker from the Security Kernel of a Computer Utility," M.I.T. Project MAC Technical Report, TR-132, 1974.

[Jo73] Jones, A. K., "Protection in Programmed Systems," Ph.D. Thesis, Carnegie-Mellon University, 1973.

[Ka76] Kanodia, R.K., and D.P. Reed, "Eventcounts: A New Model for Process Synchronization," (to be published).

[Ke76] Kent, S. T., "Encryption-Based Protection Protocols for Interactive User-Computer Communication over Physically Unsecured Channels," S.M. Thesis, M.I.T. Department of Electrical Engineering and Computer Science, June 1976.

[La69] Lampson, B. W., "Dynamic Protection Structures," AFIPS Conference Proceedings 35, (1969 Fall Joint Computer Conference,) pp. 27-38.

[La73] Lampson, B. W., "A Note on the Confinement Problem." Communications of the ACM 16, 10 (Oct 1973), 613-615.

[La74] Lampson, B. W., "Protection," Operating Systems Review 8, 1 (Jan. 1974) pp. 18-24.

[Or72] Organick, E. I., The Multics System: An Examination of Its Structure, M.I.T. Press, Cambridge, Mass, 1972.

[Re76] Reed, D. P., "Processor Multiplexing in a Layered Operating System," S.M. Thesis, M.I.T. Department of Electrical Engineering and Computer Science, June 1976.

[Ro74] Rotenberg, L. J., "Making Computers Keep Secrets," M.I.T. Project MAC Technical Report, TR-115, 1974.

[Sc72] Schroeder, M. D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," M.I.T. Project MAC Technical Report, TR-104, 1972.

[Sc75] Schroeder, M. D., "Engineering a Security Kernel for Multics," Proceedings, Fifth Symposium on Operating System Principles, November 1975, pp. 25-32.

[Sa75] Saltzer J. H. and M. D. Schroeder, "The Protection of Information in Computer Systems," Proceedings of the IEEE 63, 9 (September 1975) pp. 1278-1308.

[Wa73] Walker R. D. H. "The Structure of a Well Protected Computer." Ph.D. Thesis, University of Cambridge, 1973.

[We69] Weissman C. "Security Controls in the ADEPT-50 Time-sharing System." AFIPS Conferance Proceedings 35, (1969 Fall Joint Computer Conference,) pp. 119-133.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-163

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

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

Document Type:

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

Document Information

Number of pages: 124 (129 - IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

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

Check each if included with document:

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

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1 - 124) UNNUMBERED TITLE PAGE, 2 - 124
(125 - 129) SCANCONTROL, PRINTER'S NOTES, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 1/12/96 Date Returned: 1/18/96

Scanning Agent Signature: Michael N. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 01/12/1996
M.I.T. Libraries
Document Services