REMOVING THE DYNAMIC LINKER
FROM THE SECURITY KERNEL OF A COMPUTING UTILITY

By

Philippe Arnaud Janson
Ingénieur Civil Mécanicien-Electricien
Université Libre de Bruxelles
(1972)

This research was performed in the Computer Systems Research Division of Project MAC, an M.I.T. Interdepartmental Laboratory, and was sponsored in part by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006; in part by the Air Force Information Systems Technology Applications Office (ISTAO) and by ARPA under ARPA Order No. 2641 which was monitored by ISTAO; and in part by Honeywell Information Systems, Inc.

PROJECT MAC
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE MASSACHUSETTS 02139

REMOVING THE DYNAMIC LINKER FROM THE SECURITY KERNEL OF A COMPUTING UTILITY

by

Philippe Arnaud Janson

Submitted to the Department of Electrical Engineering on May 24, 1974 in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

In order to enforce the security of the information stored in a computing utility, it is necessary to certify that the protection mechanism is correctly implemented so that there exist no uncontrolled access path to the stored information. Certification requires that the security kernel be much smaller and simpler than the supervisor of present general purpose operating systems. This thesis explores one aspect of improving the certifiability of a computing utility by designing a dynamic linker that runs outside the security kernel domain.

The dynamic linker is designed to run in any user protection domain of a multidomain computing utility. It is shown that the dynamic linker never needs the privileges of the security kernel to properly operate. In particular, the thesis demonstrates the ability of the dynamic linker to link programs together across domain boundaries without violating the protection of either domain involved in the operation.

THESIS SUPERVISOR: Michael D. Schroeder
TITLE: Assistant Professor of Electrical Engineering

ACKNOWLEDGEMENTS

Master's theses are usually read by people in a relatively small circle. I do not expect this thesis to reach readers beyond Multics ring 7. However, I would like to express my gratitude to people in all rings (even beyond ring 7) who contributed to this research.

To start with ring 0, I would like to express special gratitude to my thesis supervisor, Professor Michael D. Schroeder. The correctness of his judgement, the effectiveness of his comments, the usefulness of his criticisms were second only to the amount of time he spent working with me.

I also thank Dr. David D. Clark for his interesting comments on selective parts of the design, and Mr. Rajendra Kanodia for the hours he spent helping me to test the final implementation of the design.

Going out to the user rings, I should like to thank Elaine Thomas, who coded the entire user ring dynamic linker in a single stroke, a real performance!

This paragraph would be incomplete without special thanks to Bernard Greenberg, whose invaluable-though somewhat noisy-help was appreciated, especially in the early phase of

the design [illegible] to the implementation. [illegible] my dear wife, [illegible] patience [illegible].

Research reported here was performed in the Computer Systems [illegible] partmental Laboratory. [illegible] Honeywell Information Systems, Inc., [illegible] Agency, the Air [illegible] Applications [illegible] of the [illegible].

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF FIGURES

I. Introduction

1. Security Kernel

The concept of computing utility designates a computer system or a network of computer systems dedicated to service a community of users (1). The type of the computers, of the services rendered and of the community of users may vary widely. Yet it remains that in all cases one of the most important features of the computing utility is to provide the users of the community with the ability to share the resources of the system. We will be specifically concerned about sharing the information stored in the computing utility. Different members of the community of users may have different intentions which are in conflict with one another with respect to the stored information. Some user might willfully or accidentally access (use, steal or modify) the information kept by another user in the computing utility. Hence uncontrolled sharing of all information poses a direct threat to the security of the information and to the privacy of the individuals concerned by the information (2-6).

In order to enforce the security of the information and to safeguard the privacy of the individuals concerned by the information, the access to the stored information must be controlled by some protection mechanism (7-11).

However, no protection mechanism will serve our purpose unless it is trusted by its users. Several features of a protection mechanism contribute to make it reliable (6,15). It is not our purpose here to discuss or even to list these features. Only one of them is of interest to us: the certification of correctness of the protection mechanism. Certification of correctness guarantees that the protection mechanism completely controls the access to the stored information, that it is an effective implementation of the desired protection scheme, and that there is no way a user program could subvert, circumvent or modify it to gain unauthorized access to the stored information. Certification of a protection mechanism is the result of a careful auditing of each component contributing to the protection of the stored information. Such auditing not only includes a verification of the intention and the implementation of each component of the protection mechanism but also a verification that interactions among them and with the outside world cannot cause malfunction or unexpected behavior resulting in unauthorized access to information.

The protection mechanisms are usually implemented by a combination of hardware and software. The programs and data bases of the software portion are a very sensitive part of the computing utility, for they control who can access what information. As a result, this protection software must be isolated from and protected against other programs in the computing utility. Any protection software component, if tampered with, could cause unauthorized access to stored information. Hence, user programs must be prevented from modifying, subverting or circumventing the protection software. Such enforcement should provide a complete control over the interactions between the protection software and other programs in a computing utility.

The security kernel of a computing utility is that part of the software which could, as a result of a bug or malicious alteration, cause unauthorized access to information. Thus it is the programs and data bases of the protection software plus any other programs (and data bases which control their behavior) that have direct access to the protection software.

In most systems the security kernel corresponds closely to the supervisor. It includes a great many programs and data bases that are not functionally part of the protection software. As a result, the security kernel is much larger and more complex than the subsystem which implements the protection mechanisms. This is unfortunate, because it is the entire security kernel which must be certified to establish confidence in the security of stored information. Extra size and complexity make certification more difficult.

This thesis will explore one aspect of making the security kernel of a computing utility smaller, simpler, and thus more certifiable by developing a system design in which the linking function is outside the security kernel. The linker of a computing utility is the program responsible for binding together separate procedure and data modules to build larger program elements. In current systems, the linker is almost always part of the security kernel, but as will be demonstrated in this thesis, is not part of the protection software. Removing the linker can significantly reduce the complexity and the size of the security kernel.

2. Dynamic Linker

In writing a complex program, it is extremely desirable to subdivide it into several modules. In doing so, the complexity of the programming task is reduced for the modules can be programmed and tested independently and existing modules may be incorporated into new programs. The idea of modularity implies the existence of some mechanism to assemble modules into larger programs. The writer of a module must be able to connect his module to others. One simple way to achieve the connection is to give a *symbolic name* to each module and to denote it by that name in other modules. This establishes a *symbolic* link between the two modules. The problem is that symbolic links are meaningless for the hardware of the processor. For a symbolic link between two modules to become a snapped link usable by the processor, the symbolic name used by the programmer must be translated into the logical (hardware interpretable) address of the module denoted by the symbolic name. When used to combine separately compiled modules translation is called linking. The program which takes care of the translation is called the linker.

There exists a wide variety of linkers which we will not describe here (12). Often a linker is invoked when a program is loaded into primary memory. Before control is given to that program, each symbolic name it uses is translated into a logical address by the linker. In other schemes, control is given to a program module as soon as it is in primary memory. When execution of the module hits a symbolic name, a hardware event (fault, interrupt, trap) triggers the linker execution to translate the symbolic name into a logical address. Execution resumes after the link is translated (snapped). This type of linking is called dynamic linking and is carried on by a dynamic linker. It is more flexible and saves the cost of loading into memory and linking together modules which may not be used by the program every time it is invoked. Although the rest of our thesis will be talking about dynamic linkers, the results of the research are also applicable to regular linkers. The problem is more challenging for dynamic linkers precisely because of the dynamic aspect introduced by the hardware events.

3. Background

Certification is a relatively recent topic in the field of computer science. Many authors have occasionally mentioned the need for certification, as we did here. But there exists no concensus on the best way to certify a large software system. The area is not very well structured and much work has still to be done to organize it. Yet most of the papers on that topic seem to agree that whatever hypothetical method is used to audit and certify the security kernel, the correctness of a "simple" kernel will be easier to verify than the correctness of a "complex" kernel. A small number of modules, strict constraints on the interactions between the modules, methodical design, systematic implementation, precise supporting documentation, simple language constructs, formatting and readability are factors likely to simplify the task of auditing the security kernel. Conversely, a large number of modules will undoubtedly complicate the problem. In addition, it is likely to increase the number of interactions to worry about. Complexity and sophistication of the modules themselves would also make auditing harder.

A good guideline when trying to simplify the security kernel is the principle of least privilege. This principle is the equivalent of the military "need-to-know" rule. It states that any program module should be granted just the privileges it needs to properly operate and no more. Modules of the security kernel should be granted the privileges of the security kernel on the basis that they contribute to the protection of the stored information. Modules not contributing to the protection goal should not be able to use such privileges. Keeping them inside the security kernel increases the size and complexity of the kernel and brings in functions and constructs that are hard to validate with respect to the protection goal of the kernel. Keeping them outside the kernel cuts down on the number of modules and interactions to be considered as part of the certification process. A module cannot abuse privileges it doesn't have to modify, circumvent, or subvert the security kernel operation.

4. Motivations

Designing a dynamic linker to run outside the security kernel environment of a computing utility is motivated by the desire to improve the certifiability of the protection mechanism in the system under concern. A linker is characterized by four features which suggest it should run outside the security kernel of the system to ease the auditing of the kernel.

Firstly, a linker does not implement any concept related to the protection of the system, or needed to support the protection mechanisms.

Secondly, in view of the function implemented by the linker, it seems reasonable to suspect that the linker does not need any of the privileges granted to typical modules of the security kernel. Therefore, the least privilege principle implies that the linker be outside the security kernel.

Thirdly, a linker is in general a very complex program. Even though its function is easy to describe, the details of its implementation require the use of intricate and sophisticated language constructs which make the reading and auditing of the program a quasi impossible task.

Finally, the linker, by its very nature handles data directly accessible to the users of the system. Such data could contain - purposely or not - inconsistencies capable of causing the linker to malfunction or perform unexpected operations. One suspects that it is much harder to verify the correct operation of a program when it can be presented with an arbitrary input than to verify correct operation when a "correct" input is guaranteed. Since malfunction and unexpected behavior are ruled out for program components of the security kernel, very sophisticated machinery would be required to verify the consistency of user requests to the linker and insure proper operation. Even if such machinery were available, it would only increase the complexity of the linker. Again we come to the conclusion that the linker should not be part of the security kernel. If so, no malfunction of the linker will ever subvert the protection mechanism of the system and cause unauthorized access to protected information.

To summarize our motivation we can say that designing the linker to run outside the security kernel environment of a system is a step towards simplifying, isolating and better defining the security kernel, thereby making its auditing easier.

5. Objectives

The motivation for our thesis is based on four arguments which suggest that the linker *should* run outside the security kernel environment of the system. The first objective of our thesis is to show that it *can* run outside the security kernel. We will have to show that the linker indeed does not contribute anyhow to the protection of the system and is never needed to support the operation of the kernel. We also will have to show the inverse relation; that is, the linker does not use or need any of the privileges of the security kernel modules. We eventually will have to show that the idea of forcing the linker to execute outside the security kernel environment does not introduce any unsuspected, unsolvable problems.

Clearly we would not pay so much attention to our problem if its solution were obvious and if all linkers known today were running outside the security kernel environment of the system for which they were designed. There exist a few systems (13) where the problem has been solved. However, it was solved only for the very simple case of a static linker binding modules together inside one protection environment. Instead our thesis will propose a general solution of the problem for a dynamic linker binding modules together across protection environment boundaries. The design to be proposed can be applied to any type of computing utility with some variations which we will eventually mention when appropriate.

Except for a few cases already mentioned, all systems are designed with their linker being a component of the security kernel, and having the privileges of the security kernel (14). The second objective of the thesis is to show the feasibility of the design to be proposed for a particular real world system. We have chosen to remove the linker of the Multics (Multiplexed Information & Computing Service) (15-18) system from the security kernel environment and to force its execution into the user environment.

The linker presently runs in the environment of the security kernel of Multics as do many other components of the system which do not belong in the security kernel either. The main reason for this design was that the cost of dynamically changing the protection environment of a computation was prohibitive in the initial version of Multics. Hence, it was decided to include many system components in the security kernel that were not part of the protection mechanisms in order to minimize the number of times the protection environment was changed in the course of a computation. Snapping a single link requires two environment changes with the linker inside the security kernel, but may require 10 to 100 with the linker outside. A second version of the Multics hardware (15) has reduced the cost of a change in protection environment to the level of a normal interprocedure call. As a result, there is no longer an economic incentive to leave the linker in the security kernel.

Before we go on to develop the design we will mention a third objective of the thesis. In removing the dynamic linker from the security kernel of Multics, we hope to establish a few more criteria for deciding whether or not a program belongs in the security kernel of a system. We also hope to better define what general programming features contribute or hinder the task of removing a program from the security kernel. These lists of criteria and features of interest will certainly be as helpful as the removal of the linker itself to better define the security kernel of a computing utility in general and of Multics in particular.

6. Plan of the Thesis

Before we come to the body of the thesis we would like to briefly describe how we will develop the research and carry it on to the detailed implementation of a linker running outside the security kernel of a computing utility.

Chapter II will develop a computing utility model where emphasis will be put on features directly relevant to our research. The model will serve as a basis to describe the design and it will help the reader to apply the design to different systems by matching the model with that system.

Chapter III will propose a complete design of relevant parts of the computing utility. Problems encountered in the design will be discussed and solutions will be proposed.

In Chapter IV we will demonstrate the feasibility of the proposed design by describing its implementation on Multics.

II. A Computing Utility Model

In order to better define the features of the design we will propose, and to generalize its applicability to any computing utility, we will describe a computing utility model. This will enable us to explain the proposed design in terms of the model. It will enable the reader to apply the design to any specific computing utility by matching that computing utility with the model.

We will develop the model in two steps. Firstly, we will describe a protection model suited to the environment of a computing utility. Secondly, we will build on top of this model an information storage model suited to the needs of a dynamic linker. The model will help us to better define the concepts of protection environment and logical address space which we have occasionally mentioned but have not carefully defined yet. We then will explain in detail the operation of the linker in terms of the model. This will greatly simplify the subsequent description of the design of a linker running outside the security kernel of a computing utility.

1. Information Protection Model

In order to better understand and study the problems related to protection of stored information, several structural and mathematical models of protection schemes have been proposed (19,20). We will briefly describe here a model based on the concept of protection domain (21,22). This model will help us understand what is meant by a protection environment and particularly what the security kernel environment is.

For the purpose of our discussion, we will talk about the environment of the computing utility in terms of objects and subjects. Objects are passive. They are the information containers of the computing utility. They must be protected to prevent unauthorized access to stored information. Objects are the procedures and data bases stored in the computing utility. Subjects are active. Subjects are the internal representation of users of the computing utility. Subjects, sometimes called processes or jobs, act on behalf of users to create, delete, modify, use and manipulate objects.

Subjects can access objects by means of capabilities. A capability is an identifier denoting some object in the computing utility. Any subject possessing a capability for an object is entitled to access that object.

The set of capabilities available to a given subject defines the domain of execution of the subject. The domain of execution of the subject is the protection environment where the subject operates.

When a subject changes domain of execution, it changes its set of capabilities. He can enter a new domain of execution only through a gate. A gate is a procedure object which forces entrance to a domain to coincide with invocation of certain procedure objects in the domain. These procedures completely determine the activity of the subject in the domain. For a given subject, a gate is an entry point into a given domain. However, for two different subjects, the same gate object leads into distinct domains. We make the assumption that each domain can be entered by only one subject. Thus when two subjects wish to enter the "same" domain, they are actually installed into distinct domains containing equivalent sets of capabilities.

With this model in mind we can better talk about the environment of the security kernel. For each user computation, i.e. for each subject of the computing utility, there exists one domain-the security kernel domain (23,25)-where capabilities exist for the subject to access procedure and data objects of the security kernel. Access to the data objects is constrained by the access pattern encoded in the procedures of the kernel. Access to the procedures is further restricted to certain entry points: the gates into the security kernel domain. Hence complete control is gained on the interactions between the kernel and the outside world. The security kernel is a so called protected subsystem, (24,25) an instance of which exists in the first domain created for each subject in the computing utility.

2. Information Storage Model

The previous paragraphs have made more precise the notion of protection environment. We will now consider the concept of logical address space.

The set of all objects in a computing utility constitutes the file system of the computing utility. Among these objects is a particular set of objects called catalogs. Catalogs are data bases containing descriptive information about some set of objects. One of the items contained in a catalog about each object described in that catalog is the physical address of each object. The physical address of an object defines where the object is located on some memory device attached to the computing utility. The physical address of an object must be clearly distinguished from its logical address. The logical address of an object is the address by which an existing subject references the object. Only logical addresses are meaningful to processors executing machine code. An object always has a physical address even when it resides on secondary storage and no subject uses it. But it may not have any logical address if no subject uses it. Assigning a logical address to an object on behalf of a subject is the role of the file system manager (FSM).

When a subject wants to assign a logical address to an object, it must pass to the FSM the unique identifier of the object. The unique identifier of an object can be a unique name, a unique number, or a catalog unique identifier and the symbolic name of an object in that catalog. Unique identifiers are different from symbolic names in that more than one object may have the same symbolic name as long as they are described in different catalogs, but no two objects can have the same unique identifiers. When given a unique identifier, the FSM performs two distinct functions. Firstly, it searches the file system to find the description of the object denoted by the unique identifier. If the search fails or if the FSM decides that the requesting subject does not have the right to know about the object under concern, an error message is returned and no action is taken. If the search succeeds and the requesting subject has the right to know about the object, the FSM maps the object into a logical address of the address space currently seen by the subject (enables a logical address), remembers the binding between the unique identifier and the logical address, and returns the logical address to the subject.

One question is now in order. What is the real nature of a logical address? Since the FSM, a component of the security kernel, releases logical addresses on the basis of a protection decision, a logical address is merely a capability to access an object. As long as a subject has no enabled logical address for an object, it cannot reference that object. If and when a logical address is enabled and delivered to the subject by the FSM, it gains access to the corresponding object, i.e. it has a capability for that object. This establishes the connection between our information protection model and our information storage model.

This connection between the two models brings up the question of the nature of the logical address space. Since a capability for an object is granted to a given subject in a given domain, one might wonder whether the logical address allocated to the object is valid only for that subject in that domain. In other words, once a logical address is assigned to an object for some subject in some domain, will that subject see the same object at the same address in other domains? Will all subjects see the same object at the same address in all domains? The answer to these questions depends very much on the type of logical address space supported by the system under concern. In the simplest case, where the logical address of an object is its primary memory address, if any, then we can talk of a system wide address space. Once an address of the space is allocated to an object, all subjects in all domains will see that object at that address if they have access to it. On a virtual memory system, each user, i.e. each subject may have one address space of its own. When an address is allocated to an object in a subject address space the subject will see the object at that address in all domains where he can access the object and the address will be meaningless (not usable) in other domains. But all other subjects may or may not use the corresponding address of their own address space for the same object. Finally in some systems, there may be one address space in each domain. Such is the case, for instance, of base and bound machines. A domain is defined by the base and the bound of its address space. A logical address is mapped into a physical address by relocating it relatively to the base and within the bound of the address space of that domain. Once an object is mapped into one address space, the address space of another domain may or may not contain the same object at the same logical address depen-depending on what its base and bound are. To conclude this discussion, we will assume for the rest of this thesis, that the concept of address space, when unqualified, means the address space seen by the given subject in the given domain. Unless specifically stated, no assumption will be made about who can see the same address space in what domain.

3. A Dynamic Linking Model

The last paragraph described the models we will use to support our design. Before we move on to the design itself we will describe the detailed operation of linker with respect to the models. In doing so, we will not have to worry about what a unique identifier, a logical address, a domain, or a gate is. We know that all these concepts can be identified in any computing utility and that our description can be based on them without ambiguity.

Whenever a subject executing an object encounters a symbolic name of, or a symbolic link to another object, a hardware event called a link fault occurs. As a result of the link fault a copy of all machine registers, called the machine status, is handed to the linker.

The first task of the linker is to analyze the machine status to determine which symbolic link caused the fault and which object was being executed at the time of the fault. This object is called the faulting object. The domain where it was executed is called the faulting domain.

By searching the faulting object, the linker will find a complete description of the symbolic link and in particular the symbolic name associated to the link which designates some object of the environment. This object is called the target object of the link. The domain in which it belongs is called the target domain.

The second task of the linker is to search for the target object in the file system and to map it into the logical address space. In order to do this the linker will of course need to invoke the FSM. The search is driven by so called search rules. Each domain has associated with it a different set of search rules. Search rules are an ordered set of catalog unique identifiers. Of course, it is irrelevant to talk about search rules when the file system is one single catalog. However, in general, it contains many catalogs. The search rules force the linker to search only some of these catalogs in the desired order. The linker takes one search rule at a time, combines it with the symbolic name of the target object thereby making an object unique identifier. The linker hands the unique identifier to the FSM to search the file system. If the search fails, the FSM returns an error code to the linker. The linker will keep trying the next search rule, if any, until a search succeeds.

In this case the FSM returns the logical address of the target object to the linker.

The third task of the linker is then to translate the symbolic link into a *snapped link* usable by the processor. This is called *snapping the link*. The linker just replaces the symbolic name in the link by the logical address of the target object.

Finally the linker must modify the machine status to force the executing subject to reuse the now snapped link.

By a mechanism external to the linker itself, the machine status is then restored so that the executing subject jumps back to where it was just before the link fault.

Once a symbolic link is replaced by a logical link, it will no more cause any link fault for the current subject in the current domain.

III. Design

1. General

The last chapter presented a computing utility model which will be used to support the discussion of the design. The steps in the operation of a dynamic linker have been described. As it should now be clear to the reader that programming the linker itself is a feasible task, the current chapter will rather concentrate on the problems of inserting such a linker into the overall design of a computing utility such that it be outside the security kernel. The next chapter will then present a test case implementation of the design to demonstrate the use of the model in identifying the components of a real system and to show the feasibility of implementing the design on a real system.

In developing the discussion of the design we will try as much as possible to progress naturally and to handle each problem as it shows up. In a first section we will explain how the security kernel can operate without the help of the dynamic linker. In the remaining sections we will demonstrate that the dynamic linker can operate without the privileges of the security kernel. This order of discussion coincides with the order of events when a computing utility is brought up into operation: the security kernel by its fundamental purpose is the first subsystem to be operational and is used to bring up the rest of the system functions, the dynamic linker among others.

We do not claim in any way that the design to be outlined is the only possible design solving our problem. By its very nature, the topic of the research poses several structural problems which are easy to identify and to describe. However, designing solutions to these structural problems cannot be done systematically as would be the case for mathematical problems. Solutions to a particular structural problem may bring up other structural problems. It is hard to predict and to control the propagation of the effects of a particular solution to a particular problem. Hence it is hard to estimate a priori which solution minimizes the number and the magnitude of hidden potential problems. As it is impossible to discuss all solutions in detail, we will attempt to justify our choice between different solutions whenever possible, and especially where a sophisticated solution has been prefered to an apparently more obvious one. Even so, we do not claim that all possibilities will be discussed. We are convinced that equivalent designs could be proposed. We believe only that our design is among the simplest ones.

Finally, we will attempt as much as possible to be sufficiently precise in the discussion of the design to convince the reader that subsequent implementation is practical and straightforward. At the same time, we will try to remain sufficiently abstract to enable the reader to implement the design on any general purpose computing utility.

2. Security Kernel Initialization

Before any user can request service from a computing utility, the system must be brought up into operation. This initialization task is done under the responsibility of a subject called the initializer. The initializer must cause the loading and set up of all programs required to support the operation of the system. The first of all subsystems which needs to be initialized is the security kernel because of its fundamental function: generating other subjects and domains for these subjects would be impossible without an operational security kernel. We are concerned about one aspect of making the kernel operational. Like all subsystems in a computing utility, the security kernel is a modular program. Hence its operation does require a linking function to combine the modules together. However, our objective is to propose a design where no dynamic linker exists in the security kernel domain. The security kernel is not allowed to cause link faults. Hence all links of the security kernel must be snapped prior to the operation of the kernel. This task is part of the security kernel initialization.

Linking together all modules of the security kernel requires the help of a static linker. Essentially two types of static linker could be used: a binder or a prelinker. The binder is a static linker which prepares once and for all a fully operational security kernel that can be used without any further initialization as many times as desired. The prelinker is a static linker which links the modules of the security kernel together each time the system is stacked, during an initialization phase. We will not describe the detailed design of either a binder or a prelinker. This topic is below the level of our discussion. We will ask the reader to realize that writing a static linker is feasible in many ways. We will just discuss the properties of each type of static linker.

The technique of the binder seems both simple and economical. It is economical because the links of the security kernel are snapped only once for a given system version and the resulting operational security kernel can be reused as many times as desired. It is simple because auditing and certification of the kernel must be done only once on the final operational kernel. The binder is kept outside the environment to be certified; only the results of its operation are to be audited.

The technique of the prelinker instead requires that the prelinker be audited and certified. Since domains are meaningless until the security kernel is initialized to support them, the virgin environment seen by the initializer may be viewed as just one single domain bound to become the domain of the security kernel. Consequently the prelinker of the security kernel which is executed prior to any module of the kernel is in some sense a component of the soon-to-be kernel. The prelinker must therefore be certified. By now the reader may wonder what is gained by the prelinker technique. We want to remove the dynamic linker from the security kernel but we propose to keep a prelinker in the kernel.

Firstly, the use of a prelinker may make the system initialization more flexible. The use of a binder frequently implies that not only the version of the system but also the initial configuration of the system (hardware configuration and sizes of various supervisor tables) always be what the binder assumed. Instead, in the case of the prelinker, even though the version of the system used may always be the same, the configuration of the system may be changed each time the system is started by properly notifying the prelinker of relevant configuration data to be respected. Thus a prelinker is more flexible than a binder.

Secondly, believing that the certification of the prelinker is just as bad as the certification of the dynamic linker is wrong. By its dynamic aspect, by the requirement that it be able to deal with objects scattered in a large file system, and by the fact that it may support miscellaneous sophisticated linking features needed by user programs (see next chapter), the dynamic linker is a much more elaborate program than the prelinker. The prelinker is a static linker; it deals only with objects of the supervisor concentrated in just a few well known catalogs of the file system; and it may not support sophisticated linking features because security kernel modules, unlike user modules, may be programmed to avoid such features. In addition, by its very nature, the prelinker is an atomic program while the dynamic linker is a modular program. All such factors make a prelinker a lot simpler and hence easier to certify than a dynamic linker. Finally since the prelinker is needed only during initialization the security kernel can discard its own capability to ever again access it during regular system operation. Thus the prelinker cannot be executed again once the system is initialized, and therefore it cannot hurt the system. This also simplifies the problem greatly.

Consequently, the choice between binder and prelinker is a choice between relative certifiability and flexibility. In general this choice is independent of where the future dynamic linker will be running. Since the implementation to be described in the next chapter is based on the prelinker idea, we will assume the same idea in this chapter. However, we acknowledge the fact that using a binder is most probably equivalent as far as our thesis is concerned. We will now temporarily abandon the operational security kernel we have obtained. The next section will first discuss a few design principles and then carry on the development of the system by building other domains around the security kernel.

3. Dynamic Linker Initialization

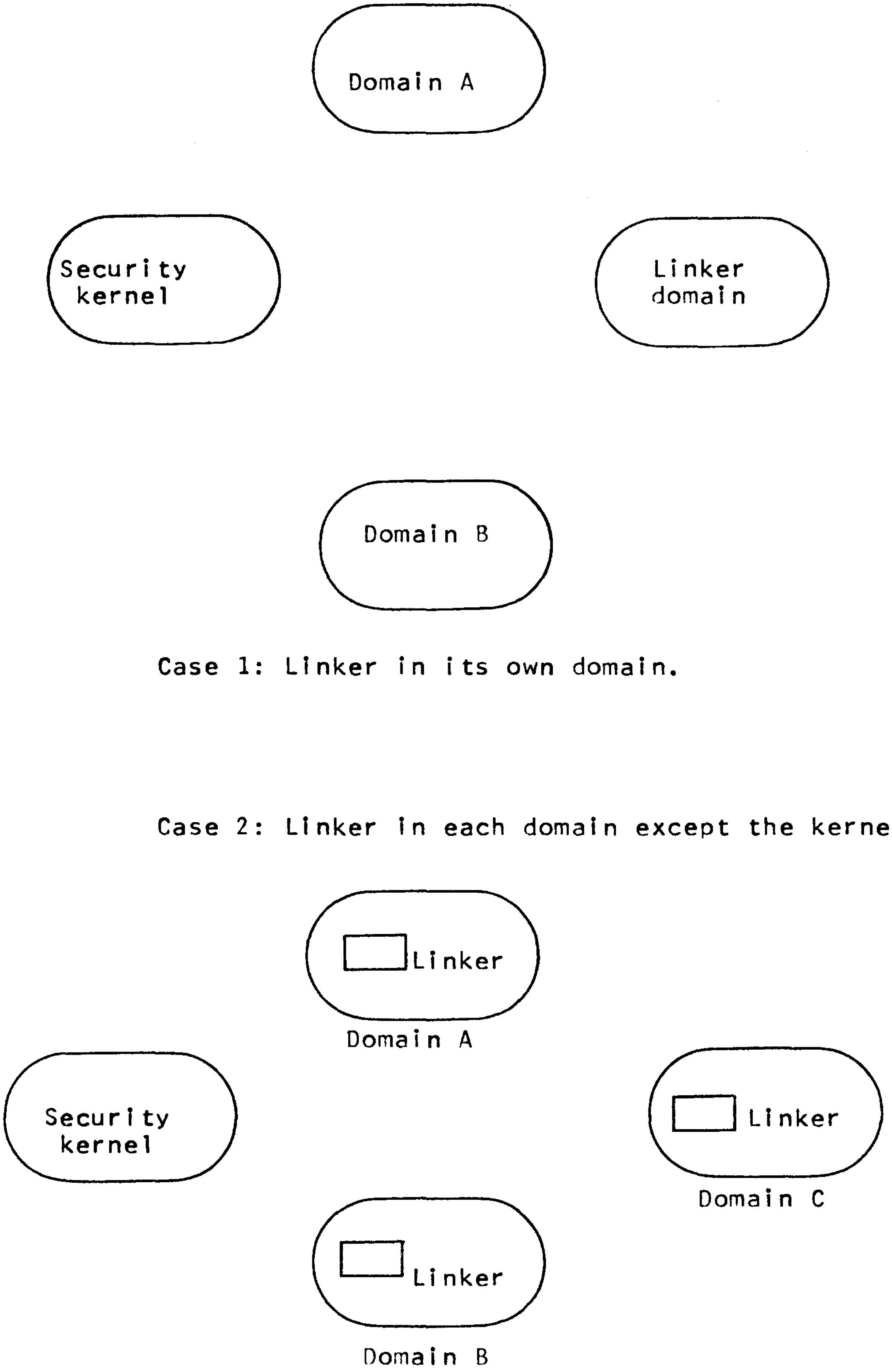
a. Design Principles

In the previous section, we have shown how the security kernel modules can be linked together without the help of the dynamic linker. Once linked, they no longer need any linker, thus they can operate without one. The rest of this chapter will examine the other side of the design. It will be demonstrated step by step that the dynamic linker can operate outside the security kernel.

It seems that the first problem we encounter is to define what "outside" means. One half of our design is to remove the linker from the domain of the security kernel. The second half of it is to decide in which other domain or domains the linker will run.

It seems very appealing to simply install the linker once and for all in a domain of its own (see figure 1) where a subject will be able to go if and when necessary. Even though this solution may seem clean and obvious, it is very likely to raise implementation problems. Indeed, on each link fault, the linker domain would have to be provided dynamically with appropriate capabilities to access the faulting object, and perhaps the target object or even other objects in the faulting or the target domain. When the dynamic linker was always running in the same domain and that domain was the security kernel domain, providing it with dynamic capabilities was easy given the unique privileges available in the security kernel. However, this is no more true if the linker runs in a domain different from the security kernel domain. Furthermore, a linker domain containing capabilities for objects in several domains, even if only one at a time, can potentially operate as an unauthorized information channel between these domains if it malfunctions. Therefore, such a linker must be certified to prevent potential unauthorized

Figure 1: Different environments for the linker.



Case 1: Linker in its own domain.

Case 2: Linker in each domain except the kernel.

access to the information.

A second potential answer can be found by thinking in terms of capabilities. Since the linker will need to access objects in the faulting domain and perhaps in the target domain, both domains seem potential candidates to host the linker. The target domain is actually not a good candidate because it is not determined until the target object is identified. Hence it is undetermined at the time of the fault and the only domain where the linker could initially run is the faulting domain which is easily determined by the machine status.

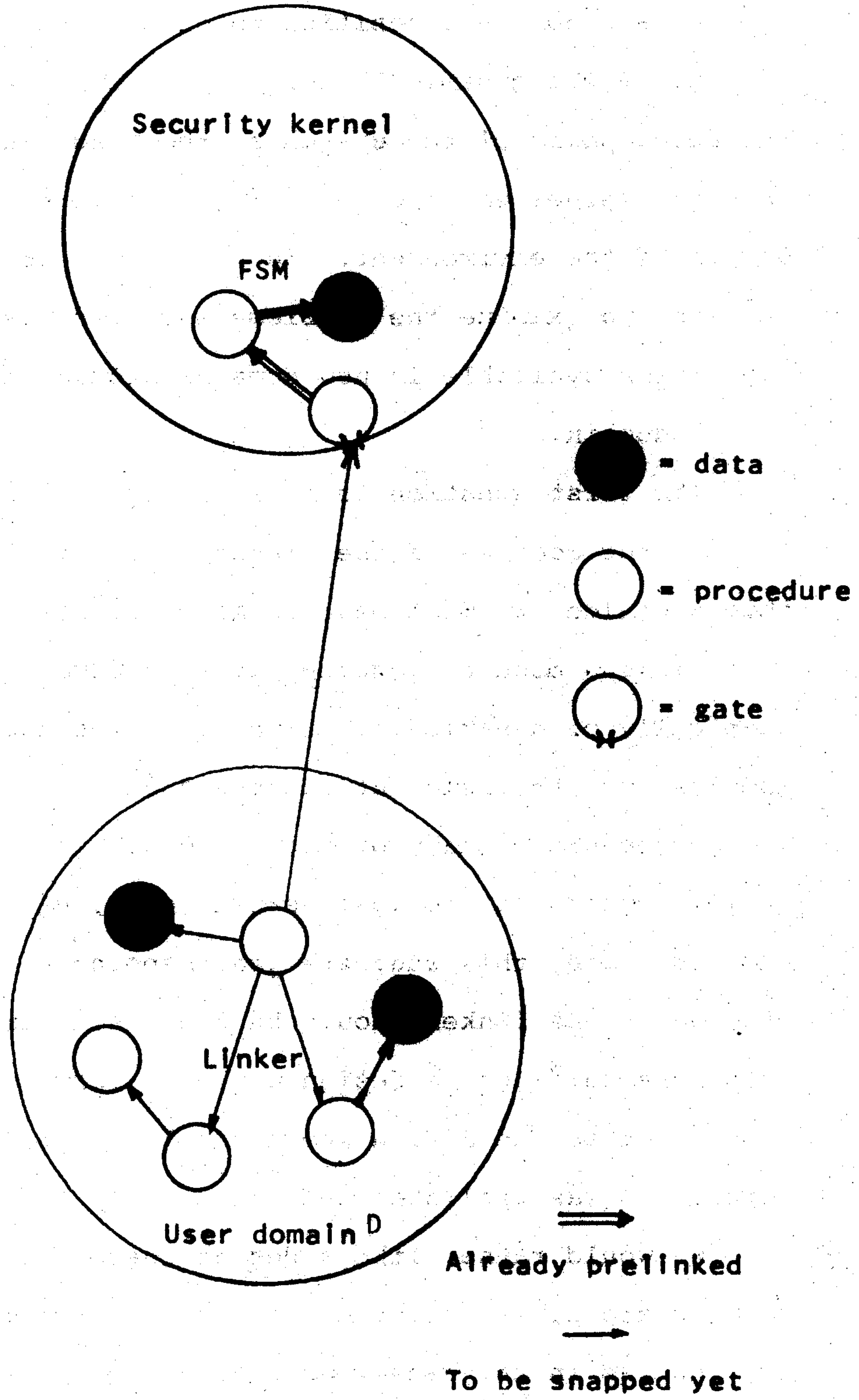
Consequently, even though we do not definitely reject the first solution, we strongly recommend and will further assume the second solution which at least guarantees easy access to the faulting domains and eliminates a security threat. It will be seen that access to the target domain is usually not required and eventually easy to provide. In the above discussion we have identified the major problem of removing the linker from the security kernel domain: it no more has all the privileges to access any object in any domain; each particular invocation of the linker will see access capabilities constrained to those of the faulting domain for the invocation (see figure 1).

We have just decided to design the linker to run in "the" faulting domain. Since any domain is a potential faulting domain except for the security kernel domain, the linker must be made "available" in all domains except the security kernel domain. The second problem which we will now discuss is the notion of availability of the linker in a domain. What does availability of the linker mean?

Firstly, it means that capabilities must exist in all domains, except the security kernel domain, to execute the linker. Providing such capabilities in each domain is rather trivial and should pose no implementation problems.

Secondly, a dynamic linker, like most programs of a computing utility is a modular program. As such proper operation will be possible only if there exists a means to snap links between the various modules involved in dynamic linking. For most programs in a computing utility links can be snapped dynamically. In the case of the dynamic linker, this proposition is nonsense: if the dynamic linker contains unsnapped links, it is not operational and cannot count on itself to snap its own links. Hence a static linker must be used to link the dynamic linker modules prior to using them. As long as the linker was part of the security kernel, its modules were linked together by the prelinker of the security kernel. Now we have removed the linker from the kernel, it will no more be automatically prelinked. Hence, its modules must somehow be linked together independently to make it operational in other domains. We may ask ourselves what sort of links exist in the dynamic linker and have to be snapped statically. The linker is a set of procedures and data modules which according to our objective can be executed in any domain except the security kernel domain. Clearly at least all links between these modules must be snapped to ensure proper operation. In addition, the earlier description of the linker operation mentioned the need to invoke the FSM. Since the linker is anywhere but in the security kernel, it can invoke the FSM only through one or more gates into the security kernel. Hence there will exist links to these gates. They must also be snapped. Consequently, the situation can be pictured by figure 2. Each domain has capabilities, like domain D, to execute "the" linker. "The" linker is the set of all procedures and data bases potentially invoked in dynamically linking two modules. The linker also contains one or more links to security kernel gates. Notice that these gates, as kernel components, are guaranteed to be further prelinked to internal modules of the kernel during system initialization. Hence we do not need to worry about them anymore even though they contain links to be involved in dynamic linking.

Figure 2: Linker and security kernel initialization: configuration of the links to be snapped.



b. Prelinking the linker

We are now in a position to discuss how static linking of the dynamic linker can be done. We had left the development of the system at the stage where the security kernel was operational in the first and only domain of the environment. We will now pursue that discussion and examine the problems involved with making the linker available in new domains around the security kernel domain.

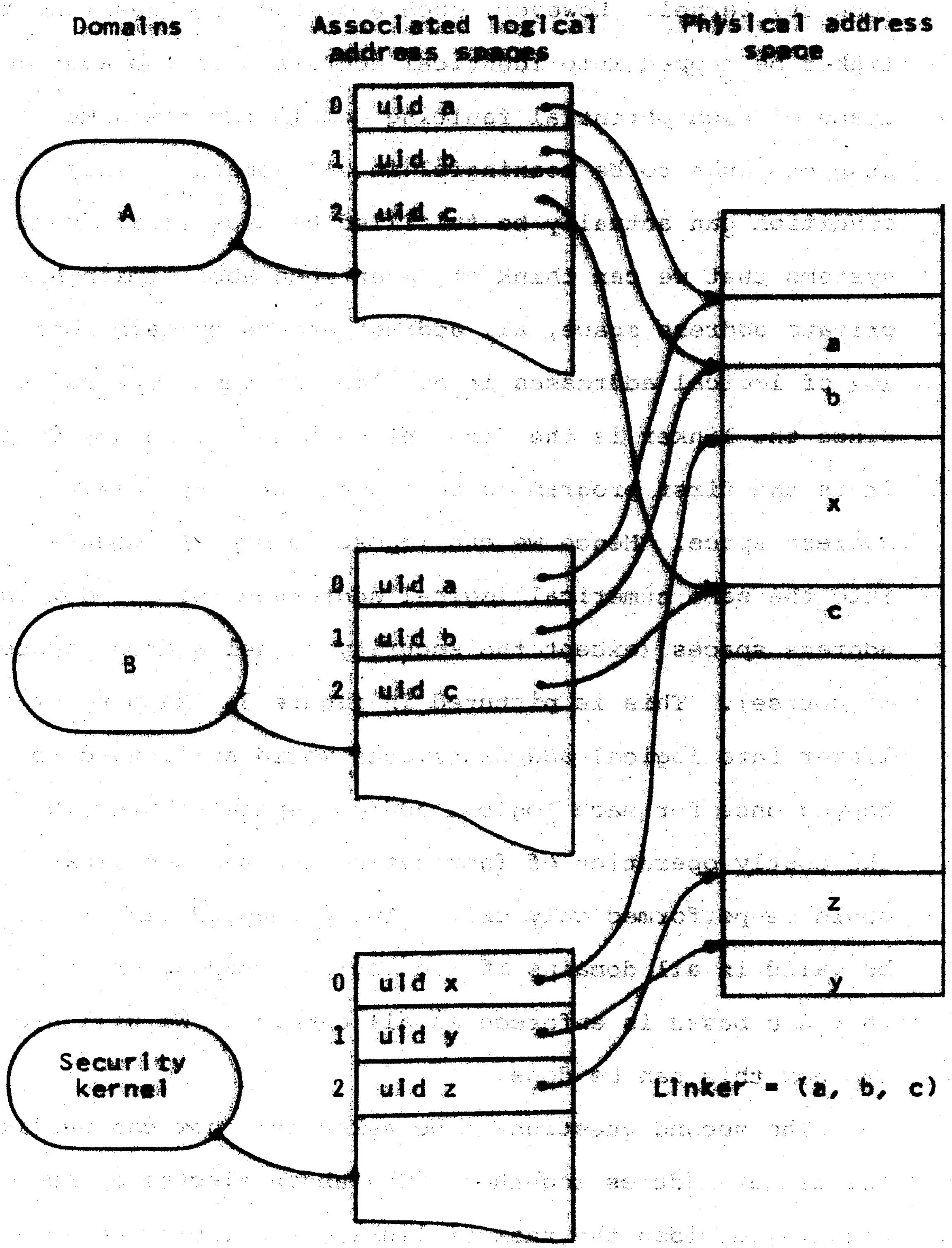
The first question to be asked is: when do we want to link the modules of the linker together? To answer this question, we must bear in mind the important fact that linking modules together in some domain, whether statically or dynamically, first requires mapping the modules into the relevant address space.

Since each domain or future domain in the computing utility could, in the most general case, have its own address space, this suggests that mapping and consequent linking of the linker should be done each time a domain is generated. Such a design would be very expensive in comparison to the design where the linker was in the security kernel and was prelinked only once.

We would rather like a design where the linker modules are linked together only once for the whole system just as in the case where the linker was in the security kernel. However, such a design requires that the linker be mapped into identical addresses in the address space of each potential faulting domain for the same snapped links to be meaningful in all domains. This condition can actually be fulfilled because in all real systems that we can think of, even when each domain has a private address space, all address spaces contain some set of logical addresses in overlapping numerical ranges. Since the linker is the first program needed in any domain, it is the first program to be mapped into any domain address space. Hence we can impose to map its modules into the same numerical logical addresses for all domain address spaces (except the security kernel address space of course). This is pictured in figure 3. Mapping of the linker into logical address spaces would still have to happen once for each logical address space created, but the costly operation of fabricating the snapped links could be performed only once. These snapped links will be valid in all domains if the logical mapping on which they are based is enforced in all domains. We will now see how this can be done.

The second question to be asked is: how can we link the linker modules together? The above discussion has actually divided the task of linking the linker modules

Figure 3: Domains and their address space.



into two. We first must fabricate all necessary snapped links on the basis of some fictive mapping (to be decided upon). We then must enforce that mapping in each domain address space we create and we must communicate the snapped links based on that mapping to each new domain. We will now examine these two steps in detail.

Fabricating the snapped links is, as we already mentioned, the task of a static linker. Since the snapped links must be fabricated before any domain is created around the kernel domain, the static linker must do its job before or during system initialization. "Before" corresponds to the idea of a binder. "During" corresponds to that of a prelinker. The choice between the two is the same as in the case of the security kernel initialization. As we have assumed the idea of the prelinker for the security kernel, it is all but natural to keep the same idea for the linker. The flavor of the design is of course to use the security kernel prelinker a second time (with some variations perhaps) to prelink the dynamic linker. This saves the trouble of writing and certifying another prelinker. Once the security kernel is prelinked, and just before capabilities to use the prelinker are discarded, the initializer invokes the prelinker again to prelink the future dynamic linker. The following paragraphs will discuss step by step the operation of the prelinker on the linker because some aspects of that operation have hidden implications that the prelinking of the kernel did not have.

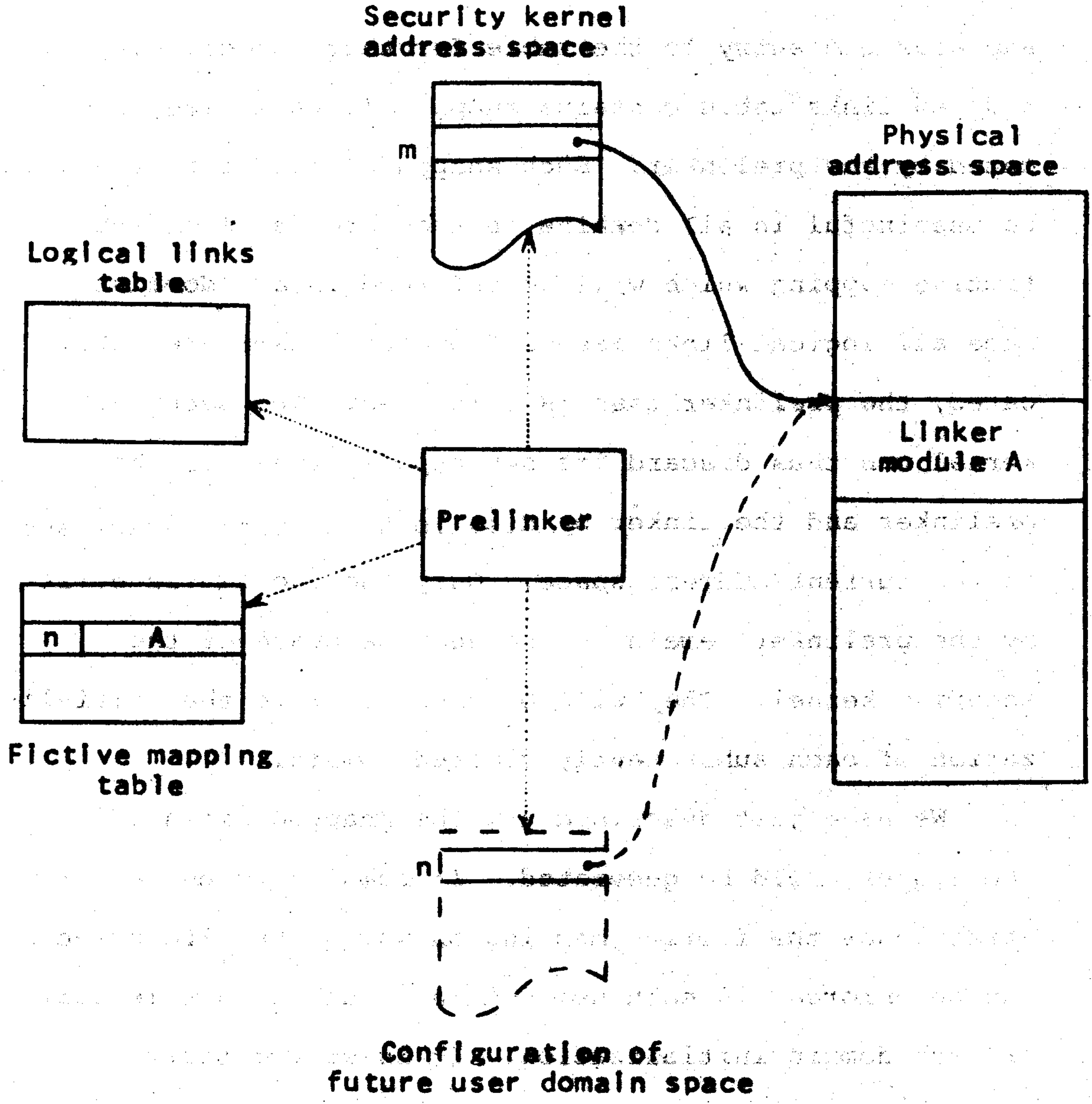
In prelinking a link between two modules, the first task is the retrieval of the symbolic name corresponding to the target of the link. This symbolic name is stored somewhere in the origine object of the link. Since we want to prelink all links issued from the linker, all modules of the linker must be mapped into the security kernel address space during system initialization. The prelinker will then have the ability to discover all symbolic links it must translate by a methodical scanning of all modules of the linker accessible in the address space.

The second task to be accomplished in prelinking a link is to search in the "file system" for the target object corresponding to the symbolic link being translated. The nature of the "file system" in the elementary environment of system initialization is however questionable. Any computing utility includes some FSM to support a file system during normal operation. But it is not obvious that in all computing utilities, the file system and the FSM are initialized and available at the time the prelinker is run. If they are, searching of a target object can be achieved by the FSM. If they are not, the target object must be initially brought into the address space of the security kernel from whatever memory device is used to load and start the system, otherwise it could not be accessed and identified by the prelinker. In the latter case, searching is reduced to a simple scanning of all objects in the address space and will succeed when the right symbolic name is found. This of course implies that any potential target object, i.e. the linker and any security kernel gate it calls, be in the address space of the kernel.

Finally we have to worry about mapping. Once the target object of a link has been identified, a logical address must be obtained for it to build the link to it. The problem may seem trivial here since everything referenced by the linker and the linker itself is mapped in the current address space to start with. However, we must remember that whatever mapping we base the snapped links on will have to be enforced in all future domains. It may not be feasible or reasonable to map the linker and security kernel gates it calls into all address spaces at the addresses where they currently are in the kernel. In particular, we have mentioned that logical addresses in a domain are a form of capabilities for that domain. We have also mentioned that after initialization, the security kernel will want to discard its own capabilities to ever again access the prelinker. This means it has to unmap the prelinker from the address space it currently sees. Along the same lines of thought, the linker is mapped in the initial kernel address space for the purpose of prelinking. But the linker is not part of the security kernel. Hence the initializer will also unmap it after prelinking is completed. Consequently, we are facing the following problem. All objects we are interested in are currently mapped into the only valid address space, but this mapping is temporary and the future mapping to be used in all domains other than the security kernel domain may be entirely different as represented by figure 3. This future mapping is of course the fictive mapping we discussed earlier. Determining the fictive mapping is thus done by the prelinker by assigning the target object of each link it translates a logical address suitable for all future domains.

Let us now conclude the above discussion by describing the mapping function of the prelinker. Figure 4 illustrates this function. The prelinker uses and progressively builds up two tables. The fictive mapping table contains a set of entries of the form (logical address - unique identifier). Each such entry defines the future logical address of the uniquely identified

Figure 4: Prelinking the linker.



object. Each time the prelinker snaps a link to a target object in the linker not already assigned a fictive logical address, it generates a suitable fictive address and adds one entry to the table for that object. The snapped links table contains snapped links already fabricated by the prelinker. Such snapped links of course will be meaningful in all domains as they are based on the fictive mapping which will be enforced in all domains. Once all logical links issued from the linker are fabricated, the prelinker task is completed. The security kernel can thus discard its own capabilities for the prelinker and the linker by disallocating their addresses in the current address space. Only the two tables built by the prelinker remain in the address space of the security kernel. They will be used to drive the initialization of each subsequently created domain.

We have just described how the snapped links of the linker could be generated. It remains to be demonstrated how the fictive mapping on which they are based can be enforced in each new domain. Such a task is part of each domain initialization. It is straightforward. Each time the security kernel creates a new domain, it uses the fictive mapping table to drive the FSM and have it enforce the mapping in the new domain. Each entry of the table is interpreted as a request from the new domain to search the file system for the object uniquely identified by the entry and to map it into the specified fictive logical address. After having done so for all entries, the fictive logical addresses are actual valid logical addresses for the new domain. Then the security kernel maps a copy of the snapped links table into the new domain address space. This will finally enable the linker to properly operate in the new domain by using the snapped links based on the now real mapping for that domain.

What we have achieved is providing each domain with an operational linker, i.e. a prelinked linker. The first section of this chapter described how the security kernel could be initialized without the help of the dynamic linker. The current section has described how the dynamic linker could in turn be initialized in much the same way. A fictive mapping of the linker and some security kernel gates had to be generated during system initialization and must be enforced by the FSM independently for each domain created during system operation. Each such domain then sees the linker and relevant security kernel gates in its logical address space. In addition, each domain has a copy of the snapped links required by the linker to operate. Link faults can now safely occur in such domains. This will be the topic in the next section.

4. Link fault handling

So far we have shown how to initialize a security kernel without the help of a dynamic linker. We then have shown how the security kernel can in turn initialize a linker in each domain it creates. It remains to be demonstrated how the operational linker we now have in each domain can handle link faults without the privileges it would have if it were in the security kernel domain. As long as it was part of the security kernel, the linker had all the capabilities it wanted to access faulting domain objects, target domain objects, and any object in general. We now will show that the constrained privileges available to the linker in the faulting domain are still sufficient to guarantee proper operation.

The first problem we will now discuss is that of invoking the linker in the faulting domain. Suppose that an object being executed in some domain causes a link fault by attempting to reference another object through a untranslated symbolic link. This link fault is an event recognized by the hardware of the system. As a result of the event, control must be given to the linker.

Some faults-access violations for instance-are very sensitive events and must be handled by the security kernel. Since the processor recognizes hardware events themselves, but may not know about their nature or their sensitivity, it is frequently necessary that all faults be sorted by the security kernel before being passed to any other domain for handling. Consequently, on a link fault, the first program to be invoked is the security kernel. It in turn has to invoke the linker in the faulting domain. Such action may seem straightforward. The security kernel could just call a gate into the faulting domain and that gate could in turn call the linker. However, if we want to be absolutely general, our design must fit systems which support a very large number of domains. In that case, since any domain is a potential faulting domain, the security kernel needs to know about a gate into each domain. But since domains and gates can be created and destroyed at will during system operation, it is impossible to prelink the kernel to a gate into each domain at system initialization time. Hence we must find some means to enter the faulting domain without knowing about any gate into it. And we must somehow invoke the linker in that domain. Many different solutions can be proposed to these problems depending on the details of a particular system. In general, a computing utility always has a mechanism to transfer control from the security kernel to another domain without knowing anything about that domain.

We will only mention one possible solution for the sake of completeness, but we do not claim authorship for it and we insist on the fact that different systems may require different mechanisms. Since the security kernel maintains and enforces protection, it usually has the power to dynamically and temporarily force access to any object in any domain if necessary. For instance, on many machines, the supervisor can reset the privileged mode bit at will. Consequently, even though the linker is not a gate, the security kernel can force control to jump to the linker in the middle of a faulting domain. This solves the problem of entering the domain but we still have to know where the linker is in that domain to jump to it. For that purpose we can simply store the logical address of the linker at some conventional address in the faulting domain. Hence, on a link fault, the security kernel analyzes the machine status to determine the faulting domain. It then looks up the logical address of the linker for that domain at the conventional address and forces the control to jump to the linker in the faulting domain. Initialization of the conventional location is part of the domain creation operation.

This design has a side advantage. By changing the address of the linker in the conventional location, the subject executing in the faulting domain can define any other program to be its linker. It just has to prelink its own linker with the standard linker prior to changing the content of the conventional object.

Having described how the linker is invoked in a link fault, our second topic will be to demonstrate that the symbolic link which caused the fault can be snapped with only the capabilities of the faulting domain. In the earlier description of the operation of the linker, we identified three steps in the snapping of a link:

- Identification of the symbolic name of the link
- Search for and mapping of the target object corresponding to that name
- Translation of the symbolic link into a snapped link based on the previous mapping.

The first and third steps require exclusively access to the faulting domain because that is where the symbolic link and the mapped link belong. The target object and the target domain do not contain any information about links directed towards them. The linker has access to the faulting domain and can thus handle steps one and three. If the target domain is different from the faulting domain, the second step might require information embedded in the target domain. However, searching and mapping are actually performed by the FSM in the security kernel. The security kernel can access information about any target object. Thus the linker just calls the FSM through a gate into the kernel. The FSM then searches for the target object, decides whether the faulting domain has the right to know about it, eventually maps it into the faulting address space and returns a capability, i.e. the logical address of the target object, to the linker in the faulting domain. We will see in the next chapter that in some systems, complementary information about the target object must nevertheless be extracted from the target domain. It will be shown then how this can be done.

We finally discuss the third problem, namely returning control from the linker to the faulting object The goal is that the action of the dynamic linker be entirely transparent to the faulting object. The only noticeable difference in the environment is the now translated link. Apart from this, the faulting objects expects to find everything unchanged.

The machine registers must reflect the machine status just before the hardware fault occurred. For this purpose the linker needs to restore the status of the machine.

When the linker was invoked it received a copy of the status of the machine to find out what caused the fault. Restoring this status in the machine registers must be an atomic operation to guarantee consistency of the status as a whole. It would be a protection violation to allow any domain other than the security kernel to restore the status of the machine. Restoring the machine status is done by copying data out of some object into the machine registers. If any domain could perform such an operation it could set the machine status to a pattern reflecting a subject in some other domain. This would be equivalent to jumping right in the middle of a domain and by-passing the entire protection mechanism. Hence restoring the machine status requires security kernel privileges which the linker does not have. The only solution is to have the linker call the security kernel. A gate must be installed in the security kernel for that purpose. The gate will examine the machine status it is asked to restore. If and when properly validated, the machine status is restored and control jumps back to where the fault occurred in the faulting object. Validation of the machine status to be restored must determine what domain is defined by the machine status, and verify that that domain is the faulting domain. Again, the latter mechanism described is one among several possible designs of a feature of general interest which any computing utility supports under some form. In many cases, the simple fact of trying to restore the machine status from the faulting domain causes control to switch to privileged mode in the supervisor. The restore instruction itself is the return gate. Again we do not claim authorship for the mechanisms just described.

5. Cross domain problems

The first two sections of this chapter have discussed the initialization of the security kernel and of the dynamic linker. The previous section has then discussed the handling of link faults by the operational linker. The design may therefore seem complete. It is not. We will now discuss a hidden problem which we have only indirectly approached and carefully avoided mentioning so far. The problem is directly related to the multi-domain aspect of the computing utility. It is a problem of general interest which exists in any multi-domain computing utility. Our research came across it and uncovered it for the first time. We believe that it may have been solved in particular cases almost by accident. In general, it has been ignored. Hence we will propose a general solution for it.

The linker is invoked on a link fault and completes its task by asking the security kernel to restore the machine status. It is not properly speaking called by the faulting object and does not properly return to that object. It takes no "input" or "output" arguments. Instead the objects it receives to work on are defined by the machine status automatically saved by the security kernel and the result of its computation is a snapped link The question we will now discuss is where does the linker store the snapped link so that the faulting object can later retrieve it? Or in other words, what is the nature of a logical link?

In a computing utility where information sharing is a fundamental objective, special care must be taken to organize the sharing of program modules. In order to operate, a program requires working storage to store and retrieve data. One usually distinguishes three kinds of working storage: in a PL/1 environment, these classes or types are known as external, internal static and automatic storage. Data modules or data objects as we referred to them in the thesis are examples of external storage. Many programs can refer to a particular piece of external storage. That piece is external to each program and shared by all. External storage can be created or destroyed at any time and can exist as long as desired. Automatic storage on the other hand belongs to a given program, is not shared, is created when the program is invoked and disappears when action resulting from that invocation terminates. A stack frame in an Algol machine is a typical example of automatic storage. Internal static storage shares features of automatic and of external storage. Like automatic storage it is private to one program and not shareable. Like external storage, its life time can be more than just one invocation of the program. Internal static storage by definition is allocated to a program when that program is invoked for the first time in a domain, and is destroyed only when the domain is destroyed. In other words internal static storage continues to exist between invocations of a program as long as the domain which contains it exists. Going back to the problem of information sharing in a computing utility, it is clear that procedure code (provided it is pure) can be shared by different subjects in different domains. Similarily, external storage can be shared, perhaps with some precautions: sharing external storage allows sharing data. However, it may be desirable not to share internal static, and it is certainly desirable not to share automatic storage. Let us consider the case of internal static storage. Sharing internal static storage may lead to conflicts since subjects in different domains may carry on different computations with the same procedure. Thus mutual protection and independence of domains will in such cases require different static storage areas to be allocated in each domain where a procedure is currently used. We will assume such a case in the following discussion and will propose a design which allocates static storage on a per domain basis. It should now be clear that a snapped link is a typical example of an internal static information item. It is meaningful only in a given domain during the existence of that domain. Hence in each domain where some procedure object is currently used, an instance of each link issued from the procedure is stored in the static storage area assigned to that procedure in that domain. The set of all links issued from a procedure is referred to as the linkage section of the procedure. Thus, an instance of the linkage section of a procedure exists in each static storage area assigned to that procedure in the domains where it is currently used. Both the linker and the procedure can retreive the appropriate linkage section according to some system wide convention which is left to the discretion of the designers of the system.

The hidden problem we mentioned earlier is that of deciding how static storage should be allocated when a procedure is about to be used for the first time by some subject in some domain. Often this task is left to the dynamic linker. Such awkward design results in a major protection violation instance. We will now discuss why and propose a correct design.

Clearly we do not want to allocate static storage for all programs executable in a given domain when we initialize that domain: it is impossible to scan the whole file system to find all procedures executable in the domain and allocate static storage for them; it is simply impossible to know in advance about all procedures executable in the domain because of the dynamic aspect of the file system. On the other hand we want to be certain that when a program is invoked for the first time in a given domain, static storage is already allocated for its linkage section so that the executing subject can look it up when it needs to follow a link to some external object.

The first solution which comes to the mind is to allocate the space when the object is invoked for the first time. On the assumption that all objects are invoked by symbolic names and given that all symbolic links are handled by the linker, we conclude that the linker should allocate static storage when it discovers it is snapping a link to a target object which has not yet any static storage in the target domain. Although this seems to be a clean solution, it violates protection. Indeed, if a subject could get the logical address of a target object by guessing it or by appropriatecalls to the security kernel, it might call that object directly by logical address and not by symbolic name. In doing so, it will by-pass the linker and will end up executing an object which has not been provided with static storage; this is likely to terminate the life of the subject. Protection wise it is perfectly legal as long as the subject hurts only itself in the current domain. But if the target object the subject was calling is a gate into another domain, by-passing the linker could cause damage to the target domain by not initializing some static storage as expected. This of course is a violation of the protection of the other domain. In addition, having the linker in the faulting domain allocate storage in the target domain could be very hard to achieve.

The second solution which comes to the mind and seems perhaps easier to implement is to make static storage allocation a function of the FSM. Since using a procedure in a domain requires mapping it into the address space of that domain, the FSM is guaranteed to be invoked for any procedure each time that procedure is used in a different domain. Thus the FSM could at that time allocate static storage to that procedure in the appropriate domain. The FSM is more likely than the linker to have the capabilities to do so. However this design also violates protection. Since the linker invokes the FSM, by symbolically referencing without even invoking all gates into a domain B, a domain A could create a mass of link faults causing static storage to be allocated to each gate into domain B. Such mass allocation could overflow the storage available in domain B thereby violating its protection since it would have been triggered by domain A.

As our research naturally came across the question of static storage allocation, the above problem was uncovered. Obviously another solution had to be proposed which would solve the protection problem. In addition, it was felt that static storage allocation did not functionally belong to the dynamic linker to start with. Thus a correct design, but also a much cleaner and more efficient design is proposed hereafter. It is based on the fact that static storage allocation is triggered by the domain itself where it must be allocated. Thus no protection violation is possible.

When execution of a procedure object starts, the subject must, according to the system convention already mentioned, retrieve the linkage section of the object in the current domain. We suggest that this search generate a hardware internal static storage fault (ISS fault) when and if it fails. This ISS fault should be handled by the system in a manner very similar to a link fault. It should be passed to the faulting domain. Analysis of the machine status would tell which object requires static storage to be allocated. Static storage would be created in the faulting domain for that faulting object. After the machine status is restored, the subject would successfully retry the search. Of course just like the linker had to be prelinked, the static storage allocator must have its static storage allocated at domain initialization to be operational.

The design we have just proposed guarantees the protection of all domains because static storage allocation is made independent of dynamic linking. Hence allocation is no more triggered by the execution of a random untrusted object, but by the execution of the object itself which needs static storage. The design stems from the simple fact that no object, and particularly no gate into any domain, can depend on a caller action to perform any task in general, static storage allocation in particular.

Given that links are per domain static items, it is now clear why the security kernel must communicate a copy of the linker links independently to each domain it creates. This copy is installed in the static storage area of the linker in that domain.

6. Summary

This chapter has attempted to present a complete design of a dynamic linker running outside the security kernel of a computing utility. Four main problems have been distinguished. It has been demonstrated first that the security kernel could be made operational without the help of a dynamic linker. It has been shown that the dynamic linker could be made available in all domains while being prelinked only once. It has then been explained how the linker handles link faults. Finally, the hidden although fundamental problem of static storage allocation in a multidomain system was discussed. This concludes the presentation of the complete design. The following chapter will illustrate the use of the computing utility model and the principles of the design by identifying the components of the model to those of a real world system and applying the design to that system. Concluding remarks on the actual implementation will convince the reader of the feasibility and usefulness of the design.

IV. Implementation

1. General

In developing our thesis we have first discussed a computing utility model which enabled us to give a formal description of the operation of a dynamic linker. In a second stage we have presented and discussed in terms of the the model the general design features of a computing utility where the dynamic linker is executed outside the security kernel domain. We will now build up the third level of the thesis. This level consists in demonstrating the feasibility of the proposed design by describing and analyzing the details of its implementation on a real world computing utility.

The Multics system has been chosen as a test case for the implementation. The Multics system (15-18) is a commercial computing utility developed jointly by the Massachusetts Institute of Technology and Honeywell Information Systems, Inc. It is supported by the Honeywell 6180 computer system. It implements a powerful virtual memory time sharing system with extensive information sharing facilities. In addition to being easily available for this research, Multics was a very interesting test case for our design.

Firstly, Multics was designed with protection of information as an initial objective. Protection has influenced almost all of its design features. Protection mechanisms are embedded in most of the functions available on Multics. Even the hardware of the 6180 processor was designed to support the concept of domain (15).

Secondly, a recent project has been launched with the objective of defining and auditing the security kernel of Multics to certify the correctness of the protection mechanism. Since the dynamic linker of Multics was initially designed to be executed in the security kernel environment, the present research matched exactly the objectives of the certification project.

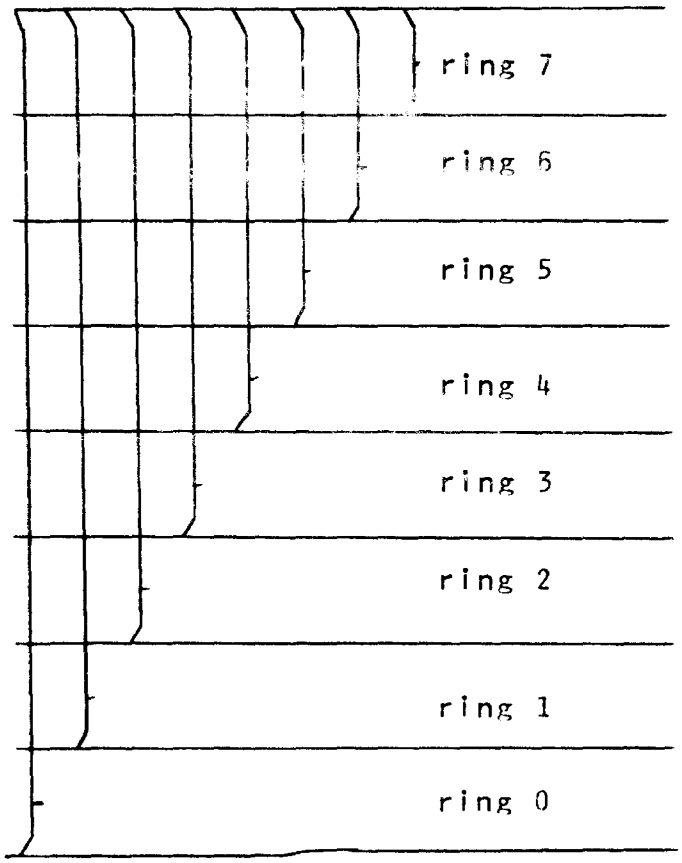
Finally, the protection mechanism of Multics matches very closely the domain protection model as described earlier. Hence there is a direct parallel between the description of the domain based design and its implementation.

We will divide the discussion of the implementation into four parts. The following two sections will at the same time briefly describe the general design features of Multics and match the real system components with the concepts of the computing utility model described earlier. The next section will then talk about a dynamic linking specifications on Multics to familiarize the reader with the nature of the functions which the dynamic linker is expected to support. The remaining sections will present the reader with a discussion of the implementation of the dynamic linker. Emphasis will be put on the discussion of selected specific problems encountered by the implementation. We do not claim that the problems to be discussed constitute an exhaustive list of all problems which the implementation faced. Out of the complete list of problems encountered during the implementation, we have carefully selected specific problems which we believe are instances of more general problems that any designer is bound to face on any computing utility under some form or another.

2. Information Protection in Multics

The equivalent of a domain in Multics is a ring (15, 18). Rings can be viewed as a set of domains with a linearly nested ordering of privileges. The set of capabilities of any given ring is a subset of the capabilities in the next most privileged ring, as represented in figure 5. The 6180 hardware processor supports up to eight rings for each user. The eight rings are numbered from 0 to 7 by decreasing order of privileges. Because every ring has at least the capabilities of the next

Figure 5: Multics protection rings.



(Brackets indicate the scope of capabilities available in the different rings.)

higher numbered ring the concept of gate exists only in the downward direction of cross-ring calls. A subject executing in ring n must ask entry permission to a gate if he wants to obtain the extra capabilities of ring m (m smaller than n). On the other hand, a subject executing in ring m and willing to move to ring n (again m smaller than n) can freely do so. The idea of a gate into ring n for ring m is irrelevant.

All users (presumably) trust the security kernel more than their own programs which may contain bugs capable of causing trouble. In turn they probably trust their own programs more than other user's programs. This relative ordering of programscan be superimposed to the relative ordering of rings. Since the security kernel is by nature the most trustworthy set of programs, it is designed to be executed in ring 0. But it must be isolated in this ring from everything else in the environment. Hence the rest of the supervisor should be rejected to ring 1. Perhaps programs under development or less sensitive programs of the supervisor should be installed in ring 2. This idea is currently being studied. User programs, commands, compilers and other tools directly related to the actions of users can be executed in rings 3, 4 and 5. The normal case is ring 4. This allows the user to execute protected subsystems in ring 3 on the assumption that everything in rings below 3 is trusted and will not subvert the subsystem in ring 3. A user can also test untrusted programs in ring 5. Rings 6 and 7 are absolutely virgin: no function of the operating system is available there. They initially have no capabilities for any gate into lower rings. Hence a user may use these two rings to install any two-ring system he wants and keep it entirely within his control.

3. Information Storage in Multics

The Multics equivalent of a *subject* is a process. A process is defined by a site of execution and a logical address space. Each process has its own address space. A process is the entity representing a user in the machine.

The address space seen by a user in a two-dimensional virtual memory of very large capacity (15). Along one dimension the memory is partitioned into segments addressed by their order number. Along the other dimension, it is addressed by word. Hence the *logical address* of an object in this virtual memory is of the form (s,w) where s is a segment number and w a word number in that segment. The format of such references limits the size of the virtual memory to 256^{K} segments X 256^{K} words.

Multics file system is a tree-structured hierarchy of catalogs. Catalogs are called directories. The leaves of the tree are called segments. A segment is the equivalent of a collection of objects in our model. An atomic object is an entry in a segment. Directories are also atomic objects. The unique identifier of a directory is the tree-name of the directory. The unique identifier of a segment is the tree name of the parent directory concatenated with the symbolic name of the segment. Directories and segments of the file system are of course mapped into segments of the virtual memory when they are used. Such mapping is supported by the FSM.

The security kernel of the operating system is shared by all users. Since it is the very first thing which has to be operational in any process, it is the first thing to be mapped into any process address space. Hence the security kernel always occupies the same locations of the virtual memory of each process. Furthermore, all rings in a process share the same address space.

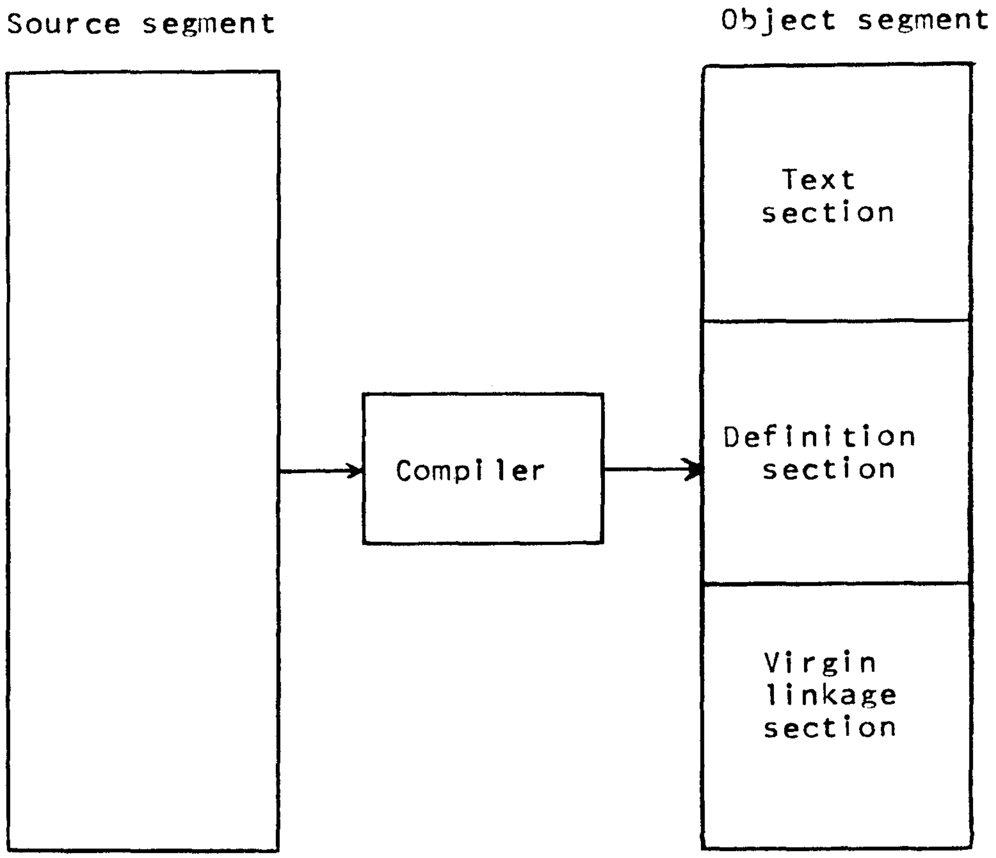
4. Dynamic linking in Multics

The previous two sections have established a parallel between the Multics system and the computing utility model of the thesis. Our second step towards the discussion of the implementation will be the statement of dynamic linking specifications in Multics.

The Multics system supports various high-level languages but was initially designed to support PL/1. Most of the system programs of Multics are written in PL/1. As the address space of a Multics process is two-dimensional it was both easy and desirable to have a two-dimensional name space for PL/1 symbolic names. An object symbolic name or entry name is of the form segname$entryname where segname is the symbolic name of the segment containing the entry and entryname is the symbolic name of the word offset where the entry is located in the segment.

Given a source program (or source segment) any compiler generates an object program (or object segment) which contains three sections as described in figure 6. The last section contains the pure executable code of the program. The definition section contains on one hand the list of entry names and word offsets of all entries in the object segment. On the other hand it contains the list of all names of entries into external object segments which this object segment may reference. Finally there is the virgin linkage section. We insist on the word virgin which is used to distinguish the present type of linkage

Figure 6: Multics object segments.

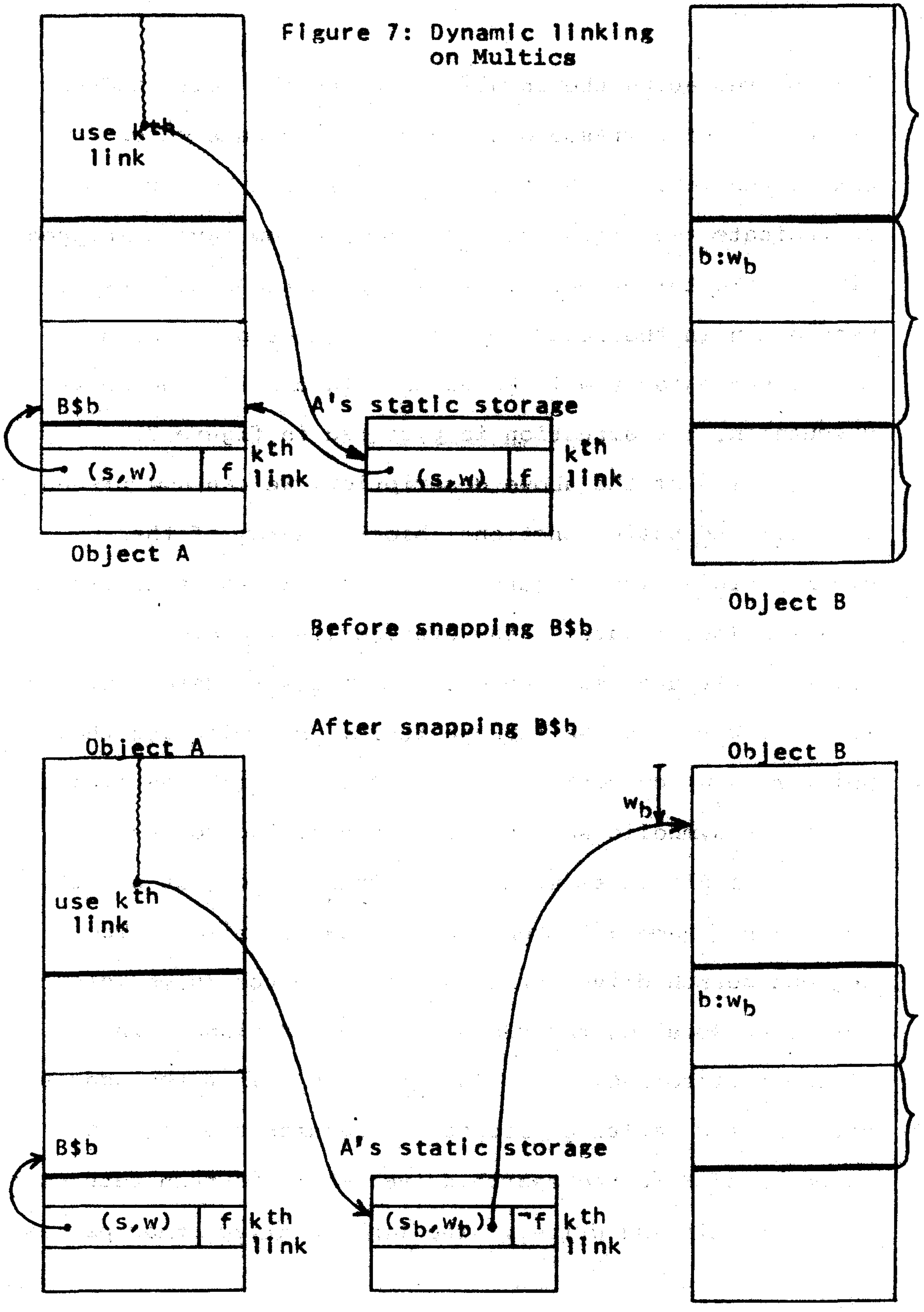


section from a non virgin linkage section which will be derived from the virgin one and is in the static storage area as described in the thesis. The virgin linkage section always remains virgin and is sharable. For each external object referenced in the source program, a link is inserted in the virgin linkage section.

A link is a triple (s,w,f). (s,w) is a logical address as defined earlier and f is a flag. In a symbolic link, the flag is always a bit pattern indicating that (s,w) is invalid. Attempting to use (s,w) as such will cause a link fault. At this point (s,w) somehow points to the symbolic name associated with the link, in the definition section and not to the target object of the link. When the object segment is first executed in a ring, static storage is allocated for it in that ring. The virgin linkage section is copied into the static storage area yielding a non-virgin linkage section. The address of the non-virgin linkage section is stored in a conventional location where an executing process can always retrieve it when it uses the object segment. When execution encounters a reference to an external object, the linkage section address is used to look up the corresponding link. This triggers the hardware fault since (f) is set. As a result of it, the linker will snap the link by replacing the invalid (s,w) by the valid address of the object corresponding to the entry name which caused the fault. The fault flag (f) will be turned off to indicate the validity of (s,w). We now have a snapped link to the target entry. If and when the same link is used again in the future by the same user process, no more linkage fault will be taken. To clarify the above discussion, the situation is pictured in figure 7.

In view of the above description, we can now present a simplified basic functional block diagram of the dynamic linker (see figure 8). On a link fault caused by object A (see figure 7) the dynamic linking driver is invoked. It analyzes the machine status to determine which link caused the linkage fault. By following the pointer (s,w) currently in the symbolic link, the linker finds the symbolic name B $ b correpsonding to that link in the definition section of the faulting object A. It then passes name B to the segment search driver. The segment search driver tries a set of search rules (directory treenames) on the FSM until the FSM finds B in one of the directories. The FSM then maps B into the address space of the faulting process and returns the segment number s of B to the search driver which in turn returns it to the linking driver. The linking driver then passes

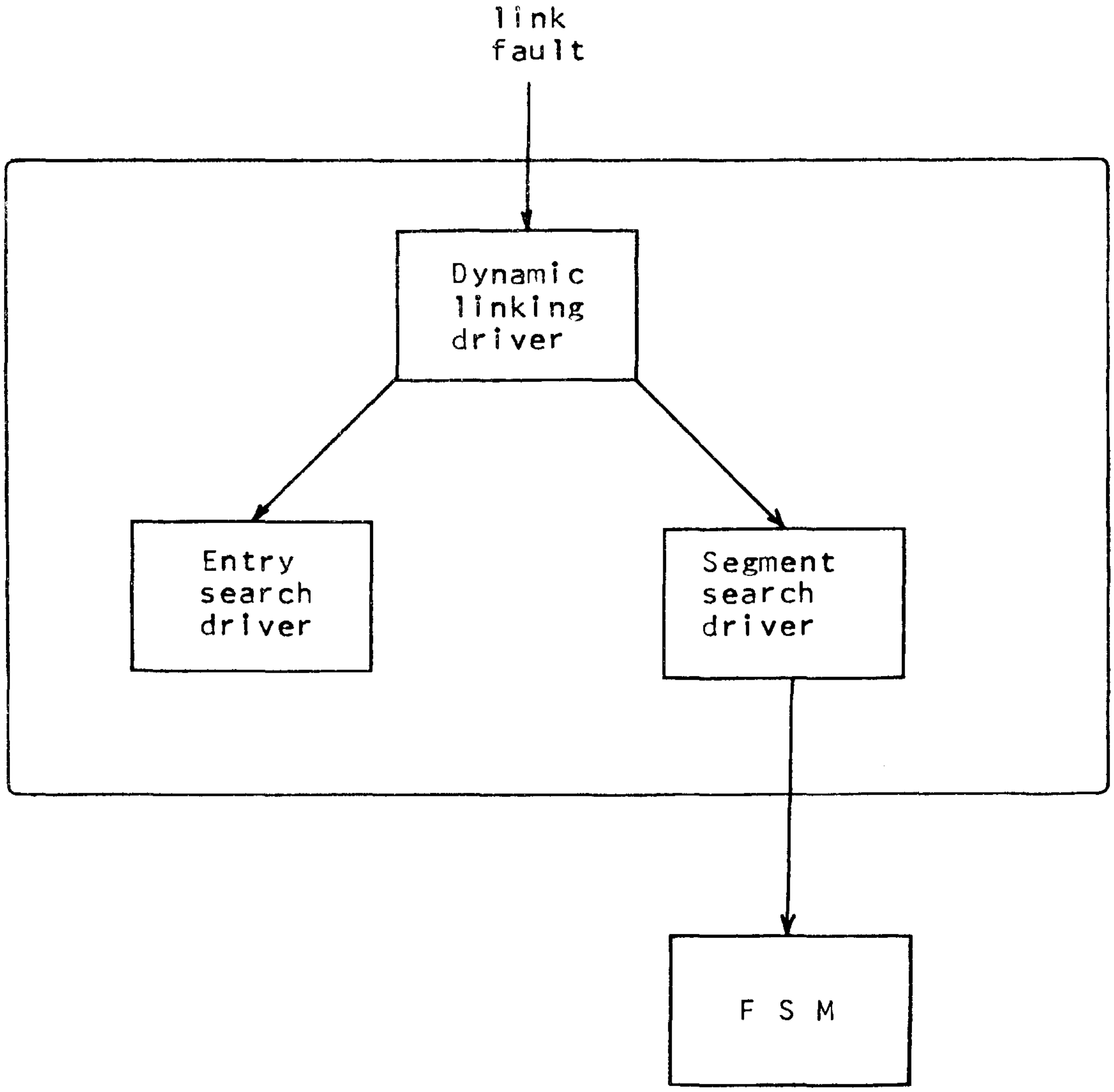
Figure 7: Dynamic linking on Multics



Before snapping B$b

After snapping B$b

Figure 8: Functional diagram of the Multics dynamic linker.



the segment number s and the name b to the entry search driver. This one scans the definition section of segment numbered s (i.e. B) until it finds the name b. It then returns the offset w of b in B to the linking driver. The dynamic linking driver finally replaces the address (s,w) in the symbolic link by the address (s,w) of B $ b and turns off the flag (f) to make the link a snapped link. The machine status can then be restored and execution can proceed.

We do insist on the fact that the above description is a simplified strictly functional definition of the linker. In no way should it be assumed that the linker contains only three modules and that linking happens as naturally as we described it. In the course of this chapter we will progressively complicate the description we have just given and discuss the problems encountered by the implementation. This section concludes the descriptive part of the chapter. We will now apply our design to Multics and present selected aspects of the implementation.

5. Initialization

In this first section about the implementation of the design, we will outline how the security kernel and the linker are initialized. This outline will be brief because no particular problem was encountered. The implementation of the design was relatively straightforward.

The Multics system is initialized by a dedicated initializer process. All modules of the security kernel are loaded into the system from a generation tape. Immediately after the loading, the virtual memory addressing mechanism is initialized so that the initializer process sees a regular virtual memory with the restriction that the capacity of that virtual memory is temporarily constrained to that of the real memory. A prelinker is then invoked to link together all modules of the security kernel which are read in from the tape. After the prelinker is run, miscellaneous initialization tasks are performed. When the security kernel is entirely operational, the prelinker, as well as other initialization programs are unmapped and thrown out of the addressable space. We have described this mechanism for the sake of completeness. However it existed before we implemented our design. We used it as a basis for our implementation.

We now turn our attention to the initialization of the linker. Since the security kernel is initialized by a prelinker, it is all but natural to use the same prelinker a second time to initialize the linker. Actually the implementation uses a hybrid technique involving both a binder and a prelinker. Multics provides its users with a binder of which the goal is to take several object segments and to merge them into one which has only one text section, one definition section and one virgin linkage section. Of course any link between the original distinct object segments submitted to the binder are directly translated into relative offsets within the resulting bound object segment. The binder was used to bind together the modules of the linker, i.e. the modules inside the main box of figure 8. Consequently the only links issued from the bound linker, which the binder could not translate are links to the FSM and links to external data bases. Notice that figure 8 shows only one link to the FSM. In reality there are several such links. As we said earlier figure 8 is only a simplified functional diagram. To be more accurate too, the links to the FSM are actually links to ring 0 gates since the FSM is in the security kernel and is accessible only through these gates. Also the links to external data bases are not represented in figure 8. The external data bases are error code tables and system data tables. They are used by the linker but are not really part of it and do certainly not belong in its functional diagram.

The task of the prelinker is thus to snap the links from the bound linker to the external data bases and to the security kernel gates. The operation of the prelinker matches exactly that described in the general case. Since the prelinker does not know about any file system, (even though the bound linker, the external data bases and the security kernel gates are catalogued in the file system and stored on secondarymemory) a copy of each module must be loaded into the initializer address space from the system generation tape. The bound linker is loaded with attributes such that it does not get prelinked as a module of the kernel. Instead when the kernel is initialized and just before it throws the prelinker out of its address space, it invokes the prelinker a second time to prelink the bound linker. The prelinker builds a fictive mapping table and a snapped links table as stated in the general design. In the particular case of Multics, the snapped links table is simply a copy of the virgin linkage section of the bound linker where all symbolic links are replaced by snapped links reflecting the fictive mapping. The fictive mapping table is a little more interesting. Since there is only one address space per process common to all rings instead of one per process and per ring, the reader may wonder why a fictive mapping of the linker, the data bases and security kernel gates is necessary. Couldn't they just stay where they are? The answer is negative because of the peculiar way the security kernel is mapped into each process address space. It is a convention that all segments which are part of the security kernel during regular operation are mapped into the lowest segment numbers of each process address space. Hence all lowest segment numbers are reserved for the kernel and constitute some sort of private address space. No such segment number is ever used outside the kernel. Hence, even though the linker, the external data bases and the security kernel gates are in the address space during initialization, they must be remapped into higher segment numbers for the higher numbered rings. That fictive mapping will be valid for all rings (1 to 7) of all processes. To summarize the problem, although the address space of a process is common to all rings, a fictive mapping must be installed by the prelinker because some specific rule cuts a piece out of the process address space and turns it into what may be regarded as a private kernel address space. If this rule did not exist, clearly, the initial mapping could be kept and be the final real mapping. After the two tables are generated, the security kernel throws away its capabilities to access the prelinker, the linker and the external data bases by simply deallocating their current segment numbers. Remember that the linker and the data bases are still stored in the file system on secondary memory, so that the system can retrieve them there later on when they will be needed. Of course the two tables built by the prelinker may not be thrown away. Since they will be used throughout the life of the system each time a ring is created, they must remain permanently in the address space of the kernel.

We finally come to discussing the task of enforcing the fictive mapping. This task is also straightforward and identical to the general design. In order to operate correctly, Multics object segments need a static storage area and an automatic storage area. Automatic storage is allocated in a special segment called the stack. This segment is used as an Algol call stack. Static storage is allocated in a special segment called the combined linkage segment (cls). There exists one stack and one cls per ring and per process. There exists a system wide convention stating that the stack of a given ring always occupies the same segment number in the address space of any process. This enables any process to find the right stack in the right ring. Each stack header contains (conventional) the address of the cls for the same ring. This enables any process to retrieve the right cls for the right ring. Given these two conventions, it is clear that no process will ever be able to touch its cls in a ring before it touches its stack in that ring. Hence the convention is that when the process uses its stack segment number for the first time, a hardware fault occurs which is interpreted as a ring initialization fault and triggers action of the kernel to initialize the ring. When the stack and the cls for that ring are initialized, the kernel invokes the FSM. As stated in the general design, the FSM uses the fictive mapping table prepared by the prelinker to map the linker, the external data bases and the security kernel gates in the process address space. Finally the kernel copies the snapped links table built by the prelinker into the cls just fabricated for the new ring. Control is then restored into the new ring. The linker has been mapped into the address space and its non-virgin linkage section containing only snapped links exists in the cls of the new ring. Thus the linker is operational in that ring.

The last question which needs perhaps a brief comment is why do we need to invoke the FSM each time a ring is initialized in a process? Doing so for the first ring should be enough since the address space in which the FSM enforces the fictive mapping is the same for all other rings. Our implementation is justified by an aspect of the Multics virtual memory. In mapping a segment into a segment number, one needs to specify the unique identifier of the segment and the ring on behalf of which the mapping is done. Once the bound linker for instance is mapped into its final address for one ring all rings will see the address occupied but it will not be meaningful to them until they also require the linker to be mapped there on their behalf.

This discussion completes the section on initialization of the kernel and of the linker. It has been demonstrated that straightforward implementation of the design was possible on a computing utility like Multics. No major problem and no particularly interesting issue was raised so far. Now we have shown how to implement an operational linker, we will proceed by showing how to invoke it in the faulting ring on a link fault.

6. Fault Handling

We have shown how the Multics dynamic linker was made operational in a ring. Our next step is to show how link faults are passed to it and how it can return control to the faulting object. Again this can be done by a straightforward application of the design, using pre-existing mechanisms.

All faults on Multics are intercepted by a special module of the kernel. This module existed already in the initial version of Multics and its purpose is to analyze and sort faults. Just a few lines of code had to be modified so that link faults would be directed to a signalling module instead of being directed to a ring 0 linker. The signalling module of the kernel existed as well in the initial version of Multics. It is already used to signal events other than link faults in outer rings. Because of the hierarchy of rings, the security kernel and the signalling module in particular can access any object in a higher numbered ring and can switch the ring of execution of a process. These privileges are exploited to signal a link fault. When the signalling module receives a copy of the machine status saved by the fault interceptor module, it analyses it to determine the number of the faulting ring, and the segment number of the stack used at fault time. It then makes a stack frame for itself on that stack and copies into it the machine status. It copies as well a return address to be used by the linker. It finally switches ring of execution and calls the linker. The address of the linker is found in the stack header (conventional). This address must be set at ring initialization and may be changed by the process if it wants to define another linker of its own in that ring.

Let us assume for a moment that we know how the linker itself works and suppose that it has snapped the faulting link and wants to restore control to the faulting object. The linker simply returns to the signalling module in the current ring. The signalling procedure then calls a gate into the kernel. The purpose of this gate is to validate the machine status returned to it by the signaller and to restore it. Validation simply consists in verifying that the status reflects a ring of execution not lower than the faulting ring. This is to make sure that the linker which handled the status in the faulting ring did not maliciously set it so that control would be restored in a lower numbered ring than the faulting ring, which of course violates protection. The gate then destroys the signalling stack frame in the faulting ring to make the stack look as if nothing had happened. Restoring the status is finally done in one indivisible hardware instruction which reloads all the machine registers, thereby forcing control back into the formerly faulting object.

7. The dynamic linker

The last two sections have discussed respectively the prelinking of the linker and the handling of link faults. It remains to be demonstrated how the linker itself can be implemented to translate links properly. So far the implementation did not encounter any major problem or any operation of outstanding interest. In this section we will only very briefly outline the implementation as a whole and then concentrate on selected interesting features of the Multics system of which the implementation cannot be derived directly from the global design principles. As we mentioned it before, these selected topics are only instances of broader problems which any designer would face in any computing utility perhaps under different aspects.

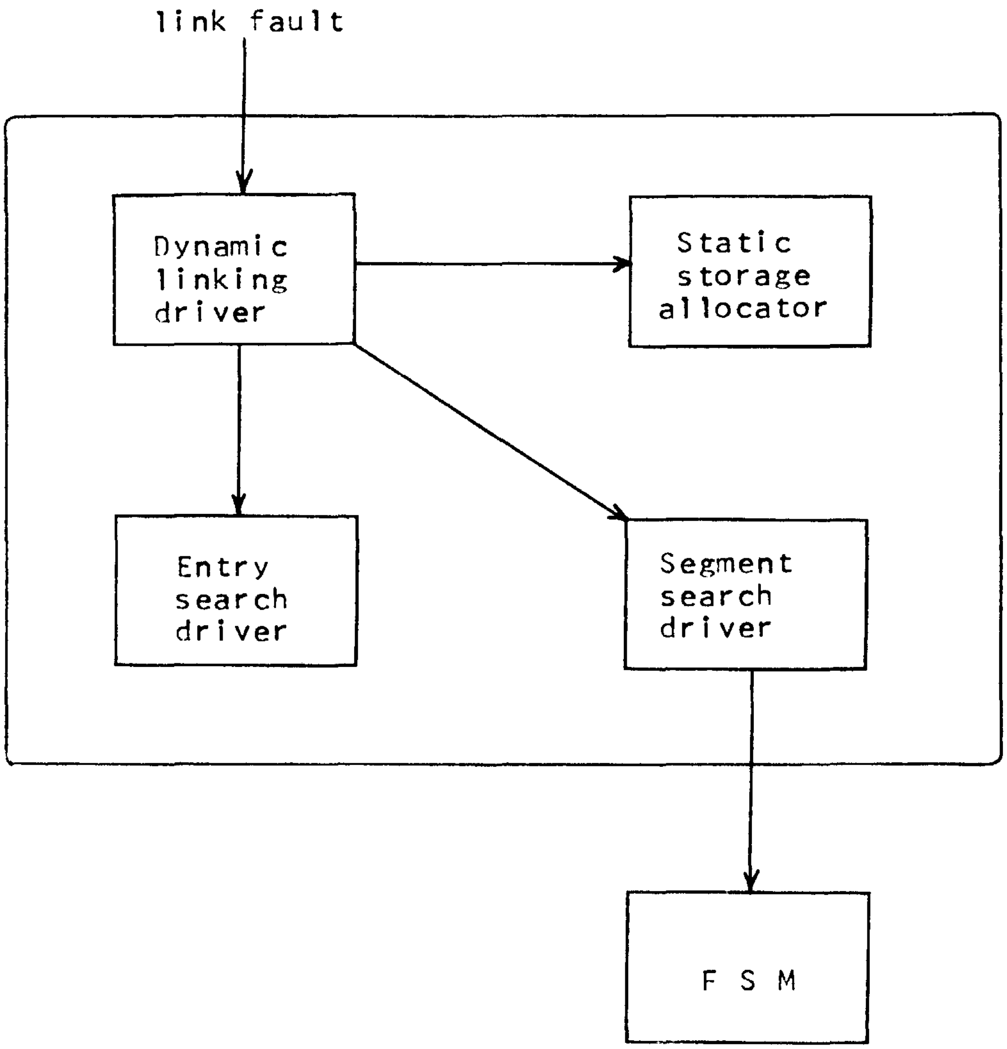
The starting point of the implementation is the block diagram of figure 8. The basic dynamic linker is programmed according to the functional specifications of that diagram. This basic linker contains a dozen independent program modules. Once compiled, the resulting object segments are bound together by the binder. A bound object segment results which contains about forty links to data bases and kernel gates and can itself be invoked through about fifteen different entries; one of which is the main link translation entry used for link faults.

On top of this basic linker we will now progressively add other features, functional boxes and specifications as we go about discussing specific implementation problems.

a. Implementation of peripheral features

Let us first turn our attention to the question of static storage allocation. As we mentioned it in the chapter about the global design, static storage allocation is a general problem which must be solved in any computing utility. The wrong way of solving it is to leave it in the responsibility of the linker. One correct way to solve it is to install a hardware fault which we called the ISSF. When a process attempts to get a hold of the address of the static storage (non-virgin linkage section) of the program it is executing and if that storage is not yet allocated, a ISSF occurs which triggers storage allocation. The old design of the Multics dynamic linker was such that static storage allocation was part of the linker task (see figure 9). On snapping a link, the dynamic linking driver used to always verify that the target of the link did have static storage in the target ring. As stated in the thesis, this design violates protection because a target object in a target ring cannot depend on a faulting object in a faulting ring to use the linker and allocate static storage where appropriate. In addition, even if this was not a protection violation, it would simply be impossible for the new linker in a faulting ring to allocate space in a target ring if the target ring is

Figure 9: Old Multics dynamic linker.



lower than the faulting ring. This was possible in the old design because the linker was in the security kernel and could access any ring.

Consequently we have proposed to implement a hardware ISSF as described, such that dynamic linking and static storage allocation are functionally distinct. Yet there is still one advantage in keeping them physically together (see figure 10). Keeping dynamic linking and static storage allocation physically together means keeping them in the same bound object segment, the bound linker. Thus they are prelinked and initialized together at the same time. Adding the static storage box in figure 10 increases the complexity of the dynamic linker but does not increase the complexity or modify the design of prelinking and ring initialization.

The operation of the linker is thus as follows. Assume object A in ring 4 wants to invoke gate B in ring 3. Whether A invokes B by symbolic name (link fault) or directly by its address it happened to already know is irrelevant. When execution moves to the target segment B in ring 3, as soon as segment B tries to find a presumably unallocated static storage, an ISSF occurs which results in the linker (static storage allocator part) to be invoked in ring 3. Allocation can and will thus safely

Figure 10: New Multics dynamic linker.

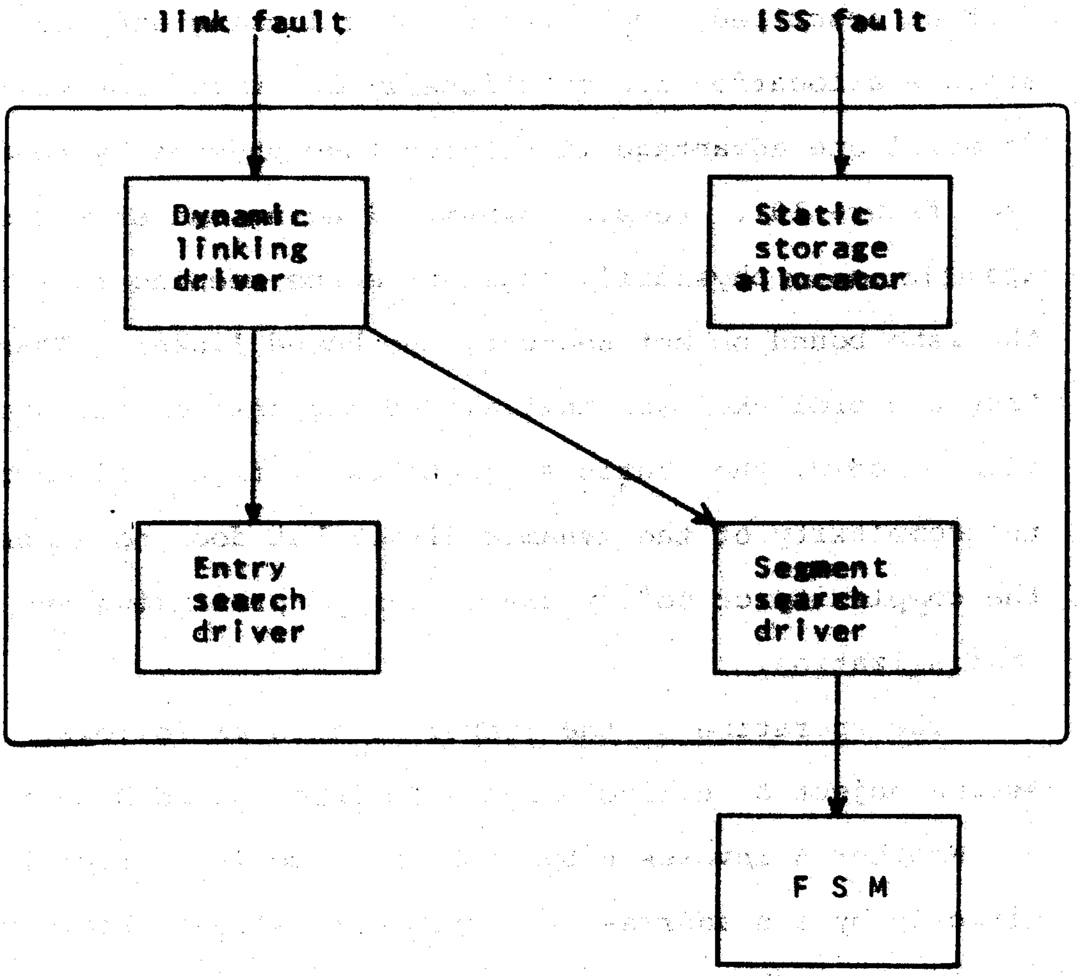
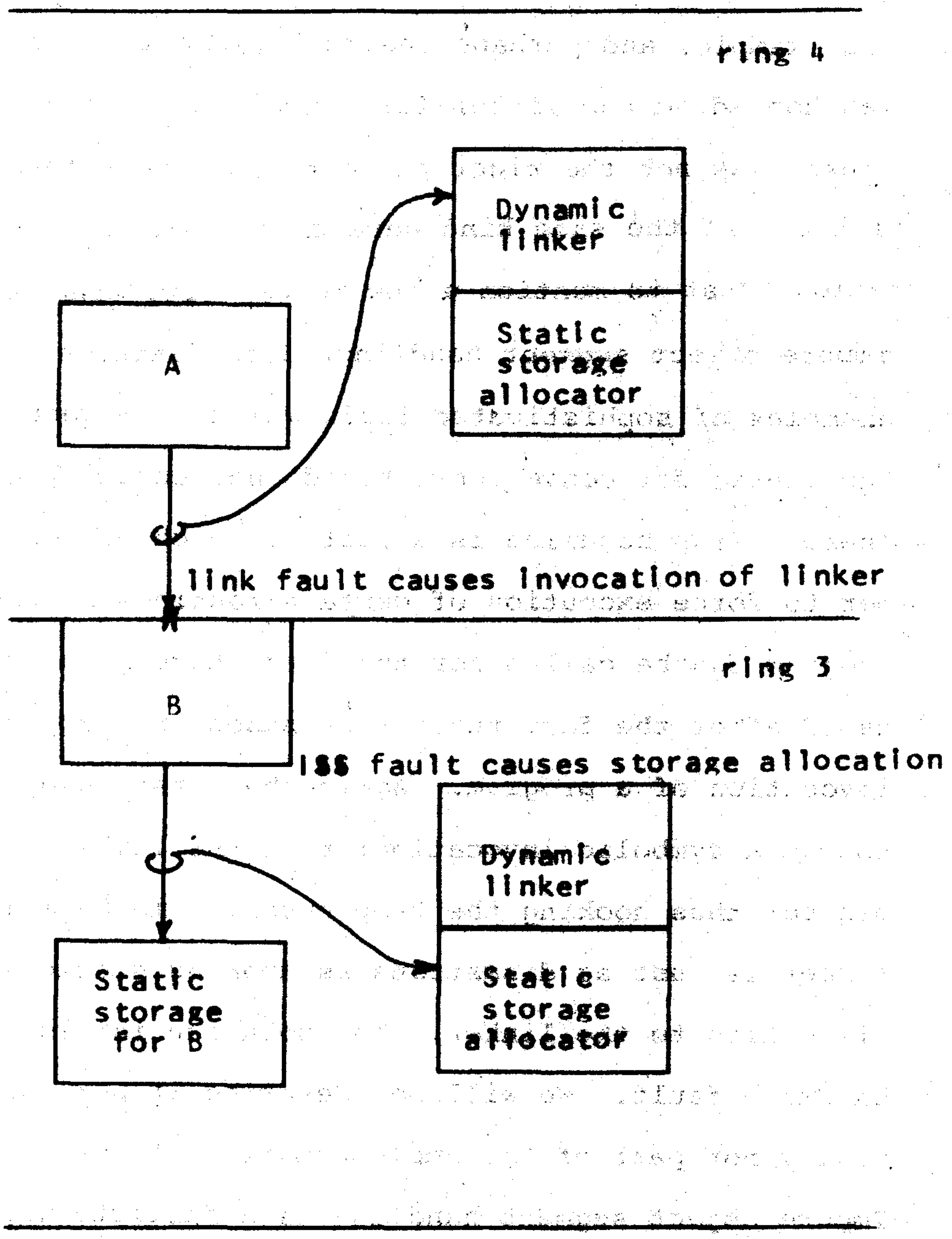


Figure 11: Static storage allocation on Multics



occur. This is pictured by figure 11.

The problem of static storage allocation was just one example, and perhaps the most typical, of a feature which was hooked to the linker for convenience. Unfortunately, the linker was not the right place to hook that feature to. Other problems of the same kind were encountered during the implementation. Just to mention a few we can cite trap handling and impure object segment handling. Such features are typical examples of sophisticated tools which have been hooked to the linker for convenience but do not actually belong there. Trap handling is a feature which allows a programmer to force execution of certain routines before his program can be called for the first time. The feature is named after the fact that it is based on trapping the first invocation of a program. Again the first invocation may not be a symbolic invocation; thus the linker can be bypassed; thus hooking the trap handling mechanism to the linker is just as disastrous as hooking static storage allocation to the linker. The solution is also to use a hardware fault. We will not describe it here as it is really not part of the implementation of the linker. Impure object segment handling is a facility which provides users with the ability of creating an object segment and then writing into it perhaps over the definition and virgin linkage sections. Of course such an object segment is not sharable. It is important to save the definition and virgin linkage section before they are overwritten (by copying them into the cls before the first reference). Such task was left to the linker. Again it did not belong there. By-passing the linker and thus not saving the definition and linkage sections could cause damage to the object segment. In addition it did put an extra burden on the linker by always forcing it to check for writeable object segments. The solution to the problem is to always save the definition and virgin linkage sections of a writeable object segment in a separate segment when the object segment is created. Compilers can take care of this very easily and already use such mechanisms to handle other features on Multics.

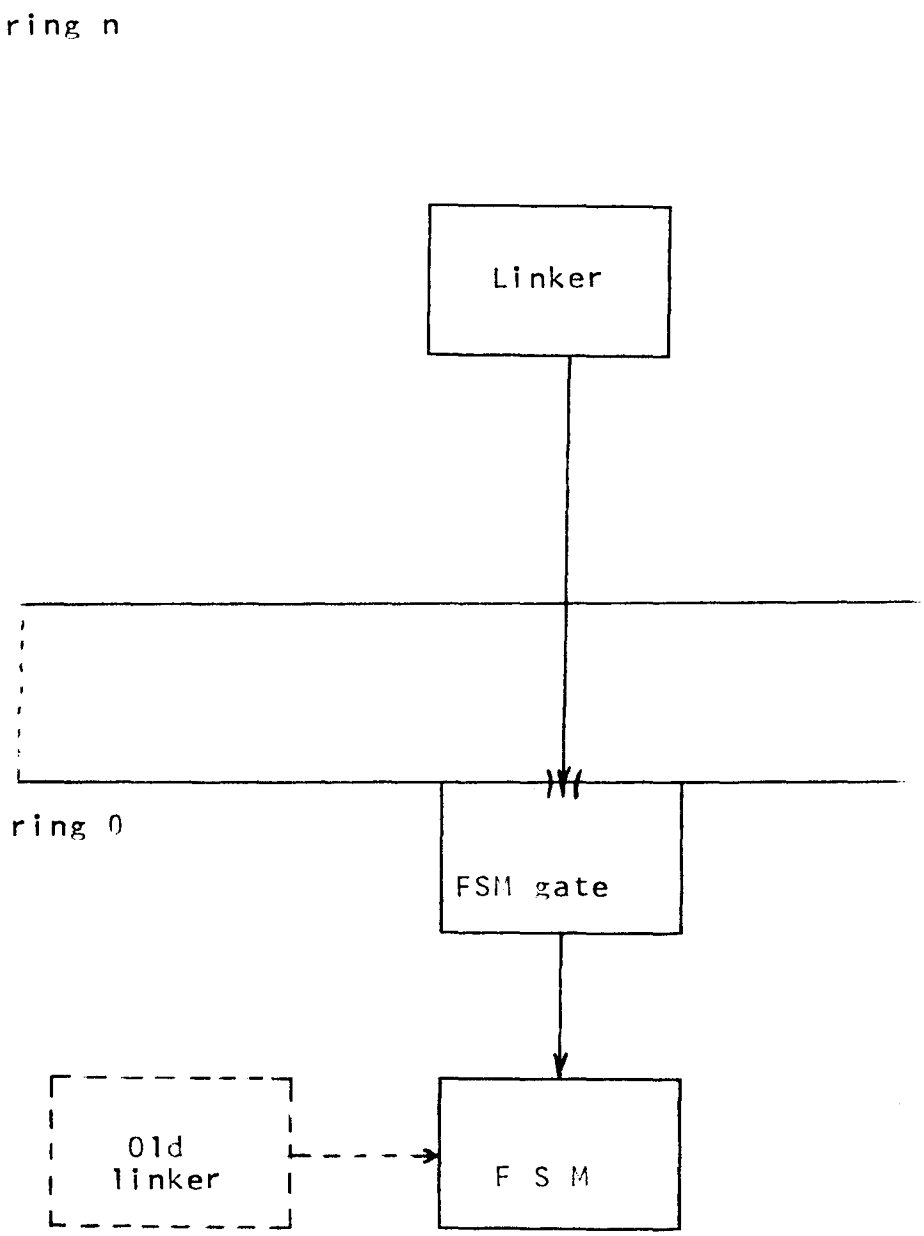
Static storage allocation, trap handling and impure object segment are typical examples of peripheral features which have been hooked to the linker for convenience. As a result, they were mishandled, violated protection, complicated the linker and interfered with it performance. Our design has corrected that situation.

b. Compatibility of interfaces

We would now like to mention a second problem which the implementation encountered. This problem is specific to Multics but problems of the same kind would certainly arise in any computing utility. The present problem does not have so much to do with the linker itself as it has with the general idea of pulling a module outside the kernel.

Any program which is part of the kernel is very likely to use other functions of the kernel. In trying to pull that program outside the kernel, one must make sure that it still can use the other kernel functions as it did before. In the particular case of the linker, the old Multics linker used the FSM inside the security kernel. Of course, once the linker is pulled outside the kernel, it cannot call the FSM directly. All it can do is invoke it through appropriate gates (see figure 12). Fortunately the FSM of Multics was already available to the higher rings through such gates. We did not have to implement them. However the interface to the FSM across these gates is not the same as the interface which the linker used to see directly inside ring 0. Directories are currently implemented as ring 0 data bases. Their logical address in a process is also a protected item. User rings (1 to 7) may talk about directories only by treename and not by segment number. Directory segment numbers are exclusively used inside the kernel. Thus when the linker was inside the kernel, the search rules

Figure 12: Interface of the linker to the FSM.



it used across the interface with the FSM were a set of directory segment numbers. Now the linker is moved outside the kernel, directory segment numbers are not suitable directory unique identifiers. Therefore the linker must use directory treenames. This implementation of search rules has the disadvantage that for each directory searched or each link fault, the treename presented to the FSM gate must be converted into corresponding segment number to perform the search. Such conversion is costly and has a negative effect on the performance of the linker. A parallel project is currently on its way to make directory segment numbers available in user rings. Such a design will restore the interface to the FSM which the linker used to see. However it has some major protection implications of which the solution is not obvious. We will not discuss these implications here.

The problem of the search rules was a typical example of a compatibility problem. By removing the linker from the kernel, we were forced to make it compatible with the interface of the kernel seen by the user rings.

c. Limitation of Privileges

The last problem which we propose to discuss will illustrate the impact on the capabilities of a program of removing that program from the kernel. The problem deals with snapping downward cross ring links, a feature which the ring the linker used to support very easily and which is now complicated by the fact that the linker is in the faulting ring.

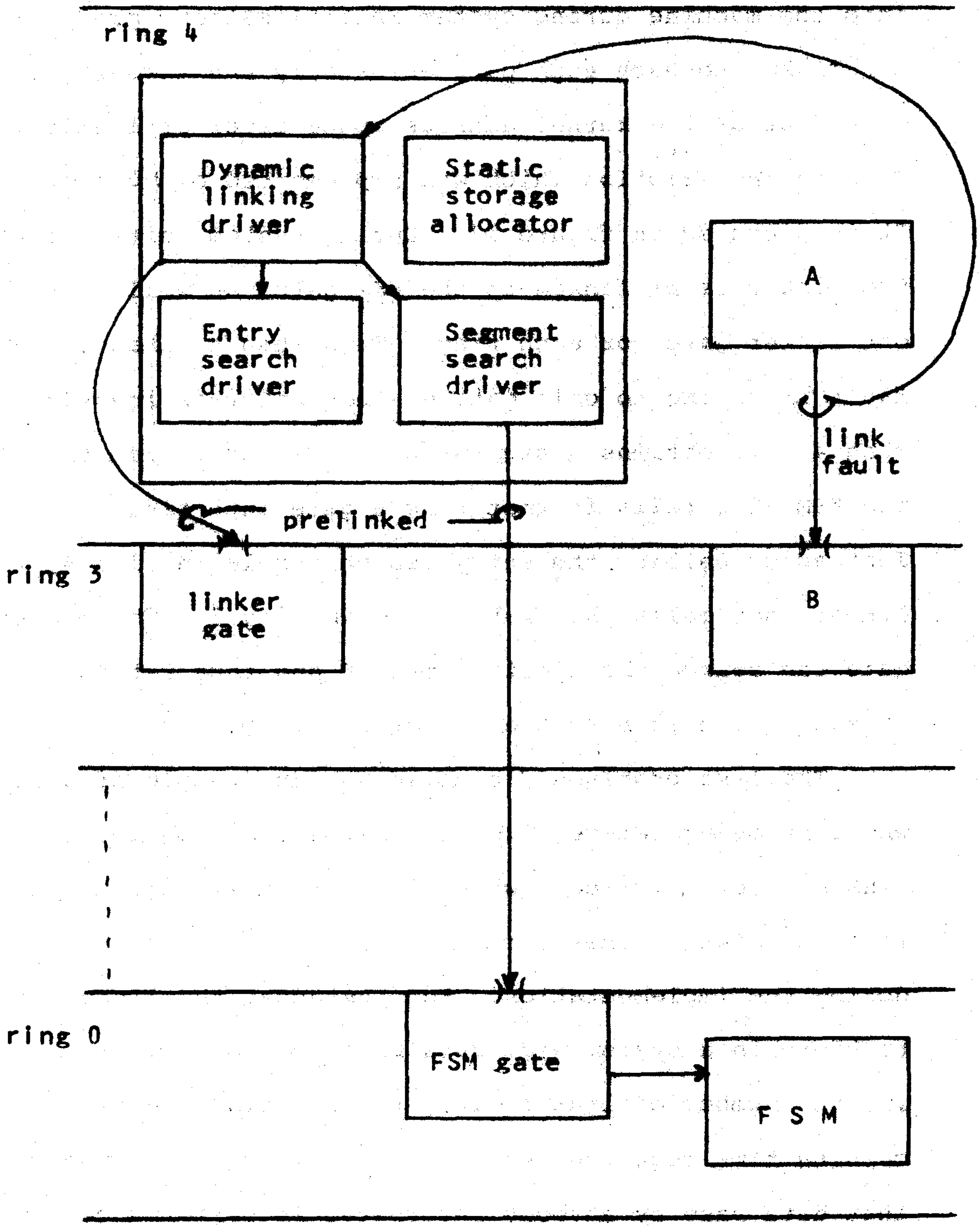
In the general design described earlier, the FSM was described as a security kernel primitive which given a catalog unique identifier and an object symbolic name returns a logical address. On Multics, this is not the exact function of the FSM. The FSM takes a directory treename and a segment name and returns a segment number. The difference between these two descriptions is that a segment name is not an object symbolic name and a segment number is only a partial logical address. As a consequence a search of the definition section of the target segment must be performed to find the offset of the target object in the target segment (see figure 8). When the target object is in a ring equal to or higher than the faulting ring, such search poses no problem. But when the target object is a gate into a ring lower than the faulting ring, the linker in the faulting ring does not have the capability to read or search the target segment. The old linker executing in the kernel did have that capability.

When snapping a link to a gate into a lower numbered ring, the linker must extract the offset of that gate from information contained in the target segment containing the gate. The only way to extract information from that target segment is to invoke another gate, a linker gate, into the target ring. The function of the linker gate is equivalent to the function of the "entry search driver" in figure 8. But the search happens in the target ring instead of happening in the faulting ring.

The question which the reader is now entitled to ask is how does the linker know about the linker gate in the first place? There are several possible answers to this question. One way the linker could know about it is by conventions. It would be possible to impose that any ring contain a gate named after its own ring number and located in a segment of some conventional directory. The linker could then invoke the FSM to obtain a segment by giving the FSM the name of the conventional directory and the conventional name of the gate into the target ring. It would thus receive a segment number. Then, using a conventional offset into that segment, it could dynamically fabricate for itself a link to the linker gate. Such design is feasible and very appropriate if there was a large number of rings per process. However we know that the number of rings per process is finite. Thus there is a much simpler solution to our problem which consists in providing the standard Multics system with a finite set of gates (one per ring), loading these gates into the machine during system initialization, prelinking the linker to each such gate as usually and throwing the gates out of the kernel address space after prelinking. This is the solution which was implemented on Multics. It is pictured in figure 13. During system initialization, the linker is prelinked to the FSM gates as well as to one linker gate for each ring. Then when A takes a link fault in trying to call gate B, the linker is invoked in ring 4. It obtains a segment number s for B from the FSM. The FSM also tells it that B is a gate into ring 3. Instead of calling the entry search module in ring 4, the linker then calls the linker gate in ring 3. The linker gate can search the oject segment B and thus returns the offset w of b in B to the linker in ring 4.

The last problem discussed was an example of a case where by being removed from the kernel, a program, the linker, lost privileges which it used to exploit to perform its task. Other such examples were encountered during the implementation. For instance, the linker used to store in a system wide data base, various meters counting the number of link faults, the distribution of processing time required, etc. Data could be extracted from that data base by anybody interested in performance. Of course, now the linker is in user rings it could still do

Figure 13: Cross ring linking in Multics



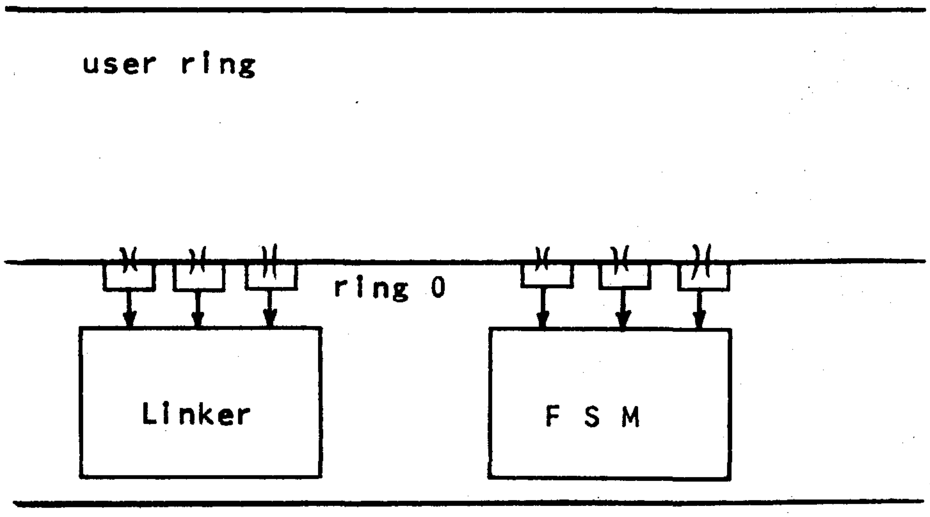
such metering, but results could not be trusted because the system wide data base would have to be accessible in user rings too. Hence anybody could write garbage into it. The solution which we propose instead is to just keep a count of link faults in ring 0. This is done by the fault interceptor module. The count is thus protected. Other meters can be stored in per ring data bases if the user desires. Such meters would of course reflect only the activity of that user in that ring.

This is the last problem we proposed to present here about the implementation. In no way do we suggest that the implementation faced no more problems than explained here. The problems presented here were just typical examples representative of classes of problems relevant to the topic of our research. Problems not discussed here either fell into categories for which we have given examples or into categories not relevant to our thesis topic.

V. Conclusion

To conclude this thesis, we would like to step back and consider the design and its implementation as a whole to summarize what has been achieved, try to abstract the main results of the thesis, and examine the cost of the implementation.

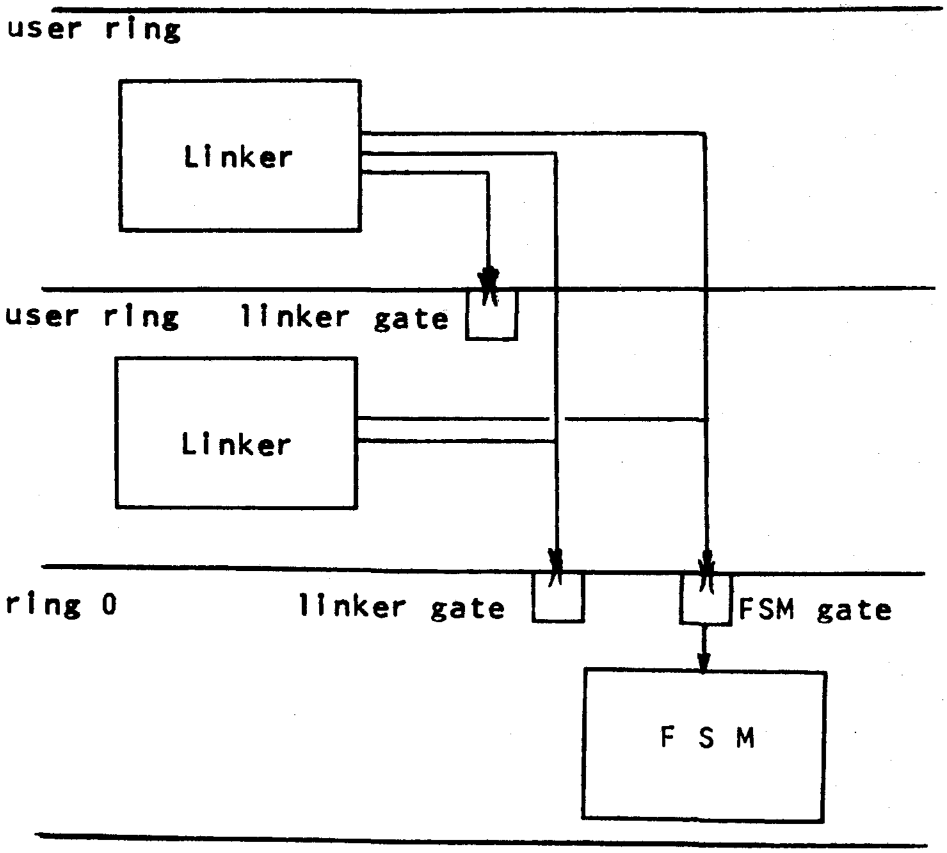
We first propose to compare the old design of with the new design we have implemented. Our comparison is based on figure 14. The old dynamic linker was part of the security kernel. It was constituted by a set of modules scattered across the whole kernel. Some of these modules were directly available to the user through appropriate gates into the kernel (see Appendix). Miscellaneous peripheral functions like static storage allocation and trap handling were directly hooked to the linker inside the kernel. The new dynamic linker is a bound object segment. Capabilities to use it exist in all rings except ring 0. The modules of the dynamic linker which used to be available through gates in the kernel are now directly available in user rings. All peripheral features have been detached from the linker and are now handled independently as described earlier. The static storage allocator is still physically connected to the linker to simplify initialization, but it is functionally independent: its operation is



Old configuration

Figure 14: Multics linker

New configuration



triggered by a special hardware fault. As a result of the above facts the complexity of the security kernel has been reduced by a non-negligible, although hard to measure, amount. What can be measured is the reduction of the size of the kernel. The following items have been extracted from the kernel:

15000 words out of 300000 (5%),
30 entries out of 1200 (2.5%),
15 programs out of 300 (5%),
18 gates out of 165 (11%).

The case of the gates is particularly interesting. Since the linker has been removed from the kernel, all gates which used to lead to it inside the kernel could be removed too. The figure of 11% deserves a special comment. Since the interface between the kernel and the outer world is one of the most sensitive, directly threatened part of the kernel, a reduction of size of 11% is a significant improvement. We attribute this high score to the fact that the linker was, as we have shown, essentially a user ring program. Thus even though it was in ring 0, it was natural that it be available to user rings through many gates.

Secondly we propose to discuss the results of the thesis. A first result is the demonstration of the feasibility of the design. Some components of the design have not been implemented because they were thought to be of minor importance and could not have any impact on the overall success of the implementation. Other components of the design like the functional independence of the static storage allocator could not be implemented simply because the supporting hardware is not yet available on Multics. However it was approximated by software and when the hardware becomes available, only a simple change of a few lines of code is required to separate static storage allocation from dynamic linking. On the whole thus the major aspects of the design and of the implementation have been verified to work correctly. System initialization, fault handling and dynamic linking have been implemented. All features crucial to the operation of the linker itself have been extensively tested and proved to work under all circumstances. In particular cross-ring linking was carefully tested.

The second result of the thesis is the improvement of the protection and the certifiability of the kernel of Multics. Size and complexity have been reduced in the proportions mentioned above thereby making the auditing of the kernel an easier task. In addition, the thesis has corrected some bugs in the Multics system. The protection threat resulting from having peripheral features hooked to the linker has been eliminated. The protection of the kernel itself is no more threatened by the uncontrollable operation of the linker. Moreover the careful study and the redesign of the linker uncovered and remedied several unsuspected protection flaws, not the least of which is the problem of static storage allocation.

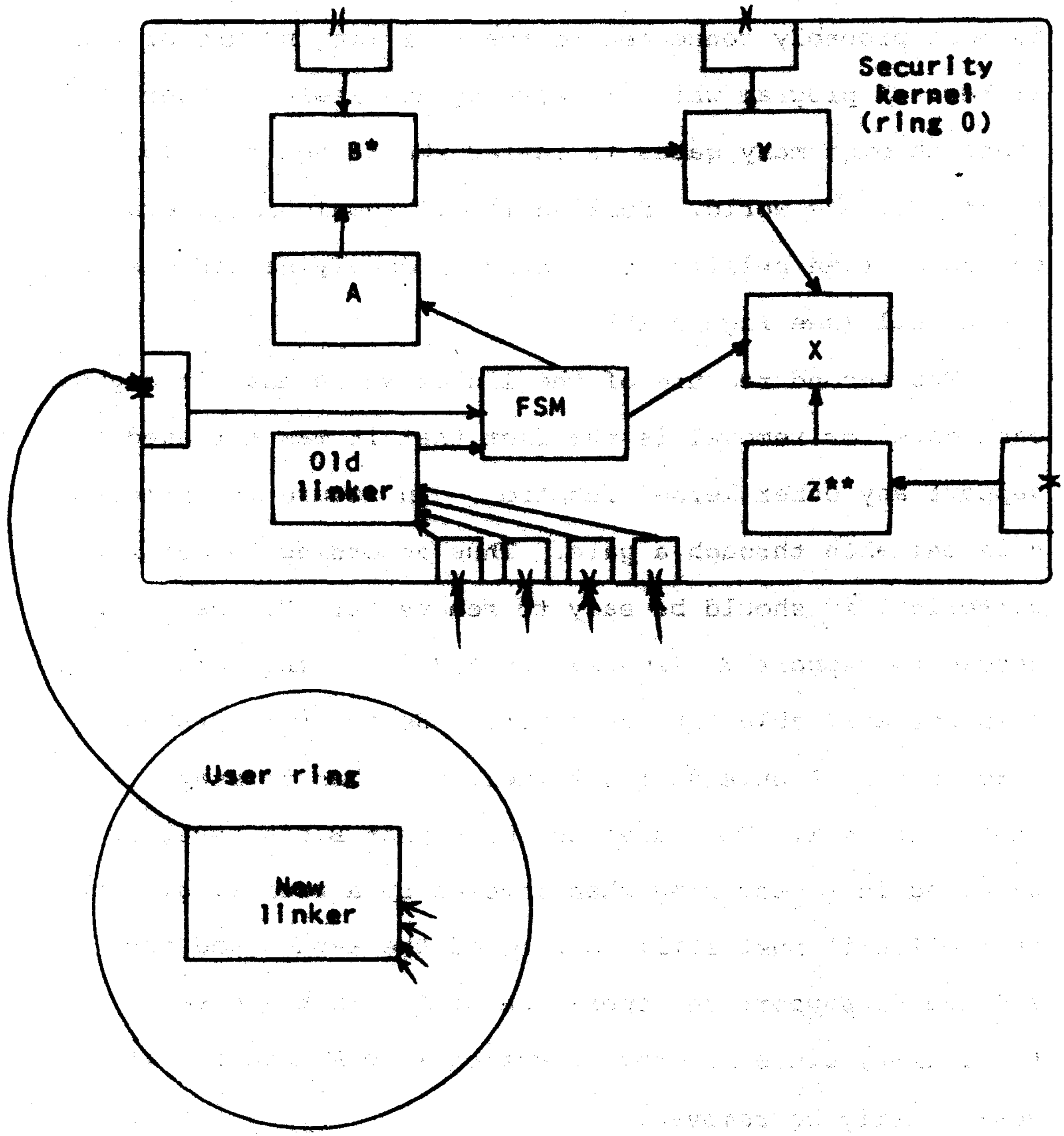
The last major results worth mentioning here are the insights gained about the nature of a kernel. Although the thesis has not provided any definition of what programs belong _inside_ the kernel, it certainly has provided a few insights about what programs can easily be moved _outside_ the kernel. The a posteriori analysis of the linker has revealed a few interesting features which at the same time made the linker an easy to remove program and are a direct result of its user ring nature. We do not suggest in any way that all programs exhibiting the features to be described should or even could be removed from the kernel. We only suggest that such programs are certainly better candidates for removal than others and that any attempt to simplify a kernel should start by examining such programs.

The first feature which made the linker a good candidate for removal is the number of gates which lead to it inside the kernel. As we already suggested, this fact is most probably connected to the user ring nature of the linker. A program which is already available to user rings through many gates is inside the kernel but close to the outside world. Pulling it out should in general be easier than pulling out a program deeply nested inside the kernel (see figure 15).

The second feature of the linker which made it a good candidate for removal is the fact that it was not used to support any other kernel function. In figure 15, program B is callable through a gate. Thus according to our first criterion, it should be easy to remove it. However B is needed to support A (invoked by A) inside the kernel, and A is not available through a gate. Hence it is probably hard to pull A outside the kernel and B has to stay inside as well. This does not mean that B can never be executed in a user ring when invoked by a user ring, but it implies it must still be part of the kernel and thus audited to support the operation of A. In the case of the linker, since no other function like A used it, it could easily be removed.

The third interesting feature of the linker is that

Figure 15: Multics security kernel.



*: B cannot be removed because it is used by A;
**: Z may be hard to remove because it would need a
gate to reach X, which may be hard to provide.

all kernel primitives (e.g. the FSM) it used to invoke from inside ring 0 were already available to user rings through gates. Thus removing it simply moved back the boundary of ring 0 without even creating new gates through it. Instead removing Z from the kernel in figure 15 would require a gate to be added to reach X because X is not yet available in the user rings.

The last three paragraphs have described overall features of a program which make it a good candidate for removal. Of course further functional investigation may reveal that such a program cannot possibly be removed simply because it deals directly with protection and is a proper component of the kernel.

We finally would like to examine the cost of our implementation: how much did the removal of the linker alter the performance of the system? Given that performance and performance evaluation were not among the goals of our thesis, we will not present an exhaustive performance study of the linker. However we have run a few simple performance tests which consisted simply in measuring the time required to snap "average" links. By "average" we mean links of the type most frequently handled by the linker. That is links not going cross-ring and not using any sophisticated features. The measurements were taken in two different cases. First, we measured the time required to snap a link to an object currently mapped in the logical address space. Secondly, we measured the time required to snap a link to an object not currently mapped in the logical address space. Such measurements were carried on for both the old linker and the new linker.

In the first case, the new linker requires 10 more milliseconds than the old linker, which represents an increase of 40 to 60 percent of the total time required by the old linker to snap the link. This fixed increase in time is independent of the amount of processing required to handle the link itself. We attribute it to the fixed overhead involved in signalling the link fault in the faulting ring, invoking security kernel primitives through gates, and requesting the kernel to validate and restore the machine status. All these operations are required for the new linker to operate and were not required or not so complicated with the privileges of the old linker. This increased overhead is the basic price paid by our design.

In the case of the second set of measurements, the new linker requires roughly twice as much time as the old linker does. Such overhead is not a fixed overhead although it contains the fixed overhead of 10 milliseconds. Instead this overhead is relatively proportional to the length of the search for the target object in the file system. In order to speed up the search for and mapping of a target object, it is standard practice on Multics to first look in the logical address space in case the object is already there. The first set of measurements corresponds to this case. Only if the object is not found in the address space is the FSM invoked to search the file system. The reason why this search is roughly twice as long for the new linker as it used to be for the old one is mainly because search rules are now directory treenames instead of directory segment numbers. As we mentioned it earlier, we expected this to yield a non-negligible overhead because translation of a treename to a segment number prior to each directory search is very expensive. Fortunately, when the project of removing name space management from the kernel is finished, we will be able to restore the search rules under their old form and the performance will no more suffer from the overhead described above.

To conclude the discussion of performance, it must be said that clearly some fixed overhead (10 ms) was paid by the new design. However the overhead in the search is a price paid only temporarily. In addition it is believed that the figures presented can be improved. They are the results of very rough measurements; a more careful analysis is clearly needed to identify the bottlenecks in the new linker and try to optimize the code there. Also, when static storage allocation, trap handling and other features will be separated from the linker as recommended, the performance of the linker is likely to increase significantly because it will no more have to check and worry about all such peripheral features. Thus the performance perspective is not as bleak as the above figures seem to suggest.

Summary

This thesis has attempted to open a road towards security kernel simplification by removing the dynamic linker from the security kernel of a computing utility. A second wave aimed at simplification of the kernel is now on its way to remove name space management from the security kernel. No matter how large an effort these two first simplifications will have required, this effort is almost negligible in comparison to what remains to be done. Even when we will have reached the minimal definition of a security kernel, the hardest part of its certification will remain to be worked out: the auditing. There exists so far no formal theory of kernel auditing. While program verification techniques are a first step towards kernel auditing, they are not the panacea. Auditing a kernel is much harder than auditing the sum of its program components because of all hidden interactions between these components.

Yet because of the increasing need for security and reliability of information stored in a computing utility, more powerful and carefully verified protection mechanisms are demanded. Protection of information is not only the fact of defense, census, medical or criminal information systems. It is a vital characteristic required by our society from any information storage system, computers not in the last place. Thus it is worth paying the price of certification to satisfy the fundamental need for true protection.

BIBLIOGRAPHY

(1) R.M. Fano
"The Computing Utility and the Community"
IEEE Int. Conv. Record, Part 12, P30-37, 1967

(2) A.R. Miller
"The Assault on Privacy"
Signet, March 1972

(3) National Bureau of Standards
"Government Looks at Privacy and Security in
Computer Systems"
U.S. Department of Commerce, NBS TN 809, February 1974

(4) A.M. Noll
"The Interactions of Computers and Privacy"
Honeywell Computer Journal, 7, 3, P163-172, 1973

(5) D.B. Parker
"Threats to Computer Systems"
Lawrence Livermore Laboratory Technical Report,
UCRL 13574, March 1973

(6) D.B. Parker, S. Nycum, S.S. Oura
"Computer Abuse"
Stanford Research Institute, November 1973

(7) J.H. Saltzer
"Protection and the Control of Information Sharing
in Multics"
To appear in CACM 17, 7, July 1974

(8) B.W. Lampson
"Protection" in Proc. Fifth Princeton Symposium on Information Sciences and Systems, Princeton University, P437-443, March 1971

(9) G.S. Graham, P.J. Denning
"Protection - Principles and Practice"
Proc. AFIPS 1972 SJCC, 40, AFIPS Press, Montvale, New Jersey, P417-429, 1972

(10) D.H. Vanderbilt
"Controlled Information Sharing in a Computing Utility"
M.I.T. Project MAC, MAC TR-67, 1969

(11) L.J. Rotenberg
"Making Computers Keep Secrets"
M.I.T. Project MAC, MAC TR-115, 1974

(12) J.J. Donovan
"Systems Programming"
McGraw-Hill, Computer Science Series, 1972

(13) IBM
"IBM OS Linkage Editor"
IBM Systems Reference Library, GC28-6538, January 1972

(14) CDC
"Scope 3.4 Workshop Handbook"
CDC 6000/7000 Development Services, 1970

(15) *See note

"Introduction to Multics"

M.I.T. Project MAC, MAC TR-123, 1974

(16)

"Multics Programmers' Manual"

M.I.T. Project MAC, 1972

(17) E.I. Organick

"The Multics System: An Examination of its Structure"

M.I.T. Press, Cambridge, Massachusetts, 1972

(18) R.M. Graham

"Protection in an Information Processing Utility"

CACM 11, 5, P365-369, May 1968

(19) G.J. Popek

"Access Control Models"

Center for Research in Computing Technology,

Harvard University, ESD-TR-106, February 1973

(20) D.E. Bell, L.J. LaPadula

"Secure Computer Systems" (3 volumes)

Mitre Corporation, MTR-2547, 1973

(21) B.W. Lampson

"Dynamic Protection Structures"

Proc. AFIPS 1969 FJCC, 35, AFIPS Press, Montvale,

New Jersey, P27-38, 1969

(22) M.J. Spier
"A Model Implementation for Protective Domains"
Submitted to the International Journal of Computer and Information Sciences, 1973

(23) M.J. Spier, T.N. Hastings, D.N. Cutler
"An Experimental Implementation of the Kernel/Domain Architecture"
ACM Fourth Symposium on Operating Systems Principles, Yorktown Heights, New York, 1973

(24) M.D. Schroeder
"Cooperation of Mutually Suspicious Subsystems in a Computing Utility"
M.I.T. Project MAC, MAC TR-104, 1972

(25) W.A. Wulf et al.
"Hydra: The Kernel of a Multiprocessor Operating System"
Carnegie-Mellon University, Computer Science Department, 1973

*Note:

This manual contains a series of reprints which originally appeared elsewhere:

- F.J. Corbató, J.H. Saltzer, C.T. Clingen
  "Multics - The First Seven Years"
- A. Bensoussan, C.T. Clingen, R.C. Daley
  "The Multics Virtual Memory: Concepts and Design"
- R.C. Daley, J.B. Dennis
  "Virtual Memory, Processes, and Sharing in Multics"
- J.H. Saltzer
  "Protection and the Control of Information Sharing in Multics"
- M.D. Schroeder, J.H. Saltzer
  "A Hardware Architecture for Implementing Protection Rings"
- R.A. Freiburghouse
  "The Multics PL/1 Compiler"
- J.H. Saltzer, J.F. Ossanna
  "Remote Terminal Character Stream Processing in Multics"
- R.J. Feiertag, E.I. Organick
  "The Multics Input/Output System"

Appendix: Gates removed from the Multics security kernel

To illustrate the variety and the number and the complexity of the functions removed from the Multics kernel by the implementation described in Chapter IV, we list here all gates removed from the kernel with their respective description.

- assign_linkage

  allows the user to request the static storage allocator to allocate a given amount of space in the cls of the requesting ring. A pointer to the allocated space is returned;

- fs_search_get_wdir

  allows the user to ask the treename of his current working directory. The working directory is used in the search rules and can be any directory so defined by the user;

- fs_search_set_wdir

  allows the user to define his new working directory;

- get_count_linkage

  allows the user to obtain a pointer to the static storage of a segment given a pointer to and the bitcount of that segment;

- get_defname_:

  is a generalization of get_entry_name for entries not necessarily into executable programs;

- get_entry_name:

  allows the user to find out the name of an entry into a program given a link to that entry;

- get_linkage:

  is essentially the same as get_count_linkage but does not require the bitcount of the segment under concern;

- get_lp:

  allows the user to get a pointer to the static storage of a program in the requesting ring given a pointer to the segment containing the program;

- get_rel_segment:

  allows the user to get a pointer to the definition or the linkage section of a segment given a pointer to the segment;

- get_search_rules:

  allows the user to find out what his current search rules are;

- get_seg_count:

  allows the user to get a pointer to and the bitcount of a segment given the segment name;

- get_segment:

  same as above but doesn't return the bitcount;

- initiate_search_rules:
  allows the user to define new search rules and enable them in the current ring;
- link_force:
  allows the user to force a link to be snapped. This is a "static linking" entry in the dynamic linker;
- make_ptr:
  allows the user to fabricate a pointer (i.e. a link) to an object from scratch, given the symbolic name of the object;
- rest_of_datmk:
  allows the user to grow a data object under a given symbolic name if that object doesn't exist yet. This is a gate into one of the sophisticated feature handler hooked to the linker;
- set_lp:
  allows the user to set the static storage pointer for a given program in the current ring;
- unsnap_service:
  allows the user to undo the work of the linker by unsnapping any link the linker may have snapped in the requesting ring to a given entry.

We hope this exhaustive list of once gates into the linker has convinced the reader of the variety and the complexity of the linker interface. This is one of the reasons why it was very desirable and rewarding to remove it from the kernel. In addition to having to audit 18 gates into the kernel, on the average 4 arguments per gate had to be validated, which increased the complexity and the certification problem even more.

BIBLIOGRAPHIC DATA SHEET

1. Report No.
MAC TR- 132

2.

3. Recipient's Accession No.

4. Title and Subtitle
Removing the Dynamic Linker from the Security Kernel of a Computing Utility

5. Report Date: Issued
June 1974

6.

7. Author(s)
Philippe A. Janson

8. Performing Organization Rept. No. MAC TR- 132

9. Performing Organization Name and Address
PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY:
545 Technology Square, Cambridge, Massachusetts 02139

10. Project/Task/Work Unit No.

11. Contract/Grant No.
N0014-70-A-0362-0006

12. Sponsoring Organization Name and Address
Office of Naval Research
Department of the Navy
Information Systems Program
Arlington, Va 22217

13. Type of Report & Period Covered: Interim Scientific Report

14.

15. Supplementary Notes

16. Abstracts

In order to enforce the security of the information stored in a computing utility, it is necessary to certify the correctness of the protection mechanism. Certification requires that the security kernel of the system be much smaller and simpler than the supervisor of present general purpose operating systems.

This thesis explores one aspect of simplifying the kernel of a system by designing a dynamic linker that runs outside the kernel domain. The linker is designed to run in any user domain of the computing utility. It is shown that it never needs the privileges of the security kernel to properly operate. In particular the thesis demonstrates the ability of the linker to link modules together across domain boundaries, without violating the protection of either domain involved in the operation.

17. Key Words and Document Analysis. 17a. Descriptors

Computing utility
Security kernel
Dynamic linker
Protection
Domains
Certification

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement
Approved for Public Release;
Distribution Unlimited

19. Security Class (This Report)
UNCLASSIFIED

20. Security Class (This Page
UNCLASSIFIED

21. No. of Pages
129

22. Price

THIS FORM MAY BE REPRODUCED