DYNAMIC LINKING AND ENVIRONMENT INITIALIZATION IN A MULTI-DOMAIN PROCESS.

Philippe A. Janson.
Massachusetts Institute of Technology.
Project MAC.

As part of an effort to engineer a security kernel for Multics, the dynamic linker has been removed from the domain of the security kernel. The resulting implementation of the dynamic linking function requires minimal security kernel support and is consistent with the principle of least privilege. In the course of the project, the dynamic linker was found to implement not only a linking function, but also an environment initialization function for executing procedures. This report presents an analysis of dynamic linking and environment initialization in a multi-domain process, isolating three sets of functions requiring different sets of access privileges. A design based on this decomposition of the dynamic linking and environment initialization functions is presented.

Key Words and Phrases: dynamic linking, protection domains, multi-domain proceses, operating system certification, security kernel, least privilege principle.

CR Categories: 4.35.

1. Introduction.

Research reported here was carried on in 1973-1974 in the Computer Systems Research Division of M.I.T. Project MAC. The Division has been concentrating its efforts on a project aimed at producing a certifiably secure operating system for a computing utility. The approach taken was to improve the certifiability of an existing system -the Multics system [1]- rather than to design an entirely new system from scratch. Two tasks were identified as necessary to produce a certifiable system: the definition of a security kernel for that system and the simplification of that security kernel to the point where an individual can easily audit it to establish confidence in its correctness. One of several ways to make a security kernel both better defined and simpler is to make it smaller by removing from it functions that do not belong there. Removing a function from the security kernel of an operating system simply means arranging that the function is never executed in the security kernel domain.

When the Multics security kernel design project was started, dynamic inter-procedure linking [2,9,10,11] seemed to be an example of a function that did not belong in the security kernel of the operating system. Since the modules of the security kernel can be linked together by some static linkage editor prior to being used, the security kernel does not need the help of a dynamic linker to operate. Also, on any single invocation by a user program, a dynamic linker does not need to manipulate data that is shared by multiple domains. Thus, the two primary reasons for including a mechanism in the security kernel are absent. To show that the dynamic linker indeed did not belong in the security kernel, a new linker was designed that duplicated the function of the old Multics supervisor-resident linker, but that executed in the domain of the invoking user procedure. An important part of the functionality maintained by the new design is the ability to link together procedures that execute in different domains.

The analysis of the function of the dynamic linker required to remove it from the security kernel of Multics revealed that it implemented not just a dynamic linking function, but a more complex function including initialization of the environment of executing procedures. More

This report is based on a thesis [3] submitted in May 1974 to the Department of Electrical Engineering of M.I.T. in partial fulfillment of the requirements for the degrees of Master of Science and Electrical Engineer. Research reported here was sponsored in part by Honeywell Information Systems, Inc.; in part by the Air Force Information Systems Technology Applications Office (ISTAO) and the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641, which was monitored by ISTAO contract No. 19628-74-C-0193; and in part by the Harkness Fellowships of the Commonwealth Fund of New York. Author's address:
MIT Project MAC,
545 Technology Square,
Cambridge, MA 02139.

importantly, the analysis revealed that the dynamic linking and environment initialization mechanisms contained several security flaws. In a multi-domain process, dynamic linking and environment initialization mechanisms may need access to data items residing in several different domains. Granting all access privileges simultaneously to a single program -as was done for the original Multics dynamic linker- violates the least privilege principle [6]: it forces that program to execute in a privileged domain, e.g., the security kernel, where it can potentially be exploited as an unauthorized path of access between user domains. The original Multics dynamic linker contained several flaws that actually allowed this path to be exploited.

This report presents a design for dynamic linking and environment initialization that removes the need for any program to have simultaneous access to data residing in several domains, thus removing the need to put the dynamic linker/environment initializer in the security kernel and removing the source of the security flaws.

In the next section of the report, the concepts of dynamic linking, environment initialization and multi-domain processes will be reviewed briefly to establish their meaning. Relevant features of a model system supporting multi-domain processes will be described; this system will be used as an illustrative example in the report.

The third section of the report analyses the impact of multi-domain processes on dynamic linking and environment initialization. The action of a procedure P calling a procedure P′ across a domain boundary will be decomposed into elementary operations. The elementary operations will be grouped into five sets. Three sets are implemented by three separate programs executing in different domains: the dynamic linker, the dynamic domain generator and the dynamic environment initializer. Of these, only the dynamic domain generator cannot be removed from the security kernel. Two more sets of operations are implemented respectively by the hardware call machinery and by the entry sequences inserted by language processors at the entry points of any procedure.

The design proposed in this report has been demonstrated viable: it has been implemented in Multics. The details of the implementation, the quantitative results of the installation and an evaluation of the simplicity and the performance of the new design are given elsewhere [3]. In summary, the size of the Multics security kernel and the size of its interface were decreased by ten percent while the performance of the system was not noticeably affected.

Throughout the report, the reader is assumed to be familiar with some information protection concepts (e.g. gate, domain, security kernel, least privilege principle) as well as with some design features of Multics (e.g. segmented virtual address space, dynamic linking [1,5]).

2. Framework for Discussion.

2.1. Dynamic Linking.

For each external reference appearing in the source code of a procedure, language processors generate a _link_. The link contains the _symbolic name_ corresponding to the external reference in the source code. Linking a procedure P to a procedure P′ means enabling P to call P′ by translating the link between P and P′ from its symbolic form (symbolic name of P′) to a processor interpretable form (address of P′). With dynamic linking, links are translated one at a time, at first use, rather than all together, at load time. This type of design buys flexibility as it avoids having to establish links universally and unconditionally. Dynamic linking consists of determining the address of P′ and using it to link P to P′.

2.2. Multi-domain Processes.

Some systems support only one user code domain per process [4]. In such systems, inter-domain communication is implemented by inter-process messages. In the present report, we consider systems supporting multi-domain processes. In these systems, inter-domain communication is implemented by procedure calls. This does not mean that all procedure calls cross a domain boundary. Some procedure calls are inter-domain (cross-domain), others are intra-domain.

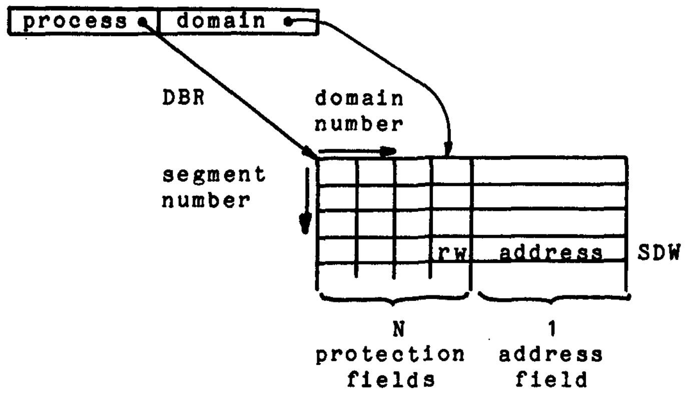
We now provide a short description of the relevant features of a system supporting multi-domain processes. This system will be used in the later discussion to illustrate our topic. This system is very much like Multics but it is a simplified and more general model. It is simplified in that any program module has only one entry point. Thus, the symbolic name space for procedure entry points has only one parameter instead of two as in Multics. It is more general in that each process can go into a finite number (N) of distinct domains which are not subject to any ordering of privileges as are the Multics rings [7]. In this respect, the model is similar to the system described by Schroeder [8]. (1)

Apart from these differences, the system is very similar to Multics as far as dynamic linking and environment initialization are concerned. There is one structured virtual address space per process. The address space of a process is defined by a _descriptor segment_. The descriptor segment is indexed by segment number. Each segment number denotes one entry in the descriptor segment, called a _segment descriptor word_ (SDW), which describes one segment of the address space. An SDW contains the physical address of a segment and N protection fields. Each protection field corresponds to one

(1) Since we assume the unordered domains defined by Schroeder, we also assume the existence of a per process, hardware managed Dynamic Access Stack (DAS): the DAS is used to transmit capabilities for arguments and return gates on a cross-domain call and to reestablish a pending domain invocation on a cross-domain return. Its operation is fundamental to protection but otherwise irrelevant to our topic.

of the N domains and defines the access privileges of that domain to the segment denoted by the SDW. The current process and domain of a processor are defined by its descriptor base register (DBR) as

Figure 1.
Descriptor segment.



represented in figure 1. A security kernel primitive can be invoked to assign a segment number to a segment. When presented with a file system name uniquely identifying a segment, the primitive returns the segment number assigned to the specified segment, assuming access to the segment is allowed. If the segment does not yet exist, a segment number is assigned to it, but the SDW is left empty. The segment and its SDW are created only when actually referenced.

A process can go into N different domains. These N domains belong to that process: no other process can ever use them. The lifetime of a domain is defined to be equal to the lifetime of the process using that domain. There is one symbolic name space per domain. This means that each domain is free to translate any symbolic name contained in a link into a unique file system name according to its own interpretation rules. This allows for identical symbolic names to denote different objects in different domains. Thus, while file system names have the scope of the entire system and segment numbers have the scope of a process, symbolic names have the scope of only one domain in one process.

2.3. Environment of Execution.

Dynamic environment initialization denotes the operations necessary to generate and set up the environment of execution for P′ when control is transferred from P to P′. This consists of creating or retrieving the domain of P′ if the call from P to P′ is a cross-domain call, and creating or retrieving the working storage of P′ in its domain.

During execution, a procedure requires two kinds of working storage: local storage and own storage. (2) To carry on the description of our Multics-like system, let us examine the two kinds of storage. Local storage is allocated in the push-down stack segment for the current domain (3) of execution, and own storage is allocated in the own segment for the current domain of execution. The stack and own segments of a domain are called the domain working storage.

It is often desirable to have executable code be non-writeable. As a measure of self-protection, this prevents a user from accidentally damaging his procedures. As a measure towards recursion and sharing, it allows direct conflict-free access of all existing invocations of a procedure to a single copy of the code of that procedure. On the other hand, the links between a procedure and the segments it references must be writeable by the dynamic linker which translates them and should not be shared across domains and processes because each process has a different address space and each domain may have a different symbolic name space. From the previous two remarks, we deduce that in a system in which access control is enforced on a per segment basis, the code and the links for a procedure cannot be stored in the same segment. In our Multics-like system, associated with each procedure is a non-writeable prototype linkage section appended to the procedure code; in that prototype linkage section, each link contains the symbolic name corresponding to an external reference of the procedure; during execution by a given process in a given domain, the prototype linkage section must be copied into a private linkage section for that domain in that process; the symbolic names may then be unambiguously interpreted in the domain name space and the process address space. Thus, the scope of the private linkage section of a procedure instance and the scope of the own storage of that procedure instance are identical, viz. one domain in one process. Hence the two kinds of storage can be allocated on the own segment -which we therefore rename linkage segment.

Figure 2 helps visualize the system. It summarizes all we need to know to understand the problems of initializing the environment of execution of a procedure in a multi-domain process. DBR defines the current process and domain of the processor, by locating the descriptor segment and specifying one of the N columns of protection fields. IPR is the instruction pointer register containing the segment number of the executing procedure and the offset of the current instruction. SPR is the stack pointer register defining the segment number and offset of the stack frame currently used. LPR is the linkage pointer register defining the segment number and the offset of the linkage section currently used.

3. Problem Discussion.

3.1. Basics of the Design.

Figure 2 summarized the functional description of the system. Figure 3 is just an abstraction of figure 2 in terms of the bindings required by the system to operate. Processor PCR, dedicated to process PCS, in domain D, executes procedure P using the working storage denoted by SPR and LPR.

(2) PL/I equivalent of local and own storage are automatic and internal static storage.

(3) Not to be confused with the DAS for the current process which we said is implemented by hardware.

Figure 2.
Environment of
execution of
a procedure.

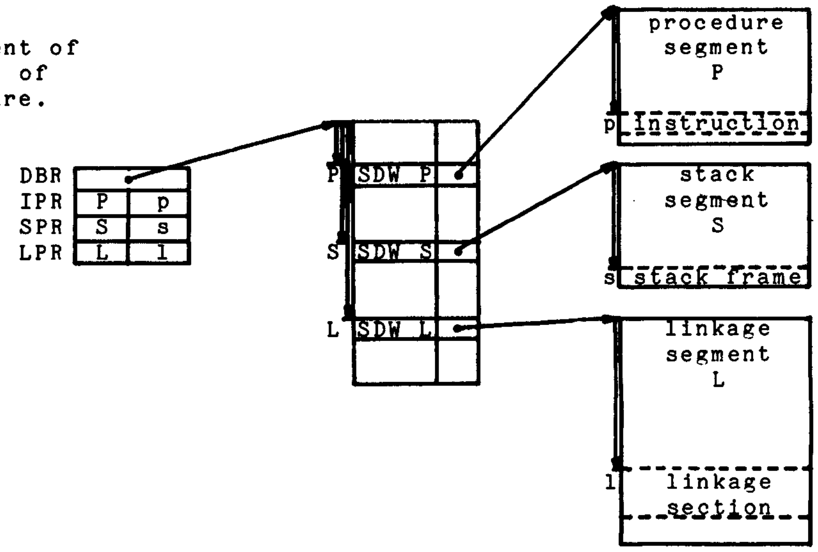


Figure 3.
Processor PCR dedicated to process PCS
in domain D executes procedure P
using stack S and linkage L.

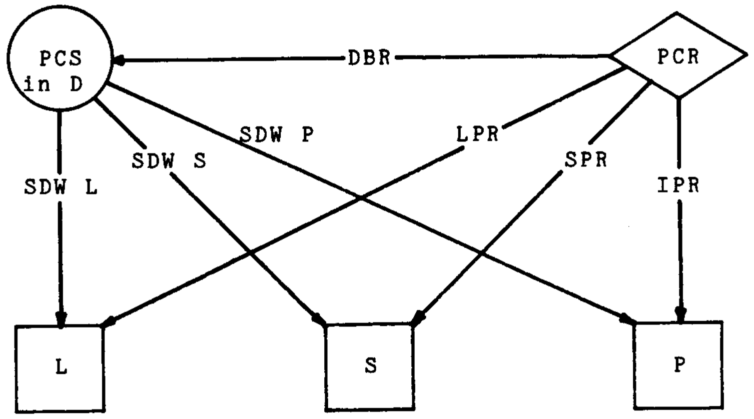
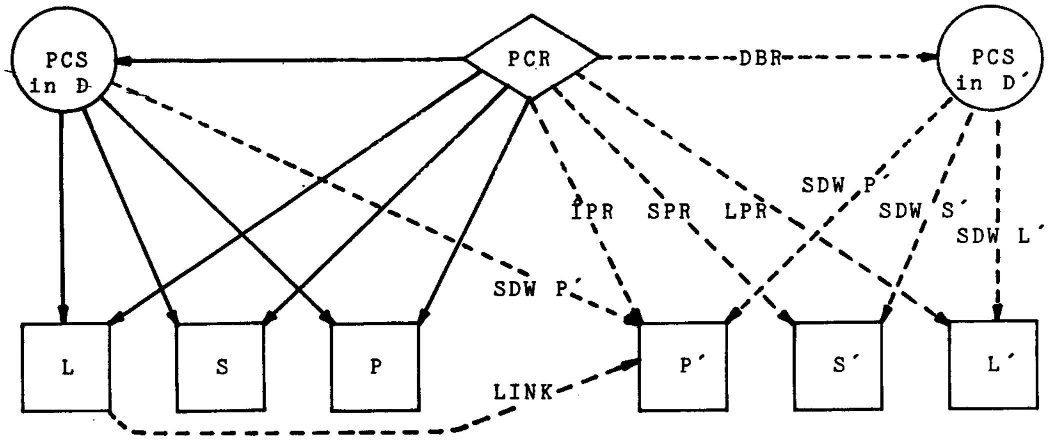


Figure 4.
Bindings involved in a cross-domain call.
Bindings to be established -------



To analyze dynamic linking and environment initialization in a multi-domain process, we consider a concrete situation. Assume procedure P calls procedure P′, P′ is a gate into domain D′ and this is the first time process PCS goes into domain D′. During the call, we will switch to an environment, as described by figure 4, where nothing is set up for execution.

We propose to examine, case by case, how the missing bindings of figure 4 can be established. The binding which is first needed is the link from P to P′ (which is actually recorded in L). When P calls P′ through that link, a processor exception, called a <u>linkage fault</u>, occurs because the link still contains a symbolic name that the addressing hardware cannot utilize. The linkage fault causes the processor state to be saved and control to be transferred to a primitive of the security kernel designed to decode and sort processor exceptions. By analysing the saved processor state, the primitive can deduce that a linkage fault occured in domain D. It will soon be seen that the access privileges available in D are sufficient to resolve the linkage fault in D. It will be seen later that the security kernel can easily find out about the segment number of the dynamic linker. Thus, the security kernel can invoke the dynamic linker in D and give it a copy of the saved processor state to resolve the linkage fault. The dynamic linker does so in three steps: first, using the interpretation rules (4) of D, the dynamic linker translates the symbolic name of P′ into a file system name uniquely identifying P′; second, the dynamic linker invokes the address space management primitive of the security kernel described earlier to obtain the segment number of P′ given its file system name (this causes SDW P′ to be allocated to P′); finally, the dynamic linker uses that segment number and a relative offset of zero as the virtual address of the single entry point (5) of P′ and stores it in the link in place of the symbolic name of P′. After the dynamic linker returns to the security kernel exception handler, the processor state is restored and control returns to the instruction of P that caused the linkage fault. At this point, the LINK and SDW P′ of figure 4 are established. Notice that establishing the LINK requires only privileges to access L and the interpretation rules, to invoke the address space management primitive and to return to the exception handler of the security kernel. The dynamic linker can thus execute in D since the required privileges are available there.

P can now call P′ without causing any linkage fault in the current domain of the current process. As suggested by Schroeder [8], the SDW for a gate must define the domain for which the gate is an entry point. Thus, after the LINK is established, the hardware call machinery can notice that the target of the call, P′, is a gate into another domain. The DBR and IPR bindings of figure 4 can be established accordingly to express that PCR is now dedicated to PCS executing P′ in D′: the domain field of the DBR will be set to D′ and IPR will be set to the value of the LINK, (#P′,0). (6)

Four bindings remain to be established: two SDWs must be created for S′ and L′, and SPR and LPR must be loaded to denote the working storage of P′.

The case of S′ and SPR is relatively easy to solve. We adopt two conventions. First, even though the stack segments for the N domains where a process can go do not exist when that process is created, we impose that N fixed segment numbers be permanently assigned in the address space of any process to the stack segments of the N domains of that process, whether these domains are used or not. Second, in any stack segment, we reserve the first word to save the offset of the next available frame on the stack segment. Thus, since the hardware call machinery finds out about D′ when P calls P′, it can leave in the segment number part of SPR the segment number, #S′, assigned to the stack segment of D′ by virtue of the first convention. Then, a suitable entry sequence in P′, using the second convention and the segment number part of SPR, can load the offset part of SPR from (#S′,0) to denote the next free stack frame on S′. According to the earlier description of the management of the descriptor segment in our Multics-like system, referencing S′ when it does not exist but a segment number is assigned to it causes S′ to be created and SDW S′ to be filled to describe access to S′. Since the stack segments have preassigned conventional segment numbers, the security kernel may be coded to distinguish their creation from the creation of other segments and initialize their first word appropriately. While this mechanism may seem ad hoc, we will see its actual significance later. In summary, establishing SDW S′ and SPR can be done conveniently with the hardware call machinery, a standard procedure entry sequence, two reasonable conventions, and a little help from the security kernel.

Unfortunately, the same is not true for L′ and LPR. If it were just a matter of creating L′ and finding empty space on L′, a method similar to the one used for S′ and SPR would be adequate. However, more operations are required. The scope of a stack frame is one invocation of one procedure in one domain; thus a new stack frame must be allocated and initialized on each procedure call. But the scope of a private linkage section is all invocations of one procedure in one domain; thus a private linkage section for P′ must be allocated in L′ and initialized only the first time P′ is invoked, and its address must be remembered for subsequent invocations. (Recall that the private linkage section is initialized from the prototype linkage section in the procedure segment.) The hardware call machinery and the entry sequence of P′ are not sufficiently powerful tools as they cannot distinguish between the first and subsequent invocations of P′. A program knowing about P′ and L′ is required to initialize the private linkage

(4) The interpretation rules can be stored in D, for instance in the own storage of the dynamic linker since it is the only program using them.

(5) In the real Multics, non-zero offsets are permitted and additional mechanisms are used to determine their value.

(6) The notation (#P′,0) is used to mean the segment number of P′ and an offset of zero.

section of P′ and save its address for later use.

3.2. Unsafe design.

The initialization of L′ and LPR was mishandled in the original implementation in Multics. The creation of L′ as well as the initialization of the private linkage section of P′ were left to the dynamic linker. The dynamic linker performed those tasks while invoked to establish the link between P and P′. It was conjectured that establishing a link between P and P′ would always be followed by calling P′. Thus, the dynamic linker checked for the existence of L′ and the private linkage section of P′ in D′, and created them if they did not exist yet. This task required the dynamic linker to have access to L′ and P′ in D′. But it already has to have access to L in D. Thus, this design forced the dynamic linker to execute in the security kernel domain where it had access to L and L′ at the same time.

In trying to remove the dynamic linker from the security kernel of Multics, it became obvious that it should not have access to L and L′ simultaneously because this violated the least privilege principle and eventually created exploitable flaws in the protection mechanism. The original Multics dynamic linker unnecessarily had too many access privileges and unintentionally misused them. At least two methods existed for P to penetrate domain D′. First, without ever intending to call into D′, P could reference links to each gate into D′ and trigger the linker to initialize the private linkage sections of all these gates. The fact that an action of P in D could cause something to happen in D′ without control is unacceptable and particularily dangerous if there is any chance of L′ overflowing. Second, without ever triggering the linker, P could find out the segment number of P′ by invoking the appropriate security kernel primitive, and call P′ directly using that segment number instead of a symbolic name. As a result, P′ would start executing without its own storage being initialized since the dynamic linker was by-passed by P. A process crash and perhaps some damage could be caused in D′ because of the action of P in D: any initial own variable P′ depends on or external reference it makes would almost certainly be handled incorrectly.

3.3. Correct Design.

We have shown that it is unsafe to let the dynamic linker create L′ and the private linkage section of P′ in D′. We now present an alternative design to generate the SDW L′ and LPR bindings of figure 4.

SDW L′ can be established in exactly the same way as SDW S′ was. We adopt the convention that a second set of N fixed segment numbers be permanently assigned in the address space of any process for the linkage segments of the N domains of that process, whether these domains are used or not. Thus, the hardware call machinery can leave in the segment number part of LPR the segment number, #L′, assigned to the linkage segment of D′ by virtue of the convention. And SDW L′ will be created automatically by the security kernel when L′ is referenced for the first time.

As to the case of the LPR binding, a solution somewhat similar to the one used for SPR can be designed. However, to distinguish the first invocation of P′ in D′ from the others, a new processor exception and a new program to handle it are necessary in addition to a convention. If K is the maximum number of segments the address space of any process can contain, the convention states that words 0 through K of any linkage segment, called the <u>linkage offset table</u>, will be used as follows. The nth word ($0 \leq n \leq K-1$) is used to hold the relative offset of the private linkage section in this domain -if any- of the nth segment of the address space. Word K is used to save the location of the next free area in the linkage segment. The security kernel must of course distinguish the creation of a linkage segment from that of another segment to properly initialize word K. A suitable entry sequence into P′, using the above convention and the segment number parts of IPR and of LPR, can thus load the offset part of LPR from (#L′,#P′) to retrieve the own storage and the private linkage section of P′ in D′. The problem, of course, is that (#L′,#P′) will not be initialized on the first invocation of P′ in D′ since P′ has no private linkage section in D′ yet. To recognize this first invocation, we suggest that an attempt to load (#L′,#P′) into LPR should trigger a processor exception, called an <u>own storage fault</u>, if (#L′,#P′) is empty. The own storage fault causes the processor state to be saved and control to be transferred to the exception handler of the security kernel. Upon identifying an own storage fault, the exception handler can invoke in domain D′, a program called the dynamic environment initializer, and pass it a copy of the saved processor state to resolve the fault. It will soon be seen that all the access privileges the program needs are present in D′. It will also be seen later how the security kernel can find out about the address of the dynamic environment initializer it must invoke. The dynamic environment initializer will notice that the fault occured when trying to load LPR from (#L′,#P′). First, it will generate the private linkage section of P′ in D′ by copying the prototype linkage section of P′ into the free area of L′ located by the address in (#L′,K). Second, it will record the offset of the private linkage section in (#L′,#P′) for future use. Finally, it will update the free area pointer in (#L′,K) to point beyond the end of the private linkage section just appended to L′. Notice that the dynamic environment initializer requires access to only P′ and L′: it can thus execute in D′ where the required privileges are available.

3.4. Initialization.

The last problem which needs to be dealt with in the new design is the initialization of the distributed dynamic linking and environment initialization mechanism. As long as the dynamic linker and the dynamic environment initializer were one single program executing in the security kernel domain, they were initialized as parts of the kernel itself, i.e. some system generation mechanism or bootstrapping procedure was used for the entire security kernel and dynamic linking and environment initialization were not special cases. Now that the dynamic linker and the dynamic environment initializer have been removed from the security kernel and may be invoked at any time in any domain, both programs must be made operational in any domain as soon as that domain becomes used. For the programs to be operational in a domain, two conditions must be fulfilled: first, the security kernel needs to know about their addresses so it can invoke them to resolve linkage and own storage faults; second, they must be prelinked, i.e. they have to have private linkage sections in that domain and all links in the private linkage sections must be translated because neither program can count on the other to bootstrap it.

Since the dynamic linker, the environment initializer and the security kernel modules are vital to any process, they will always have to be mapped into the address space of any process. Thus, they may as well be assigned the same fixed set of addresses in the address space of any process. If so, we can assert that, once translated, the links in the private linkage sections of the dynamic linker and the dynamic environment initializer will be identical in all domains of all processes since they all denote security kernel primitives with fixed addresses. Thus, all it takes to prelink the dynamic linker and the dynamic environment initializer is to submit them to a static linkage editor after system initialization but prior to system operation and to have the linkage editor produce a template translated linkage section for each program. Then, the security kernel can either make the template translated linkage sections public, read-only, or copy them into each linkage segment it creates. If the latter approach is chosen, the operation can perfectly well be performed at the time a linkage segment is created as the creation of a linkage segment coincides exactly in time with the earliest moment when the dynamic linker and the dynamic environment initializer may be needed in a new domain. The security kernel already distinguished the creation of linkage segments from the creation of any other segment, so it can easily do the extra job.

The creation of a linkage segment can also be taken advantage of to enable the security kernel to retrieve the address of the dynamic linker or the environment initializer on a linkage or own storage fault. Words K+1 and K+2, for instance, of each linkage segment can be used to save the addresses of the two programs. Then, not only can the security kernel access the linkage segment to find the desired addresses, but the domain owning the linkage segment can later access it and change the two addresses to indicate its desire to use different programs as its dynamic linker and environment initializer.

It now becomes clear why distinguishing stack and linkage segment creation is not just an ad hoc mechanism: it really corresponds to the physical generation of a domain. We will further refer to the piece of the security kernel that creates and initializes working storage as the dynamic domain generator.

4. Conclusion.

We have analysed dynamic linking and environment initialization in a multi-domain process, on a per binding basis, in a Multics-like system. We will now summarize the main results and abstract them from their Multics context. The statements in this section are based on only two assumptions about the multi-domain system under concern: the scope of own storage is that of the domain which contains it; and the scope of a domain is that of the only process which uses it.

Five pieces of machinery are involved in dynamic linking and environment initialization in a multi-domain system. First, the dynamic linker is responsible for establishing links when linkage faults occur. Second, the hardware call machinery must reload various processor registers when a procedure call occurs; this includes registers denoting the new procedure, the new domain, and the working storage for that domain. Third, if a domain is entered for the first time, the dynamic domain generator must create and initialize the working storage for that domain. Then, the entry sequence of a called procedure is responsible for locating the working storage of that procedure within the working storage of the domain; this will in general require some convention. Finally, the dynamic environment initializer must respond to own storage faults by initializing the own storage part of the working storage of the faulting procedure when its entry sequence cannot find it.

Among the five mechanisms mentioned above, three are implemented by software programs. Each of the programs can and should operate in a different domain. The task of the dynamic linker requires no more privileges than are available in the (linkage) faulting domain or calling domain. The task of the dynamic environment initializer requires no more privileges than are available in the (own storage) faulting domain or called domain. The dynamic domain generator requires the privileges of the security kernel because an empty domain cannot operate without working storage, and in particular, cannot fabricate working storage for itself. Proper initialization of the programs requires that they be prelinked and that the security kernel save their addresses to later invoke them on linkage or own storage faults.

A final comment is now in order. The design proposed in this report has the additional advantage of making inter-domain linking look identical to intra-domain linking from the point of view of the dynamic linker and the environment initializer. This was not the case in the original design of Multics where linking procedures within a domain and across domain boundaries were distinguished.

Acknowledgements.

Many people have contributed ideas to the research described in this report and many people have commented on the successive versions of the report itself. Professor M.D. Schroeder deserves special gratitude for his guidance, his encouragements, his dedication and his interest in the research project, and for his pertinent comments on the report. Professor J. H. Saltzer has made very detailed remarks and highly relevant suggestions on the later versions of the report. The helpful contributions of Dr. D.D. Clark were appreciated in the early phases of the project. Thanks are due B.S. Greenberg, R.K. Kanodia and E.L. Thomas for the ideas they provided during the design of the dynamic linker and for the many hours they spent to help implement it.

References.

1. --- Introduction to Multics. MAC TR-123, MIT Project MAC, 1974.

2. Fabry R.S. Capability-Based Addressing. CACM 17 7, 1974, p 403.

3. Janson P.A. Removing the Dynamic Linker from the Security Kernel of a Computing Utility. MAC TR-132, MIT Project MAC, 1974.

4. Lampson B.W. Protection. ACM SIGOPS Review 8 1, 1974, p 18.

5. Organick E.I. The Multics System: an Examination of its Structure. MIT Press, Cambridge, Mass., 1972.

6. Saltzer J.H., Schroeder M.D. The Protection of Information in Computer Systems. Proc. IEEE 63 9, 1975, p 1278.

7. Schroeder M.D., Saltzer J.H. A Hardware Architecture for Implementing Protection Rings. CACM 15 3, 1972, p 157.

8. Schroeder M.D. Cooperation of Mutually Suspicious Subsystems in a Computing Utility. MAC TR-104, MIT Project MAC, 1972.

9. Shaw A.C. The Logical Design of Operating Systems. Prentice Hall, 1974, p 158.

10. Tsichritzis D.C., Bernstein P.A. Operating Systems. Academic Press, 1974, p 91.

11. Watson R.W. Timesharing System Design Concepts. McGraw Hill, 1970, p 68.