工
391 函
1-0

A STUDY ON CONSTITUTION OF AN ADDRESS SPACE IN A COMPUTER UTILITY

Katsuo Ikeda

Department of Information Science

Kyoto University

1977

A STUDY ON CONSTITUTION OF AN ADDRESS SPACE IN A COMPUTER UTILITY

Katsuo Ikeda

Department of Information Science

Kyoto University

1977

A STUDY ON CONSTITUTION OF AN ADDRESS SPACE IN A COMPUTER UTILITY

by

Katsuo Ikeda

ABSTRACT

This thesis discusses the problem of constitution of an address space in a computer utility. An address space is the space where a programmer expresses his algorithm and runs the calculation when he wants to solve a problem, utilizing a computer system. Thus, its constitution affects greatly on the method of solving a problem, but it has not been discussed much so far.

Chapter 1 is an introductory remark on an address space, gives its definition, and shows its history and its constituent elements.

Chapter 2 indicates the points of issue for constitution of an address space. These points are the followings:

A programmer should devote all his energy to develop an algorithm to solve his problem, without being puzzled by the system configuration, and an algorithm should not be influenced by the configuration of a computer system, by the storage location of information, or by the mechanism of information protection. Further, the space of the "surface" world where a programmer considers his algorithm should be mapped in a natural way into the address space in a computer system.

Information is characterized and managed as a "group" called a segment. The number of segments should not be confined to a small number, and the capacity of a segment should be able to change freely during program execution.

Reference to external segments, including the referring segment itself, should be flexible and easy lest it should impose improper restrictions to the expression of an algorithm or make an algorithm needlessly complicated.

Sharing and protection of information are very important problem.

And mechanisms which support various structures such as the block structure, stack or queue are needed.

ABSTRACT

Chapters 3 to 5 discuss references of information. Reference of information is an essential function of an information processing system. Reference of information is made effective by "linking" it to the target. Chapter 3 describes that the character of an address space, and, as a result of it, the character of a computer system, are basically changed by the method of linking. Chapter 4 discusses typical three methods of linking comparing with each other, and introduces dynamic linking method which is discussed in Chapter 5.

Dynamic linking might be considered the most suitable method to link references in a computer utility. It is able to realize the direct addressing of information wherever the target information is stored. Chapter 5 discusses the mechanism of dynamic linking.

In order to be able to refer to information in a computer utility, sharing of information is required. And as the consequence of information sharing, the demand of information protection arises. Chapter 6 points out the issue of sharing and protection of information, and Chapter 7 discusses mechanisms of information protection.

Protection mechanisms in today's computer system are too simple to be incorporated in a computer utility. This chapter analyses and clarifies the logical structure of the ring protection mechanism, which was first contrived and implemented in Multics, and proposes its extension. Chapters 16 and 17 extend the discussion of protection mechanisms further.

Chapter 8 discusses the required structure of an address space and the mapping mechanism to realize such a space. Chapter 9 completes the discussion of Chapter 8 and gives the conditions to establish an address space in a computer system.

Chapters 10 and 11 discuss concretely the problem of references of information. That is, intra-procedural communications within a procedure - references of the data area and the link areas - are discussed in Chapter 10, and inter-procedural communications - a call, a return, a non-local go to, an interrupt and a fault - are discussed in Chapter 11.

So far, entirely different forms of "calls" have been employed in a computer system, and these have made the logics of the system needlessly very complicated and the understanding of the system difficult. Chapter 11 shows a unified method of inter-procedural communications. This method would make the logics of the system clear and easy to understand.

Chapter 12 revisits the discussion of dynamic linking and develops it in order to remove the linker from the security kernel and to improve the integrity of the system.

Chapter 13 discusses constitution of an address space from the standpoint of the supervisor structure.

Chapter 14 shows other address spaces of sub-processes, which are associated with the main process of a calculation, such as file i/o or stream i/o sub-process, and it also shows some considerations about the databases in a computer network.

Chapter 15 discusses constitution of a programming system which is independent of the system configuration, and shows a very useful application of dynamic linking to a command system.

Chapter 16 extends the discussion of protection mechanisms and shows the requirements of them with multiple capability lists.

Chapter 17 discusses constitution of a memoryless system which would be needed in a computer utility with a variety of competing users with each other.

CONTENTS

ABSTRACT I

CHAPTER 1
INTRODUCTION 1

1. What is an Address Space? 2
2. History of Address Space Management 3
3. Definition of an Address Space 4
4. Constituent Elements of an Address Space 5

CHAPTER 2
POINTS OF ISSUE FOR CONSTITUTION OF AN ADDRESS SPACE 6

CHAPTER 3
REFERENCE OF INFORMATION 13

1. Linking of Symbolic References 13
2. Time of Linking 14
3. Scope of Linking 18

CHAPTER 4
ALGORITHMS OF LINKING 20

1. Pre-Linking Method 20
2. Linking under the Environment of Segmentation 21
3. Algorithm of Dynamic Linking 22

CHAPTER 5
MECHANISM OF DYNAMIC LINKING 25

1. Access to a Link 27
2. Algorithm of the Linker 29
3. Comparison of Object Forms 31

CHAPTER 6
SHARING AND PROTECTION OF INFORMATION 34

1. Policies of the Information Protection 36
2. Mechanism of Information Protection 36
   2.1 Conventional Protection Mechanisms
   2.2 Domain of Protection
3. Constitution of Domains 39
4. Kinds of Access Rights 41
   4.1 Access Rights for Usual Segments
   4.2 Access Rights for Directory Segments
5. Contension of Access - Shared or Exclusive - 44

CONTENTS

5.1 Process Data Segments
5.2 System Data Segments (Common Data Segments)
6. Mechanism of the Domain Switching

CHAPTER 7
THE RING PROTECTION MECHANISM

1. Domain Switching in the Ring Protection Mechanism
2. CPU from the Viewpoint of the Ring Protection Mechanism
3. Comment on Call and Return Instructions
4. Extension of the Ring Protection Mechanism
4.1 Constitution of Mutually Independent Domains
4.2 Extended Domain Switching
5. Clustering of Domains
6. Capability of Ring i State
7. Additional Comments on the Ring Protection Mechanism

CHAPTER 8
CONSTITUTION OF AN ADDRESS SPACE

1. File System
2. Connection of an Address Space and the Information Space
3. Dimension of an Address Space
4. Recursion and Block Structure
5. Mechanism of Segmentation
6. Descriptor Segment and Descriptor Base Register
7. Three-Dimensional Address
8. Support of Lexical Levels

CHAPTER 9
ESTABLISHING AN ADDRESS SPACE IN A COMPUTER SYSTEM

1. Conditions Which Specify an Address Space
1.1 To Show a Process in the System
1.2 The Minimum Information
1.3 Information Required to Execute a Process
1.4 Dispatching a Process
1.5 Initiation of a Process
2. Protection of Pointers and Data Segments
3. Address Space Switching

CHAPTER 10
INTRA-PROCEDURAL COMMUNICATIONS

1. Management of Process Data Segments
1.1 Stack Segment

CONTENTS

1.2 Static Data Segment
1.2.1 In Case that the Pointer is Invariant in Every Ring
1.2.2 In Case the Linker is a Non-Privileged Procedure
2. Constitution of a Linkage Segment 96

CHAPTER 11
INTER-PROCEDURAL COMMUNICATIONS 97

1. Call and Return 97
1.1 Call Instruction
1.2 Return Instruction
2. Elimination of SVC 101
3. Condition Handling 102
4. Non-Local Go To 104
5. Implicit Call 107
6. Invocation of Interrupt and Fault Handlers 110
6.1 Determination of an Interrupt or a Fault Handler
6.2 Status for Handler Execution
7. Interrupt and Fault Table 115
8. Masking Interrupts and Faults 117
9. Processing of Interrupt and Fault Status 117
9.1 Point of Issue for the Saving of the Status
9.2 Point of Issue for the Restoration of the Status
10. Necessary Faults 121
10.1 Faults Caused by Errors in Hardware and Software
10.2 Faults Caused by Arithmetic Operations
10.3 Faults Caused by Address Formation
10.4 Faults Caused by Access Control Functions
10.5 Faults Intended to be Used by a Process
11. Where to Set Fault Conditions 123

CHAPTER 12
MECHANISM OF DYNAMIC LINKING - IMPROVED ALGORITHM - 125

1. Removing the Linker from the Security Kernel 125
2. Processing of the Entry Linker 129
3. Consideration on Performance 132

CHAPTER 13
STRUCTURE OF SUPERVISOR 135

1. Address Space Manager 137
1.1 Dynamic Linker
1.2 Known Segment Manager
1.3 Directory Manager
2. Segment Manager 139
3. Memory Space Manager 140

CONTENTS

4. Physical I/O Subsystem 141
5. Process Manager 142
6. Processor Manager: Get Work 142
7. Interrupt and Fault Interface 143
8. CPU 144

CHAPTER 14
OTHER ADDRESS SPACES RELATED TO A PROCESS 146

1. External World to the Address Space 147
2. Access of Databases in Computer Networks 148

CHAPTER 15
APPLICATIONS OF DYNAMIC LINKING 151

1. To Switch Supervisors for Each Process 151
2. Toward System Independent Processing 152
3. Applications of Dynamic Linking to a Command System 155

CHAPTER 16
PROTECTION WITH MULTIPLE CAPABILITY LISTS 159

1. Constitution of Protection Domains 160
   1.1 Representation of Capability
   1.2 The Ring Protection Mechanism
   1.3 Constitution of Independent Domains
   1.4 Owner's Capability
   1.5 Capability for the Reference to the Arguments
   1.6 Combination with the Ring Protection Mechanism
2. Use of Capability 169
   2.1 Designation of Capability Lists
   2.2 Reference to the Arguments
3. Switching of Domains 177

CHAPTER 17
MEMORYLESS SYSTEM 182

1. Memoryless System 182
2. Gains and Losses in a Computer Utility 183
3. Protection with Multiple Capability Lists 184
   3.1 Processe's Capability
   3.2 Owner's Capability
   3.3 Capability for the Reference to the Arguments
4. Towards Memoryless System 187

CHAPTER 18
CONCLUSION 191

ACKNOWLEDGEMENT 198

CONTENTS

REFERENCES 199
LIST OF PUBLICATIONS 205

CHAPTER 1
INTRODUCTION

In this thesis we are going to discuss the problems of an address space and its constitution in relation to computer system configurations and operating systems.

The history of electronic computer has been the history of pursuing big memory capacity and high processing speed. It is an undeniable fact that requirements of information processing increase so fast that one computing system often has more computations than it can process when it is installed and becomes operable. Average execution time of instructions such as Gibson mix is sometimes used to denote the performance of a computer. Or, comparison of total processing time, which is measured in so-called a bench mark test by processing a job stream prepared beforehand, is often tried to evaluate the performance of a computer system.

What is given to users of a computer by big capacity and by high speed?

Big capacity and high speed are certainly required for a computer system which processes daily routine works efficiently. They are, however, not the primary factors for those who are engaged in research and development works, in which case flexibility and ease in expressing and executing the algorithms to solve their problems are needed. And the constitution of an address space where a programmer expresses and executes his algorithm is an essential problem instead. So far, this problem has hardly been discussed from such a viewpoint, but some of

the constituent elements have partially been discussed from the different angle in the course of the development of big capacity and high speed computer systems. It is significant to discuss this problem putting relevant things together as a total system.

1. What is an Address Space?

A logical address space or a name space implies a logical information space, that is, a collection of programs and data, which one "calculation" refers to or operates. In this case, the method of its physical realization is not concerned in.

A physical address space implies a physical memory space where target information can be physically referred to or operated. It seems that the term "name space" or "address space" has been used since about 1965. Dennis (1965) described that the concept of name space, the set of addresses a process can generate, is contrasted with the memory space, the set of physical memory locations [DNS1]. Donovan (1972) [DON1] described that a collection of programs and data to which one process [RAP1], [VYS1], [DAL2] refers forms an address space. Per Brinch Hansen and Leo J. Cohen didn't use such a term. Richard W. Watson (1970) [WAT1] defined that a logical address space is a set of abstract or logical locations addressed by processes.

The concept of an address space originated from memory allocation in a multi-programming computer system. In such a system, having no connection with the memory location, programs were composed starting at the fixed logical address. Such programs were transformed to be executed at the specific physical address by the

relocation procedure, and then executed there.

Although the constitution of a logical address space is very important because it is the space for consideration and operation in order to manage and solve problems, the constitution of a logical address space has little been discussed in usual computer systems, while the constitution of a physical address space has largely been discussed. In discussing future computer systems or information processing systems, it is necessary to start off with this fundamental problem.

2. History of Address Space Management

In the early days of electronic computers how skillfully he used the lacking memory space was considered the capacity of a programmer. Those days there existed little idea of a logical address space, but only a linear memory space was the subject of management. With the progress of computer architecture, "multi-programming" operation started being used, but only a linear memory space which was partitioned using a base and a bound register was still given to each user. As it was required to move programs in the assigned location at the beginning of execution, the user space always started from the fixed address (typically 0) [GIB1], [COD1], [CRS1], [FOR1], [COM1], [CRT1]. It can be considered that the distinction between a logical address space and a physical address space started at that time.

Those days, however, the space was still one-dimensional, and programmers were busy, to manage the "memory space", planning overlay structures. As the result that multi-programming of high degree was required by time sharing systems [WIL1], [BOB1], technique of

paging and segmentation that realized virtual memory, a memory space of large capacity "imaginarily", came into use, and the distinction between a logical space and a physical space became clearer. But in this case, as the name of virtual memory indicated, it was still a "memory space" and it was doubtful whether it was a logical address space on its original meaning. These were the direct consequences of how to multiplex memory equipment of actually existing capacity and how to make it "imaginarily" larger capacity, but these were scarcely intended to create a logical space for thinking and solving problems.

3. Definition of an Address Space

An address space is the collection of programs and data which one "process" refers to, or as another definition, the locus of execution point and the collection of data which one process refers to. Here a "process" means the substance which executes the calculation to work out a problem or a job.

Above-mentioned data includes all the information in the processor. A process is specified by the following two items:

1. An address space (including all the activation records except the following), and
2. The execution point (a specific item in the activation records).

We will call the collection of information referred to by a process under some conditions as a "working space" of an address space. On the contrary, we can consider about the collection of all the information which "one process" refers to. The collection of

information referred to within an observation period is called a "working set" [DNG3] of an address space.

4. Constituent Elements of an Address Space

In general, each information which constitutes an address space is created separately (at a different time, at a different place). The minimun unit of information reference is a bit, but it is more often referred to in the units of a nibble, a byte, a word, etc..

It is, however, unusual that information constituting an address space is managed by such a unit of reference, and usually information is managed, regarding a group of information as one thing. Hereafter, we call this "group" a segment and give it a name [AND1]. The primary factor to constitute such a "group" is the nature of information called attributes such as:

The producer or the owner and the creation date of information,

Kind of information (procedure or data),

Access privileges of information, etc.

Attributes are all equal to every element in one segment. This paper dosen't treat a problem about catching the meanings of segments at all. A segment is registered and managed in a directory in a file system as described later.

CHAPTER 2
POINTS OF ISSUE FOR CONSTITUTION OF AN ADDRESS SPACE

A person, who is going to work out a problem and to manage data utilizing an information processing system, does not want to obtain knowledge about the structure of the computer system for his own pleasure. However, actually, the more complicated or specific the problem becomes, the more knowledge about the computer system is required in order to master it for his application. Even though this may be unavoidable to a certain degree, it should be natural that one ought to concentrate more energy on considering an algorithm to solve his problem.

It has an immediate connection with reducing the complexity of software and elevating the productivity of software development. It depends upon the constitution of a space for thinking and upon the expression form of an algorithm whether or not a computer system would function effectively to solve problems. When one is solving a problem there might not exist an information space in a definite format. At the same time it may be admitted that a certain collection of data and a certain collection of information which indicate an algorithm do exist in this information space. When one is going to work out a problem using a computer, an information space cannot help taking clearer form. Even if one might not use a computer, this is also the same in case of making others work out the problem. Because it is required that the problem and the algorithm are at least expressed by words and a collection of data is also given in a clear form.

When we are going to make a computer work out a problem, we create a process in the machine and make it execute the necessary operations. That is, a process is an agent of a computer user for his activities in the computer system. Therefore, it is more natural and suitable for the way of thinking that the construction of an address space which a process uses has the same structure as the space of the "surface" world. Also as for the representation of an algorithm, it is natural that the nearest form to the model of consideration should be desired. In this ideal case where these things are fully satisfied, a person who uses such a computer can devote himself to working out his problem, without being puzzled at all about the configuration of the computer. Now, what does it mean that one makes the structure of an address space in a computer system have the same structure as the space of the "surface" world?

First, it has been pointed out that information is characterized and managed as a "group" called a segment. Some databases are fixed while there are various cases where their length or other attributes change with time. Its capacity ranges from greatly big to extremely small.

The requirements which occur here are the followings [RAP1], [COR1], [DNS1], [DNG2]:

1. Number of segments should not be restricted to a small number.
2. The capacity of each segment should be able to range from fairly being big to extremely being small and should freely be able to vary in the course of a computation, without being influenced by the memory capacity.

Second, all the information are not composed at the same time or at the same place. Thus, pertinent segments

which are separately composed constitute an address space. How are these segments referred to? Generally, a reference to a segment is made by name. Then, "linking" is necessary so that an actual computing process may get access to the object items referred to by name and execute operations successfully. Here arise the following questions:

Has any information already existed that corresponds to the name by which one uses to refer to it at the time of programming ?

Has it already existed, when one is actually going to make reference?

Is that reference always uniquely and statically defined?

All the answers to these questions are generally "no". The demand arises here that one must provide a "powerful linking function for flexible reference". This reference ought not to prevent a program from referring to itself so long as it doesn't fall into an endless loop. It is desirable that the realization of recursive expression of algorithms should become directly possible so as not to impose needless restrictions on the logical structure of a program.

Third, a call to a procedure that is one way referring to another "segment" causes problems. Several types of procedure calls which take completely different forms are found in today's computer system. For this reason the control logic of a computer system actually takes a very complicated structure. This is not desirable, and a systematic method is required for the unification of the logic and the structure of programs.

Forth, in case of referring to others' information, problems arise concerning about permission of reference.

One must also solve problems to share information, and moreover, one should be able to take advantage of it in a natural way.

Fifth, problems arise how to support these requirements physically and how to execute computing processes.

Sixth, supports for data structures such as the block structure or the list structure are also required.

Figure 2.1 shows these relations.

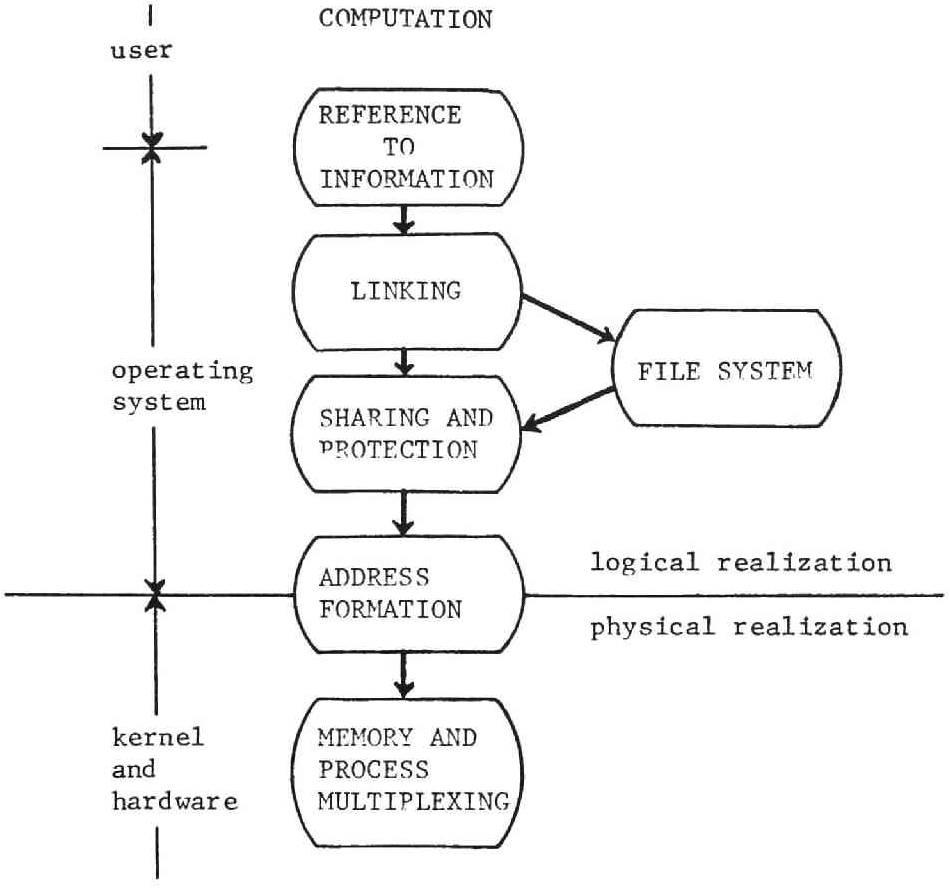


Figure 2.1 Constitution of an address space

In the actual information processing system, these requirements cannot be enough filled up. These days, one

wastes his energy about the format and the location of information, according to the system configuration, to solve problems by a computer, and one is obliged to thread through terribly complicated sequences for the protection of information.

The followings are examples influenced by the system configuration.

Examples influenced by the memory size

A programmer must control the overlay structure of his program.

A programmer must control the overlay structure of his database.

These make a user aware of the working set for economizing the physical storage space.

Examples influenced by the input/output system

The input/output access methods of a database (sequential, random, index sequential, etc)

Examples influenced by the storage location of information

Those which have relation to the input/output system:

Access of a file

Those which have relation to the working set:

Restriction on the access ordering of array elements

Examples influenced by the protection mechanism

A supervisor call:

It is carried out by a SVC instruction, which is much different in its form from the usual procedure call.

Interrupts and faults:

They are calls to the handling procedures in the primary meaning, but they are much

different in its form from the usual procedure call.

These make the configuration of a computer system more complicated and the understanding more difficult. These problems are important not only for an ordinary user but also for those participating in system programming, and they must be improved at any cost to define algorithms clearly and to raise the productivity and the reliability of software.

To sum up, it is required to constitute an address space so as to make the configuration or the logical structure of programs independent from the followings:

A. System configuration, or type of machine
B. Storage location of information (without distinction of main memory or secondary storage)
C. Protection mechanism (SVC, interrupt, or fault)

In another word, addressing which has no relation to the physical location of information should be possible, and perfect yet flexible control should be enforced to information access. And information should be shared in its original form without making any copy for the sake of consistency.

In addition to the above, the reliability and the integrity of software system are a serious problem. In order to improve the integrity and the reliability of software system, it is profitable;

1. To reduce the size of each module,
2. To organize the system in a well structured manner, to unify the structure of procedures - call and return sequences -,
3. To confine damages due to the unnecessary access privileges as small as possible, and to insulate the parts that relate to the protection control from

other parts.

It is often experienced that there is a threshold value of program size above which tremendous time and efforts are needed in accelerating way to complete them due to the accelerated occurence of bugs. Eventually, few big software systems are bug-free.

It is fairly effective to reduce the size of each module for the clarification of the functions and the logics of each module and for the well structured constitution of the system.

Usually, control programs in an operating system are executed in the privileged state, and even a slight error in a control program often results in a fatal condition. Thus, it is necessary to separate and confine the parts which relate to the protection control from other parts, and to give the least privileges for the execution of control programs in an operating system. So far, little attention has been paid to this point of respect, that is, almost all the parts of an operating system have been executed in the privileged state.

It might be expressed that many not well structured programs with possible fatal bugs are currently run with surplus access privileges with the valor of ignorance.

CHAPTER 3
REFERENCE OF INFORMATION

Information is managed regarding a segment as a unit as stated before. Reference to information is carried out by designating one item in a segment. Therefore, two elements are necessary to identify information; a name of a segment, and a name or a location of an item within a segment. Segments are registered as the elements of file systems. Here, let us suppose that the name of a segment identifies one segment uniquely in the file system. (In fact, in order to identify one segment uniquely in the file system, it is necessary to present a path-name. Nevertheless, it is usually very lengthy, and the way which does not require it is wanted. This problem will be treated later.) The constitution of the items which locate the required information within a segment is determined by the logical structure of this segment. What has an effect on the logical structure of a segment is a problem about lexical levels derived from the block structure [RAN1]. The detailed argument is bypassed for further discussion, and for the present we use an identifier or the value of displacement to denote the location of the required item within a segment.

1. Linking of Symbolic References

To get access to information is an essential function in information processing. Usually information

is referred to by a symbol. When the symbol is defined, we can refer to the information by using the value assigned to the symbol. We call it "linking" to make the reference possible, searching or deciding the value of a symbol. If a reference is made to the information within a segment, linking is carried out when this segment is composed. But, it is seldom that we constitute all procedures and databases required in one computation as one segment at a time.

We usually proceed our work with library procedures or public databases which are composed by other persons in combination with segments of our own. More remarkable example is that there might be a case of doing works cooperatively with other persons. In these cases, "external" references from one segment to other segments are often made. In order to realize "external" references and to execute processing, it is necessary

1. to identify the referred segment, and
2. to "link" the referring segment with the target object.

Dynamic characters of an address space are determined by the way of "linking".

2. Time of Linking

Linking may be accomplished at one of the following time:

1. The time of composing a program,
2. The time of language processing (compiling, assembling),
3. The time of linkage editing, and
4. The time of execution.

At the time of composing a program references within

this program are logically linked. Moreover, in the early days of electronic computer, programmers performed linking of external references as well as memory allocation, and even today they are still doing the same thing in special cases. Symbolic references within a segment are resolved at the time of language processing. There even exist such systems that external references are linked at the time of language processing, handling related programs at a time.

It is the most widely used way that resolves external references at the time of linkage editing. All the symbolic references are, indeed, fixed before starting a program.

Linking at the time of execution is performed in the case of running a program interpretively or performing dynamic linking.

In order to link references at the time of composing a program, we must make "determination" at the earliest time. Moreover, as the case may be, a programmer had frequently to do memory allocation as shown in the next example.

```
SIN   EQU   400
        .
        .
      CALL  SIN
        .


SIN   ORG   400
        .
        .
```

In systems which link references at the time of language processing, it is impossible to handle programs

which are written in different programming languages. Such systems do not exist except mini-computers equipped only with an assembler, or systems of early days, or systems for education or training such as WATFOR.

In the linkage editing method it is necessary to link "statically" all the external references. As it is not possible to determine in this stage whether these references are really made or not, both processing time and memory capacity are apt to be wasted. Actually, in a big software system such as, for example, a compiler, there are rather more program modules which are used only under the most particular conditions.

In addition to the above, it is necessary to link previously all the external references to grandchild segments, great-grandchild segments, ... , etc. which have no direct connection with a programmer and about which he doesn't know whether or not they are actually made, the great effort for this is indeed discouraging.

Taking the method of linking at the time when a reference is actually made, neither processing time nor memory capacity are wasted. One of the problems is the trade off between the overhead making links dynamically at the time of references and the loss of processing time and memory space of static linking. But this problem is of little importance.

In research and development works, many cases arise in which it is impossible to make a priori determination, and it is frequently required to set forward works "heuristically". Isn't this rather an essential character of research and development? If it is so, there should exist uncomputable problems in the case of linking references "previously". For example:

```
READ   SUB,ARG
```

```
        .
ANS := SUB(ARG)
        .
        .
```

In this example one intends to read in a function name and an argument and to compute the function value, but it is impossible to determine "previously" what function would be required. For that reason, there exist such kinds of problems as it is essentially required to link at the very instance of the reference. To solve these problems, a method is used which executes a program interpretively. A representative one is the LISP system. It is not, however, adequate to represent all the algorithms in LISP. A big software system is more frequently written in other languages such as PL/I than in LISP. Moreover, interpretive execution of programs is extremely slow and inefficient. Therefore, the static method which establishes links in advance has been adopted.

If one forces a heuristic approach in such a system, it will become an extremely inefficient system. For example, one often carries forward his work in changing his algorithm or parameters little by little. Many of such routines are often only of the order of ten statements. It is the great sacrifice that one must try to link statically the whole programs again, even if only one such small routine is necessary to be modified.

Alternatively, in some systems one selects handling routines by console commands interactively and processes data. But in this case a man always ought to monitor the computation.

Here, the requirement of dynamic linking arises. One of the advantages which one gains from dynamic linking is

that one can still use a system even in the midst of its modification. The number of modules that need pre-linking is limited, and the modules which constitute most user interfaces are able to be supported in the environment of dynamic linking. Therefore, it is not necessary to interrupt the operation of the system for the great part of system modifications, and a new module becomes effective at once if one creates or updates a module. The system of dynamic linking doesn't prevent us from pre-linking by the linkage editing method in the case that higher execution speed which "static" linking attains is required.

3. Scope of Linking

In the above argument, we didn't give any consideration about the storage place of information. If one is able to get off giving any consideration about the storage place of information, the addressing that has no relation to the storage place of information will be realized. Of course, the storage place of information is not determined statically, and one cannot always insure even its existence. Here, we will contrive a method to "link" external references in the scope of all the on-line information within a computer system [COR1]. This will unify the usual memory management and the usual information management and constitute more powerfull, flexible and well suited memory management for our purpose.

The usual information management is called a file system and takes the responsibility for the management of, and the access to, files in secondary storage. Access to a file in secondary storage is carried out by the

function of IOCS which is called the access method, and a programmer has to use input/output statements in his program. This access method depends upon each system and moreover, programming sometimes depends upon devices in the system too, so, in addition to make programming more difficult, it comes to result in programming that is sensitive to the system configuration.

The new addressing method which has no relation to the physical location of information expands memory space which directly becomes the object of CPU operation to secondary storage, and yeilds a new powerful computer utility.

CHAPTER 4
ALGORITHMS OF LINKING

In this chapter we are going to study linking algorithms, comparing the methods for:

Pre-linking system,
Segmentation system, and
Dynamic linking system.

1. Pre-Linking Method

The pre-linking method is the most prevailing and, in fact, almost the sole one used in current computer systems. In linkage edit programs which link external references prior to the execution of a program, linking is accomplished in the following steps:

1. To allocate the memory space to segments.
2. To relocate segments so as to be able to execute correctly in its place.
3. To determine the values of external symbols (registering them in a symbol table), and to modify programs and data so as to be able to refer to external places correctly (linking).

To accomplish this, it is necessary to search for a segment which defines the external symbol referred to in the file system, and determine the value of the symbol within the program being linkage-edited through the process of Step 1. External symbols are registered in an external symbol table for linking. There are two kinds of external symbols:

Segment name and entry name

4. To produce a program in executable form.

This program may be placed directly or may be placed using a separate loader in the main memory.

2. Linking under the Environment of Segmentation

Linking under the environment of segmentation [DNS1], [MCC1] requires the following steps corresponding to each step of Section 1:

1. Regarding allocation of the physical memory area as a separate problem, allocation in this case is to allocate a segment number to a segment. The algorithm is simple enough to assign merely the lowest unused number. But it is necessary to allocate the identical number to the same segment, because if it is treated as a separate segment,
   A. a problem arises in the consistency of information, and
   B. sometimes one segment might be placed in the main memory more than once.

For this reason, names of segments which have already been "known" to a process are registered in the external symbol table. The segment number is used as an index to the segment map, called the descriptor segment, by the address formation mechanism of the CPU. The discussion of the descriptor segment is left below.

2. Relocation is not needed.

This is also one of the distinctive characters of segmentation.

3. It is not necessary to determine the location of an external symbol within the program that is composed by the method of Section 1, but to determine the

location only within a segment.

Therefore, it is not necessary to make an external symbol table in order to include all the names of external segments as in the case of Section 1. It is enough only to use the global external symbol definition table of the object segment as it is.

4. Segments in a program in the executable form where linking has been completed are placed dynamically in the main memory only when they are required (dynamic loading).

The information communion with other processes is possible in this method. At the time a segment is referred to the process must ensure whether or not this segment has already been incore from the active segment table. If it has already been active, all the process has to do is to set the segment descriptor table of this process with the location of the segment found in the active segment table.

A procedure segment must be pure, in this case, and must be linked by a link placed in an impure segment as the segment number is generally different for each process. The data segment for this purpose is composed at the linking time (see Figure 4.1).

3. Algorithm of Dynamic Linking

The algorithm of dynamic linking [VSY1] is as follows:

1. The process assigns a segment number if it is an unknown segment, looking up in the known segment table of the process.
2. Relocation is not required just as in the case of Section 2.

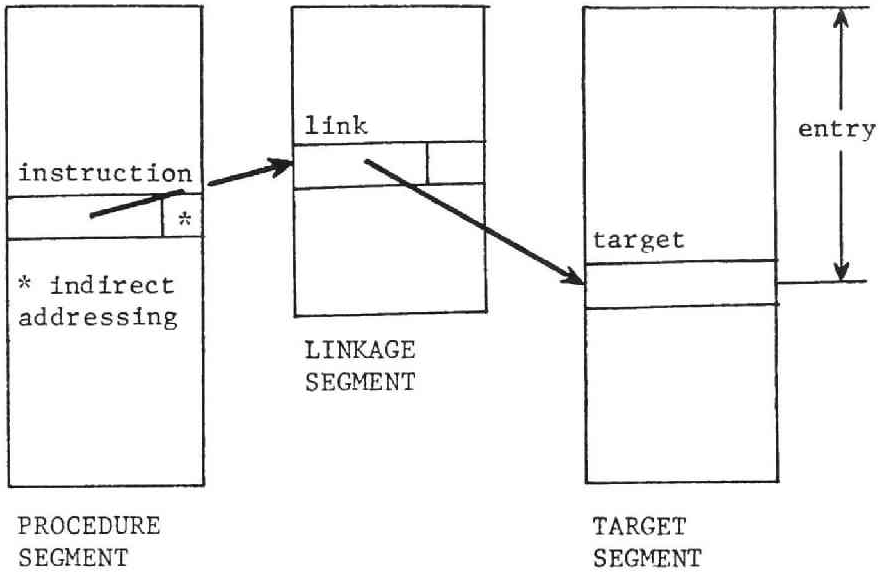


Figure 4.1 Linking to a target. A pure procedure refers to an external segment via a linking pointer (link) placed in a data segment. Address formation is undertaken by indirect addressing or by base register modification. In case that base register address modification is incorporated, the pointer in a link should be loaded to the base register prior to the operation.

3. The necessary things for linking are:
   A. The external symbol table (This is the same as Section 2.)
   B. The place to hold links.

Links should be made in data segments because procedure segments must be pure. In dynamic linking, a link is made when the first reference is done. One can also give an indication using a link whether it is the first reference or not when the process makes a reference with this link. Links which a procedure uses and their locations within this procedure are determined in its language translation stage (see Figure 4.2).

Therefore, when a procedure is referred to (called) for the first time, what the process must do is only to

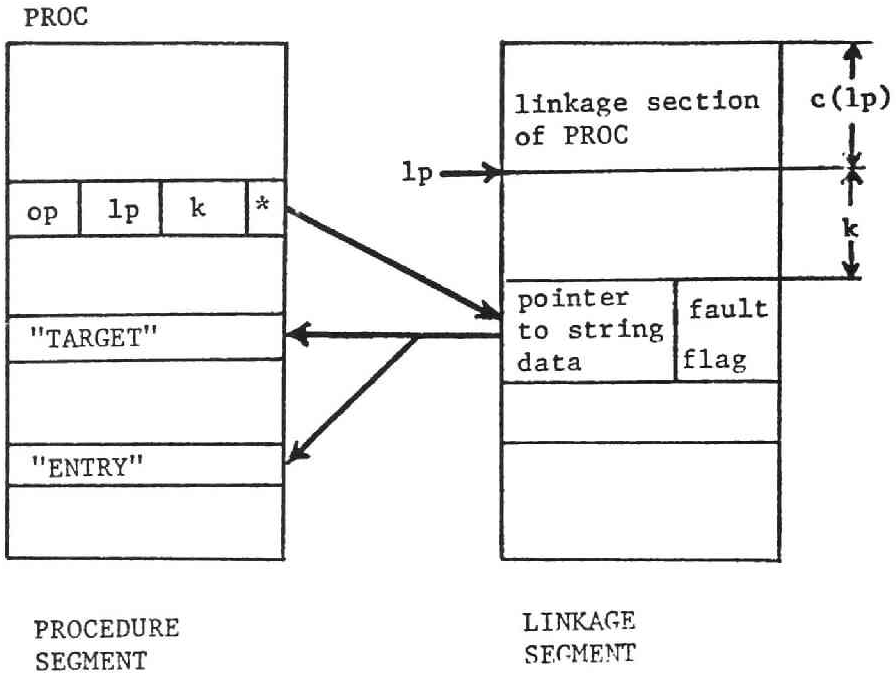


Figure 4.2 Reference to a linkage data. An unsnapped link indicates that it has not yet been linked, and gives the information which is required to identify the target and to complete linking.

copy the pertinent links (linkage section) into a data segment which are used in the procedure and then to link when the procedure is actually executed and an unsnapped link is encountered.

The point of difference between Section 2 and Section 3 is:
In the method of Section 3 a process copies linkage sections, and makes links dynamically while links in the method of Section 2 are made before the program is started.

4. A segment is placed dynamically in the main memory only if it is needed to do so after the link to this segment is snapped and this segment is referred to just as in the case of Section 2.

CHAPTER 5
MECHANISM OF DYNAMIC LINKING

This chapter discusses the mechanism for dynamic linking. The following functions are necessary for dynamic linking [BEN1], [ORG1], [MSP1], [SIM1], [DET1]:

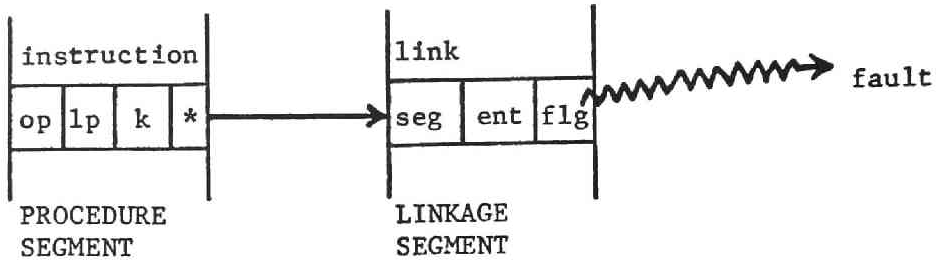
1. To make and hold a link,
2. To find out that this link has not been made yet, and
3. To point to a symbolic name.

1. and 2. must be included in the data which hardware circuitry uses to form the operand address in course of instruction execution. 3. is used in the processing of a linkage fault which will be detected when a referred link has not been snapped yet, and it is referred to by software.

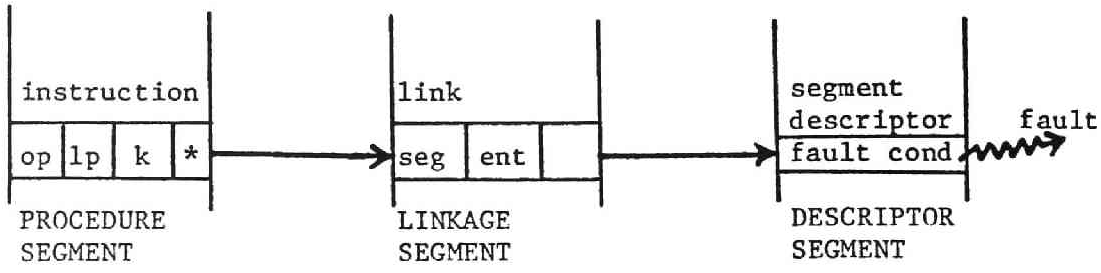
When is it necessary to find out whether or not a link has been snapped? It should be indicated in a link itself whether or not this link has been snapped. Detection of an unsnapped link is possible while hardware refers to a link and forms address (see Figure 5.1).

A. The earliest time is when a link is referred to. It requires the hardware function which immediately occurs a fault condition according to the contents of a link.
B. The latest time is when information of the next level to the link is referred to. It will be detected, for example, as an "exception of the segment number" by the hardware of segmentation.

A. requires a detecting function of the segmentation or



(a) The eariest time (when a link is referred to).



(b) The latest time (when a segment descriptor is referred to).

Figure 5.1 Detection of an unsnapped link. An unsnapped link may be detected in several ways according to the facilities in address mapping mechanism.

paging mechanism. In case of B. it is possible to be included as a small expansion of the segmentation or paging hardware when either has already been provided with.

Further, even if it is not obvious at the time of linking whether the value set to a register will be used as a pointer or merely as an operand for future use, it doesn't cause inconvenience and doesn't need to provide a special mechanism.

Dynamic linking is not possible at all in case that there are no mechanisms for segmentation, paging, or address modification by base registers, etc..

Here is a comment about the usage of instructions. Access to external segments must be made through a base register. It is not admitted to place a description of

"segment_number.displacement" in the operand address part of an instruction directly. This is because:

A. A procedure becomes impure, and
B. There is no room for a pointer to the link definition in an instruction.

1. Access to a Link

Access to a link is carried out by the following steps (refer to Figure 5.1):

1. The location of a link in the linkage section for one procedure segment has been determined at the compile time.
2. The location of a linkage section within a linkage segment, which is the database gathering linkage sections of one process, cannot be determined beforehand. This is because one segment is not always assigned the same segment number as it is given dynamically.
3. The location of a linkage section within a linkage segment is determined when the original template of this linkage section is copied as the initial value.
4. The original template of a linkage section of a segment is copied into a linkage segment when this segment is first referred to, that is, when a segment number is assigned to this segment.
5. Tabulating the location of a linkage section with the segment number, it is very easy to get the pointer to the linkage section (lp) when a procedure segment is entered.
6. The process can refer to the necessary link, using the pointer established in step 5. and the offset established at the compile time.

In a computer which has no facility for indirect addressing, a process must once establish a link in a pointer register and then gain access to the target segment. By doing so, however, the execution speed to refer to the target, at which the pointer register points, or to other targets, which have different offset values at most, is faster than that of indirect addressing (see Figure 5.2).

UNSNAPPED LINK

special seg no.	link no.	offset	pointer to link definition

SNAPPED LINK

segment no.	displacement	pointer to link definition

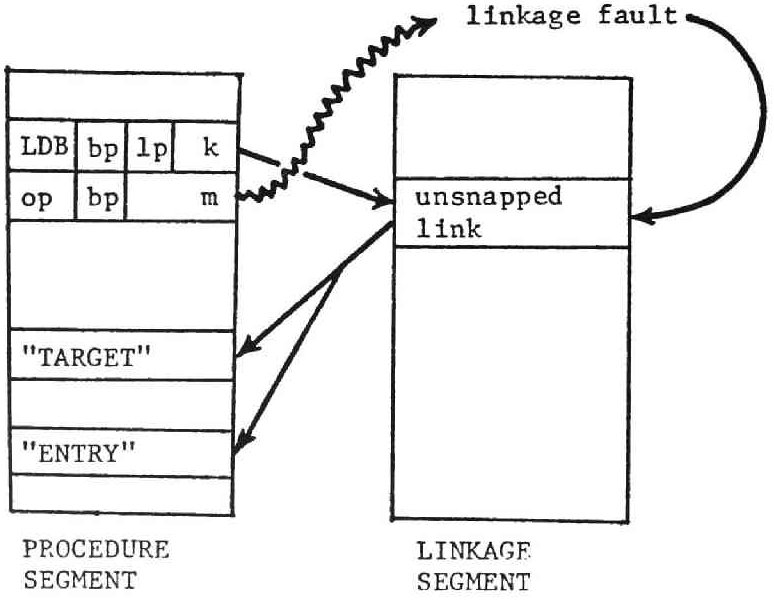


Figure 5.2 Base register and linkage data in case that there is no indirect addressing function. This figure shows a method which may be employed in a system which has no indirect addressing function.

A program can be executed in the same speed as in the linking method which fills up address (this type of linking is the fastest though needs more linking time and makes procedures impure) in case that establishment of a pointer register is finished at a stretch. As a process can modify the value of the pointer by offsets in instructions or by index registers, it needs less links in comparison with an indirect addressing method which doesn't have this facility.

In case of an indirect addressing method, a process needs to make different links for every different value of offsets.

2. Algorithm of the Linker

When a linkage fault occurs, the linker is "called".

The linker gets the target segment name from the link definition, compounds a path-name applying the search rules of the faulting ring, and requires the file system in the kernel to search for the target segment.

If the target segment is found (if necessary, the segment is made active, causing a segment fault), the linker begins to search for the entry name.

As soon as the required link in the faulting ring has been established, the linker's work is finished and the program execution resumes again.

Let's consider the case that a target segment is going to be executed by a call. In this case, the process must get the linkage pointer at first so as to be able to get access to the linkage section, which is the static storage area for this procedure.

The location of the linkage section within the linkage segment (provided in each ring independently) is

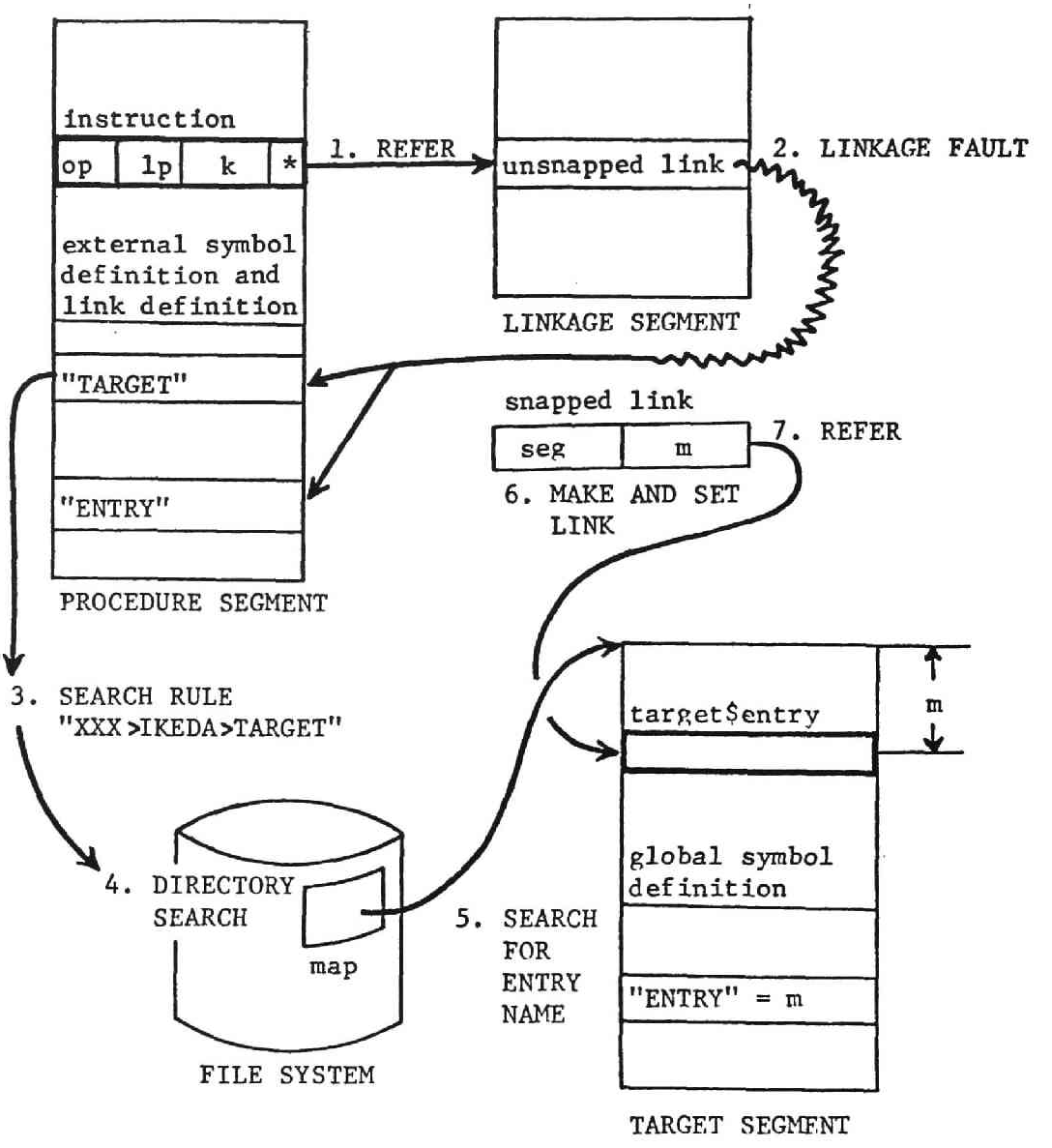


Figure 5.3 Mechanism of dynamic linking

determined from the linkage offset table of the executing ring using the segment number as the index. In case that the linkage section has not been copied yet, the initial value is returned as the value of the linkage pointer, which causes a fault when the process is going to refer

to the linkage section using this pointer [JAN1]. When such a fault occurs, the process copies the template of the linkage section for the first time. To copy a linkage section is not essentially the business of the linker and it is better to separate its management in order to make the logic of the linker clearer (see Figure 5.4).

In a system incorporating dynamic linking it is able to detect that some segments are referred to at the first time or every time. These functions can be used to account the system module usage, and are called a first reference trap and a reference trap respectively. These reference traps can be implemented by preparing trap flags in a link which is used to refer to a segment, and links which are used to refer to the corresponding trap handler.

The information protection affects the processing of the linker, and this problem is left to the later chapter.

3. Comparison of Object Forms

The following table compares the object form for dynamic linking with the one for static linking.

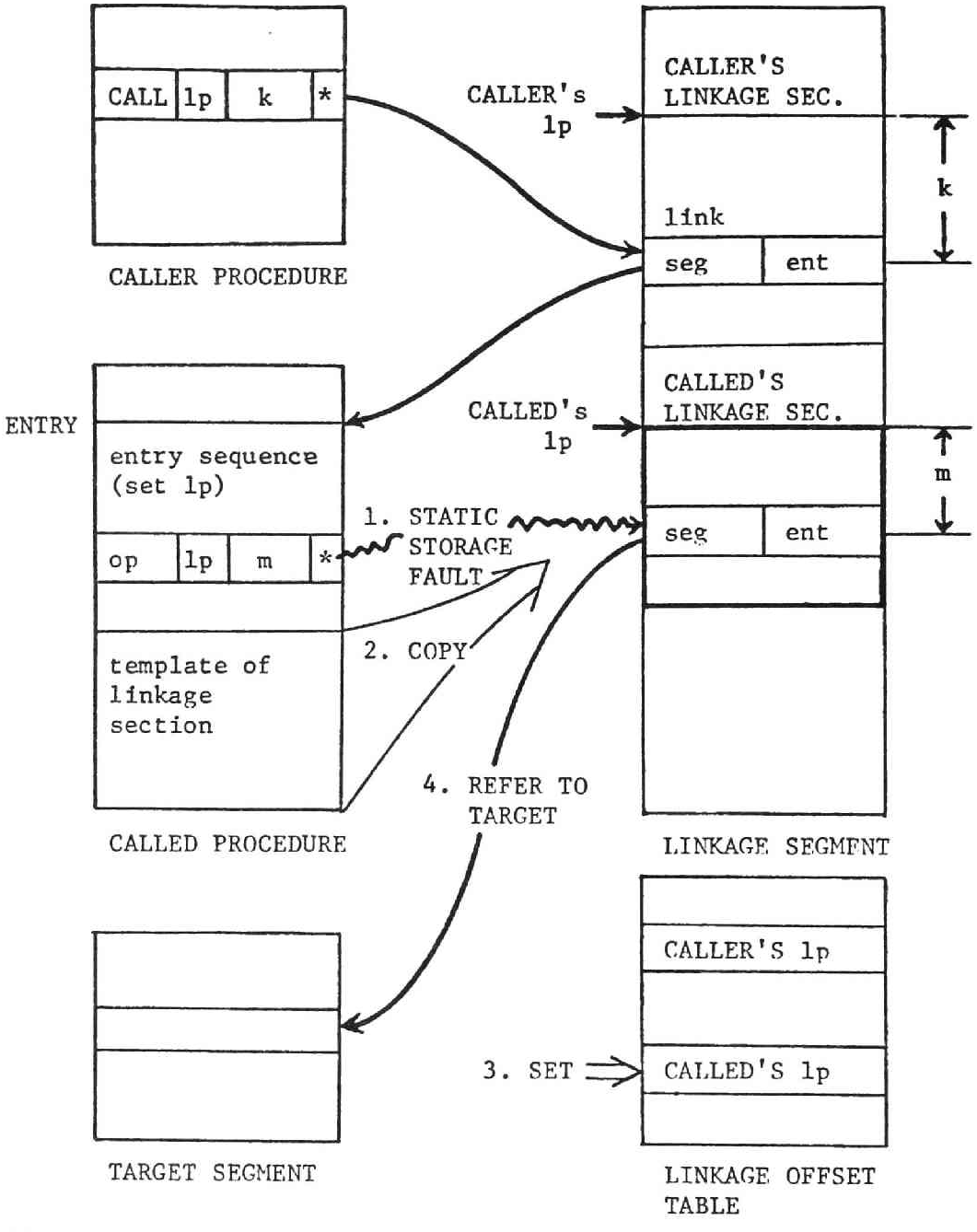


Figure 5.4 Call to a segment. If the linkage section of the called procedure has not been copied in the linkage segment yet, a (no-)static storage fault occurs in the course of the execution of the called procedure. Once the linkage section is copied, the called procedure is executed in the usual manner.

dynamic linking	pre-linking (before)	pre-linking (after)
PURE PROCEDURE	PURE TEXT	PROCEDURE BODY
LINKAGE SECTION (template for links and static variables)	IMPURE TEXT	
STACK SEGMENT (automatic variables)		
GLOBAL SYMBOL DICTIONARY	GLOBAL SYMBOL DICTIONARY	SYMBOL TABLE FOR DEBUGGING
EXTERNAL SYMBOL DICTIONARY (segment names, entry names)	EXTERNAL SYMBOL DICTIONARY	
LINK DEFINITION (type, reference traps)	RELOCATING AND LINKING DIRECTORY	

Table 5.1 Comparison of object forms.

CHAPTER 6
SHARING AND PROTECTION OF INFORMATION

Sharing and protection of information is the area where high degree of interest is paid in modern computer systems.

Significance of information sharing in a computer utility is listed in the followings:

Utilization of common databases and procedures

Execution of one work by more than one process

Cooperative operation

Effective utilization of the space

If more than one process refer to the same segment at the same time to utilize common databases or procedures, the following problems arise [COR1], [GRA1], [DAT1]:

Consistency of information

Protection of information

Access privileges of information

In case that one cooperative work is executed by more than one process, the consistency of information is an important problem. That is, modification to common information must be effective to other processes at the instance when one process modifies such information. This means that it is necessary for each process to use the "original" for its processing and that a process should not have its own "copy" of information. In most multi-programming operating systems in present computers, the same program, typically a language processor such as FORTRAN compiler, is often placed in the main memory

twice or more at the same time and executed concurrently, and run-time library routines are also often copied in each executable binary program at the linkage-edit time. Routines of the elementary functions are "copied" in most programs of numerical analysis. Such copies, indeed, waste the space both in the secondary storage and the main memory as well as processing time.

Then, in order to share information effectively and efficiently, individual copies should not be made but it is required to constitute such a mechanism as to share and refer to the "original" itself. Next problem to be duly considered is information protection. Reference and utilization of information should not be admitted unconditionally. Among information which is placed in a computer utility there may exist such sensitive information that belongs to rivals in business or such information that is opened to the public with charge. Therefore, the protection mechanism of information that can control flexibly references of information as occasion demands becomes indispensable.

The protection of information is also required in order to ensure the reliability and the safety of a computer system as well as the privacy issues. That is, in the environment of a computer utility, the protection of information is necessary in order to confine the propagation of damage caused by a software, hardware, or operation error even in a system used by one person in addition to a safe plan for privacy or interests.

In a shared system, the protection of information is mandatory in order to keep fair operation and maintain the reliability of the system.

1. Policies of the Information Protection

The basic policies of the information protection [SAL3] are the followings:

Fail safe:

Information should not be exposed to risks even by a defect of the protection mechanism. Fail safe is a basic policy of a safety device or a safety mechanism.

Need to know:

Access is permitted only to information which is needed. It is the safety side that access to information which is not required is forbidden. And it is the safety side to confine the scope of circulation of information as small as possible.

The lowest level:

Basically, it is the part of the lowest reliability of the protection mechanism that determines the reliability and safety of a system in the meaning of the information protection. For example, the reliability of data which is acquired from the data having various levels of reliability cannot become higher than the lowest.

The reliability of the result which is computed by the data of reliability i cannot become higher than i.

Therefore, it may be admitted to judge things whose reliability is i by the data of reliability i, but not to judge things of higher reliability than i.

2. Mechanism of Information Protection

The purpose of the mechanism of information protection is to fulfill the requirement mentioned above and to provide a number of users with flexible, but controlled, access to shared information. The design issues [SCH2], [SAL3], [NEE1] of protection mechanism are:

Functional capability,
Economy,
Simplicity, and
Programming generality.

The access control mechanism should have the functional capability to meet the requirement of information protection.

Economy is the well-known principle which applies any aspect of a system. Cost of protection should be proportional to the functional capability actually used. It is needed that users can easily and correctly apply the protection mechanism. Otherwise, the security of his information would be badly impaired from a misuse of the protection mechanism. In another word, the protection mechanism should be simple so that it may be completely understood, and users may have a high degree of confidence that it is safe. With regard to protection mechanism, lack of simplicity often implies lack of security. Thus, simplicity is essential both for safety and economy.

2.1 Conventional Protection Mechanisms

The usual protection mechanism is composed of keys and locks of memory area, a mode switch and a limit register. (Privileged instructions can only be executed in the supervisory mode.) This policy is of "all or nothing". All rights are given if the execution mode once

becomes the supervisory mode, and it is too simple to be suitable for the environment of a computer utility.

Presentation of a pass word, etc. is sometimes adopted as another method. The methods that admit access to information regardless of a user and an execution state cannot accomplish the necessary protection well.

There is a requirement that one wants to control access differently according to the execution state or the user or the combination of both even if the same information is referred to.

2.2 Domain of Protection

The scope where access rights [DAL1] for information are equal to is called a domain. That is, a domain is characterized by a set of access rights, which is the capabilities to refer to the target segments. Elementary access rights are defined for each information. In this case access rights imply the capabilities which are required to refer to the target segment. While a process executes in one domain, it can refer to the information which can be referred to in this domain. Protection of information can be accomplished by organizing domains according to the kinds of access rights and by executing a process in a proper domain according to the requirements. The conditions that determine a domain in which a process executes are the execution status of the past and the present of this process. What must be taken into account to constitute a protection mechanism are:

1. Constitution of domains and
2. A mechanism of the domain switching.

It is shown above that a domain is determined by a set of access rights. Followings are the elementary access

rights given to processes for each information:

read, write, execute, append, directory search, directory change, directory append,
sharable, exclusive (If there exist concurrent processes.)

A process can have the rights to get access to information only when its owner gives this process the necessary access rights. In order that a process may actually refer to information, the necessary access rights must be admitted and, moreover, the process must enter the domain in which these access rights are effective. In another word, it is required that a process can use the access rights effectively.

3. Constitution of Domains

A domain is specified by a set of access rights. If one selects a set of access rights at one's option (but under only conditions which are not contradictory to themselves), one can constitute a general domain. But in this case, it is necessary to manage dynamically the activation records about return points, as stated earlier. These records are the ones to which a process cannot refer directly in the present domain, and it is desirable that hardware manages these pieces of information directly by executing return instructions. For this purpose, a special hardware function is required. In case of simulating the return action by software, as a matter of course, the process must enter into the domain which is prepared for this purpose and manage the return of control.

We should endeavour to make so simple present day protection mechanism as stated earlier more flexible and

more available. Most protection mechanisms which are used in present day computers have two domains whose access rights are prescribed by the following two (see Figure 6.1):

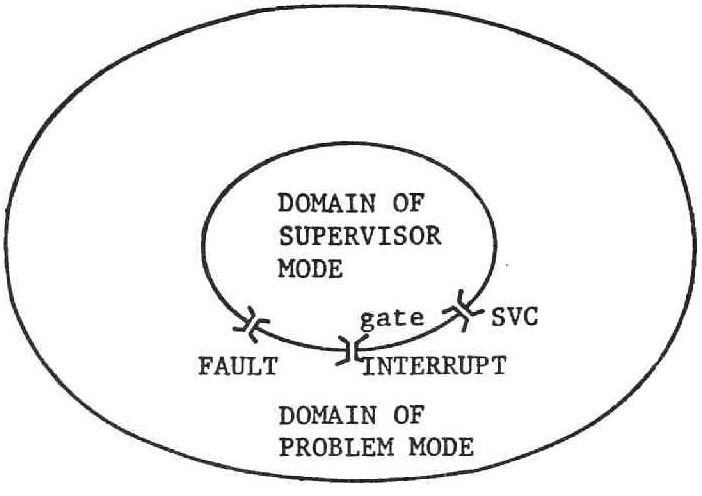


Figure 6.1 Protection domains of conventional protection mechanism.

1. Limited read, write, and execute,
2. Unlimited read, write, and execute.

As access rights of 1. are regarded as a subset of 2., it is considered that domain 2. is included in domain 1.. If one generalizes such an inclusion relation (an ordering relation) a domain is prescribed by access rights which have the inclusion relation. If one constitutes domains in this way, concentric ring domains are constituted. Protection using such ring domains is called the ring protection mechanism. In the ring protection mechanism, information which can be gained access to only in an inner domain is protected from access in outer domains.

It seems to have scarcely been discussed so far about a feature of constituting the ring protection mechanism as the generalization of the inclusion relation (see Figure 6.2).

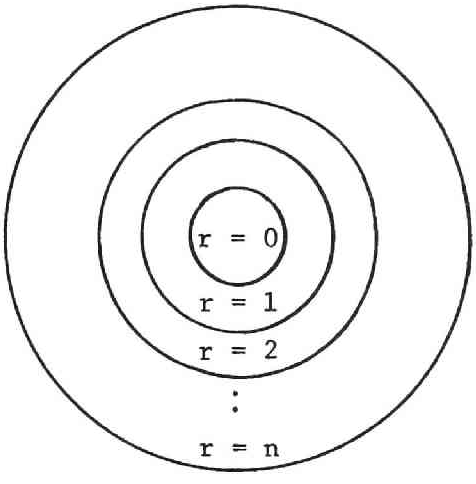


Figure 6.2 Constitution of ring domains.

4. Kinds of Access Rights

This section discusses the kinds of access rights and their effective combinations. It is supposed that a process is executing in a domain which is capable enough to employ its access rights. Here segments are classified either into directory segments or into non-directory segments from the viewpoint of access rights.

4.1 Access Rights for Usual Segments

The elementary access rights for usual segments are: read, write, execute and restricted write (append).

And these are combined with other attributes of the shared and the exclusive. The effective combinations of the access rights are listed below. Although some of them seem invalid or nonsense, they are reasonable and meaningful in different domains for different processes in the system.

read-only (database)
write-only (test paper)
read and write (usual data segment)
append-only (vote, sending message)
execute-only (pure procedure)
read and execute (pure procedure which is open to the public)
read, write and execute (special impure procedure)

Above rights combined with the shared and the exclusive attributes yield the following combinations:

shared read-only (database)
exclusive read-only (common database which is likely to be changed but should not while a process is using it.)
shared write-only (logging file)
exclusive write-only, etc.

4.2 Access Rights for Directory Segments

The followings are the elementary access rights for directory segments:

To create a directory,
To create a directory entry,
To change a directory entry, and
To search for a directory entry.

The hierarchical structure of file system affects every feature of management and control of information. This structure is mainly for the clustering of

information. This structure also affects the management of access control and physical storage space.

Speaking of access control, to be able to search for a directory implies to be able to search for the parent directory which holds the directory entry of the said directory, and so on. (See the later section of file system for the detailed discussion on the structure of file system.) This relation is recursive until the root directory which is the top directory in the file system is reached. Hence, in order to get access to a segment, it is required that all the directories on the path from the root directory to the target directory are accessible and the required access rights are given with regard to this segment.

A list of access rights for the users who are admitted to refer to a segment is prepared for a segment. Access rights are a kind of attributes of a segment, and are placed and managed in the directory entry just as the other attributes. If one could change a list of access rights in a directory entry, whose segment he is not allowed to gain access to, so that he may get access to this segment, and if thereafter he changed the contents of that segment, the security of the file system would be nullified. To avoid such sneak paths the write access to the target segment must be confirmed before the directory entry of this segment is changed. If one has no access rights to a segment, it is the safety side that even the existence of that segment should not be informed.

Therefore, the contents of the directory entries to whose segments he has no access rights should not be informed including the existence of them even if a user has the access rights to search in that directory.

Thus, access to a directory is restricted by the

access rights of the segments registered in this directory. These relations are summed up as follows:

Directory search:

The directory search access and any but "none" access rights to the target segment are needed.

Directory change:

The directory change access and write access rights to the target segment are needed.

Registering a segment:

Only the register (append) access rights are needed. No further access rights are needed because he possesses the object segment of his own.

In a hierarchical file system a unified control over the segments by the manager of the upper level could be enforced systematically. This control includes such functions as the file space management and the access right management. For example, if access rights to a directory for a usual user are confined only to the directory search and register access, he can create and register new segments but cannot delete his own segments, because he is not allowed to change his directory. This is the same situation as employers and employees in a factory. Employees produce their products but are never permitted to destroy products, while employers have all rights to their products.

5. Contention of Access - Shared or Exclusive -

When shared-write access is permitted for some data segments, it must be carefully controlled so that the consistency of information might not be impaired. This section discusses which data segments need to be paid

attention.

5.1 Process Data Segments

Basically, there is no contention of access to processe's own data segments except the one between incremental dump operation of the file salvager and update action of the process. However, if the system enforces such logging rules that the record of the update time in a directory entry is set "after" update action and the record of the dump time is set "before" dump action, it is ensured that incremental dump will be taken once more again and this cycle continues until this data segment becomes quiescent, that is, there would be a chance that no update action is taken while dump action is going on.

5.2 System Data Segments (Common Data Segments)

Access contention is caused to the system data segments which are shared among processes. Appropriate contention control is needed for the data segments for which shared write access is permitted. This contention control mechanism consists of a lock or a semaphore, for which hardware circuitry which controls access exclusively is needed. When a process wants to get access to such a data segment, it is required to lock this segment. If it is locked successfully, the process can operate on this segment. If locking fails, the process should wait to connect this resource, connecting a wait control block to this segment.

6. Mechanism of the Domain Switching

The domain switching, that is, entering a different domain implies that access rights change - new rights which have not been permitted until then are given, or rights which have been permitted are lost -, hence the domain switching should be controlled with enough care.

The domain switching is carried out when control is transferred to a procedure which requires to be executed in a different domain.

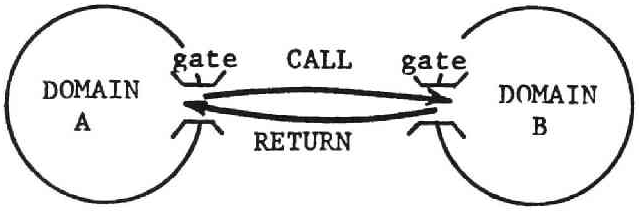


Figure 6.3 Switching of domains.

Clearly, the domain switching should be confined only under the limited conditions. Otherwise, effective control becomes difficult. Things which can impose conditions when the domain switching occurs are as the followings:

Entry point,
Exit,
Return point (entrance to the original domain),
Instruction to transfer control,
Pointer, and
Argument.

Then, what conditions can be imposed for each item?

Entry point:

The significance of restricting an entrance (gate)

is self-evident. A process can enter the domain only at the entrance for which it has been declared previously that it may be used at the time of the domain switching. Also it is possible to set limits to the state of a process which can use one entrance. Or it is possible to require to show a cryptograph at the time of entrance or to execute a particular entry sequence.

Exit:

It is not practical to set limits to the exit from where a process goes out to another domain. Rather, it is more practical to set limits to the place where a process enters a domain as mentioned above. But in case of a return for a call to a procedure, an exit is a return point which will be stated below.

Return point:

A return point is also one of the entrances to one domain. The difference between an entrance and a return point is that an entrance is declared statically at the time of composing a program while a return point is declared dynamically at the time of a call to a procedure. Moreover, in order to manage return points dynamically, a stack is needed if one doesn't set any limit to a call to a procedure including recursion.

Instruction to transfer control:

Various methods of a call to a procedure and a return from it are possible. In order to simplify and validate the logic of the protection mechanism, it is practical to set limits to the kind of instructions which relate to the protection mechanism.

Argument:

It is not desirable to set limits to arguments, as it affects the program logic. However, limitations based on access control sometimes happen to be imposed.

The domain switching mechanism consists of a register which shows the current domain, domain indicators and mode flags in address pointers and segment descriptors, and a determination logic.

In the simplest case of two layered domains, domain switching mechanism can be constituted in the following way:

1. Prepare a flag to denote the mode of execution in a segment descriptor, which is set only by a privileged instruction.
2. While a procedure whose mode flag is on is executed, the execution mode is set to the privileged mode.
3. While a procedure whose mode flag is off is executed, the execution mode is set to the non-privileged mode.
4. Inhibit invocation of a non-privileged procedure from a privileged procedure.

The mode flag in a segment descriptor is set only in the privileged mode, thus, there is no fear that this mode is set unduly by a usual user program, and a privileged procedure which has been invoked by a SVC so far can be called by a usual call instruction.

A more elaborate example of a domain switching mechanism is found in the ring protection mechanism.

CHAPTER 7
THE RING PROTECTION MECHANISM

In this chapter constitution of domains of the ring protection mechanism described in the previous chapter is explained more concretely.

The ring protection mechanism is contrived and first implemented in Multics. The first version was implemented by software and became operable in 1969. The second version is implemented by hardware and currently being used [GRA1], [SAL3], [SCH3]. The ring protection mechanism is also adopted in several systems such as HITAC 8800 and ACOS 700 [MOT1], [ACOS].

The purpose of the discussion in this chapter is to make the logical structure of this mechanism clear and to extend it a little in order to augment its applicability.

In the ring protection mechanism if some access is admitted in some domain, this access is also admitted in inner domains. In order to define domains in the ring protection mechanism, it is sufficient to define the kinds of admitted access and the largest ring numbers in which these references are admitted for each segment. The kinds of access which are commonly considered are:

read, write, execute, directory search, and directory change.

To control these kinds of access, flags of five bits and five ring numbers are required for each segment. And, in addition to these, one must prescribe access control information to every user for each segment. Therefore, we must devise a method to prescribe access rights with less

information. Here rings are numbered from 0 to some maximum, say 7, and the lower the ring number is, the greater the access capabilities are.

First, we will consider the access control information for non-directory segments. We may suppose $r_1 \leq r_2$ where r_1 and r_2 are the largest ring numbers in whose ring a process can write and read respectively. Moreover, it doesn't cause inconvenience practically that the largest ring number in whose ring a process can execute is fixed to be equal to r_2. And, if one considers the condition of "need to know", the smallest ring number may be set to be equal to r_1 because a process can write, read and execute a segment in r_1 but cannot in r where $r > r_1$.

Moreover, let us assume that the largest ring number in whose ring a process can call a segment is equal to r_3 and one can prescribe access rights by three access flags (r, w, ex) and three ring numbers (r_1, r_2, r_3) for a non-directory segment.

A user cannot get access to a directory segment directly, and he must ask the supervisor to search for, create and modify a directory segment. In this case, appropriate access control can be accomplished, taking the caller's ring number as the validation level to refer to a directory segment.

Directory search corresponds to read-access, and

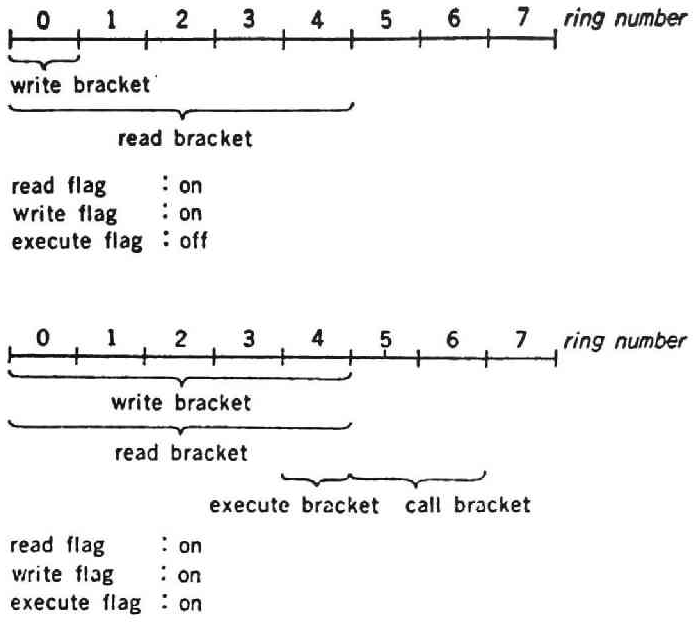


Figure 7.1 Examples of access flags and ring brackets. Bracket means the range where some access is permitted.

directory create- and modify-access correspond to write-access. There is no action which corresponds to execute-access. Thus, one can prescribe access rights by two access flags (r, w) and two ring numbers of the requester (r_1, r_2) for a directory segment.

1. Domain Switching in the Ring Protection Mechanism

There are the following three cases of the domain switching in the ring protection mechanism:

1. Move to an inner ring
2. Stay within the same ring
3. Move to an outer ring

As a move to an inner ring of 1. increases access rights of a process, it must be carefully controlled. In order

to do this, there is a problem of establishing entry points.

There is no problem from the viewpoint of protection as for stay within the same ring of 2..

As a move to an outer ring of 3. decreases access rights, there is no problem to be managed by the access control mechanism, except problems about program logic, because a process which is executing in an inner ring has a right to move to an optional outer ring.

Transfer of control is caused by the following three instructions:

A. Call to a procedure
B. Return from a procedure
C. Transfer other than call and return

It is not taken into account here that the domain switching is required within the same procedure, as the attributes are the same for the information within one segment. And execution of a call to and a return from a procedure is restricted to the call instruction and the return instruction respectively. Therefore, it is necessary to take account only of the following four cases shown in Table 7.1 from the viewpoint of protection.

direct.	inward	outward
call	INWARD CALL	OUTWARD CALL
return	INWARD RETURN	OUTWARD RETURN

Table 7.1 Ring-crossing call and return.

These four cases are divided into the following two pairs:

1. Inward call and consequent outward return, and
2. Outward call and consequent inward return.

In case of an inward call of 1. an entry is used which is fixed statically at the compile time of a program.

There doesn't exist any problem about an outward return. It is guaranteed that arguments accompanied by a procedure call which the caller can gain access to can also be referred to by the called procedure.

An inward call corresponds to the case that a procedure requires processing of higher ability which this procedure can't manage by itself to a higher authoritative procedure.

There is a problem of establishing a return point for an outward call of 2.. That is, one must manage an "entry point" for an inward return. As this return entry needs dynamic management, a return stack is necessary. Moreover, there is a possibility that the caller executes a call accompanying such variables that the called procedure cannot gain access to. As this is the case that a high authoritative procedure requires a low authoritative procedure to manage a problem, it is originally unreasonable that the caller uses the results of the processing as it is. As the reliability of the results of processing is the lowest one which occurs during its processing, which is stated in the section of the protection policy, such an algorithm of a procedure is often incorrect. Of course, it will not have any trouble if a caller never uses the results of processing but just lets it "process".

As the ring protection mechanism does not constitute domains which are mutually exclusive but constitutes ones which have the inclusion relation, mutually suspicious

information cannot be protected properly. In such a case, it is necessary to compose domains which are completely separate each other.

2. CPU from the Viewpoint of the Ring Protection Mechanism

A CPU is a kind of hardware which interprets and executes instructions, that is, an instruction is a pure procedure implemented by a logical circuitry or by a microprogram for which only execution is permitted. There are two kinds of instructions, some are usual instructions and the others are privileged instructions.

A usual instruction is an execute-only utility routine whose ring bracket is (0, 7, 7).

By contrast, a privileged instruction affects the protection status of the system and is only executable in the qualified domain where the protection status is controlled. A privileged instruction is also an execute-only routine whose ring bracket is (0, 0, 0) and which is permitted to execute only in the supervisory mode.

Typically, a CPU has several registers which are used to hold control information to control over instruction execution, and intermediate results of operations. They are:

1. Registers to hold operands of operations
   Data which are hold in these registers belong to data segments of a process by nature. The stack pointer and the linkage pointer are also included in these data. The ring brackets of these data are considered to be (n, n, n) where n is the executing ring number.

2. Registers to Control the Execution Status

2.1 Registers to Control Instruction Execution

The ring brackets of these data are (n, n, n) where n is the executing ring number.

2.2 Registers to Control the Protection Status and Address Space

Data which are hold in these registers belong by nature to the data segments of the supervisor, and the ring brackets of these data are (0, 0, 0)

3. Comment on Call and Return Instructions

Care must be taken as to saving or restoring the data in the CPU when an interruption or a procedure call and a return are taken place.

When a call or an interruption occurs, all data held in registers in the CPU are copied into the stack frame of the caller's procedure or the interrupted procedure.

When a return from the called procedure or the interrupt handler is taken place, the ring number should be the maximum of the executing ring and the ring of the stack frame, and the data which could be touched at this ring number should only be restored. (See Chapter 11 for further discussion.)

By doing so, there would be no fear that the protection mechanism of the system would be impaired, and we could unify the save and restore sequences of the status in the system.

4. Extension of the Ring Protection Mechanism

The ring protection mechanism mentioned above is the generalization of the simple two-layer mechanism which has the problem state and the supervisory state. It is rather easily realized with simple logic circuitry, and has an effect on wide applications.

However, two subsystems which are mutually suspicious cannot be protected by the ring protection mechanism because the ring protection mechanism implies inclusion while it is necessary to realize exclusion to protect two mutually suspicious subsystems [SCH1], [SAL3]. Schroeder (1972) [SCH2] proposed a scheme of a general protection mechanism which realizes mutually exclusive domains but it is much complicated and difficult to implement effectively in a usual computer system.

In this section we will discuss a method to realize mutually independent domains extending the ring protection mechanism. The points of issue are the followings:

1. What types of domains are necessary?
2. How to make domains mutually independent each other?
3. How to enter another domain?
4. Where to include or place segments and how to do so?

4.1 Constitution of Mutually Independent Domains

This section discusses a method to extend the ring protection mechanism in order to augment its applicability.

The ring protection mechanism is realized and controlled by specifing the ring brackets and the access flags in the descriptor segment. Domains which are independent each other could be realized by providing each domain with a descriptor segment.

Common system modules, mainly supervisory segments and utility segments, are shared among processes, which are all placed in commonly accessible inner rings. Conceptual diagram of such domains would be sketched as the following figure.

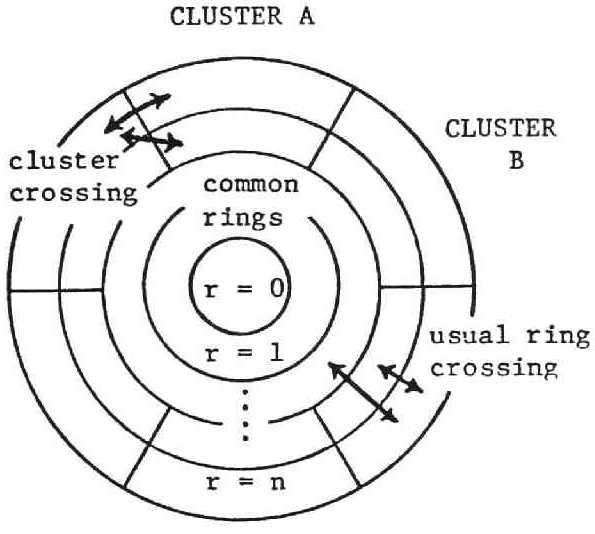


Figure 7.2 Constitution of ring domains which can separate mutually suspicious subsystems.

In this figure, domains are clustered to form sub-spaces each of which constitutes concentric domains of the ring protection mechanism. A cluster of domains might correspond to a certain subsystem. Such a configuration has in fact already existed when we look at a multi-processing system from the system-wide angle instead of the process-wide. In this case, however, the switching of domains from one address space to another is never permitted. Here, it is necessary to separate mutually suspicious pieces of information, placing them in independent (cluster of) domains, and also necessary to enter an appropriate domain in a certain cluster when

they are needed. So the protection mechanism which we are discussing has essentially different requirements from the one in a multi-processing system.

4.2 Extended Domain Switching

As mentioned above, cluster of ring domains which are mutually independent may be realized by creating independent address spaces in a system. What is required is a mechanism to switch clusters. And this is nothing but the switching mechanism of address spaces.

The switching of domains is required when a "domain crossing" call and a consequent return are executed. Calls in this case include hardware implemented calls, that is, interrupts and faults.

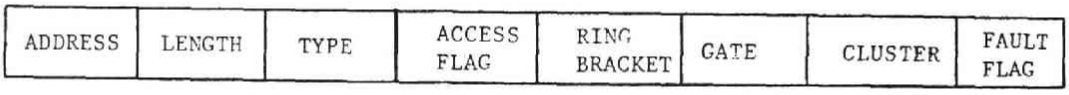


Figure 7.3 Segment descriptor for the cluster switching. The cluster field is newly added and is used to determine the target clusters when the cluster switching that is indicated by the fault field is needed. The type field, the access flag field, the ring field, and the gate field are used in the ring protection mechanism.

There are two kinds of the domain switching, one switches domains within a cluster, and the other switches domains crossing the wall of clusters. If it is necessary to switch clusters of domains, a fault will notify this and the supervisor can change the setting of the descriptor base register which specifies the selected cluster of domains. The condition that needs to switch clusters of domains when a call or a return occurs could be designated by specifying the identifier of a target cluster in a segment descriptor (see Figure 7.3). (The

discussion on the segment descriptor, the descriptor segment, and the descriptor base register are left behind.)

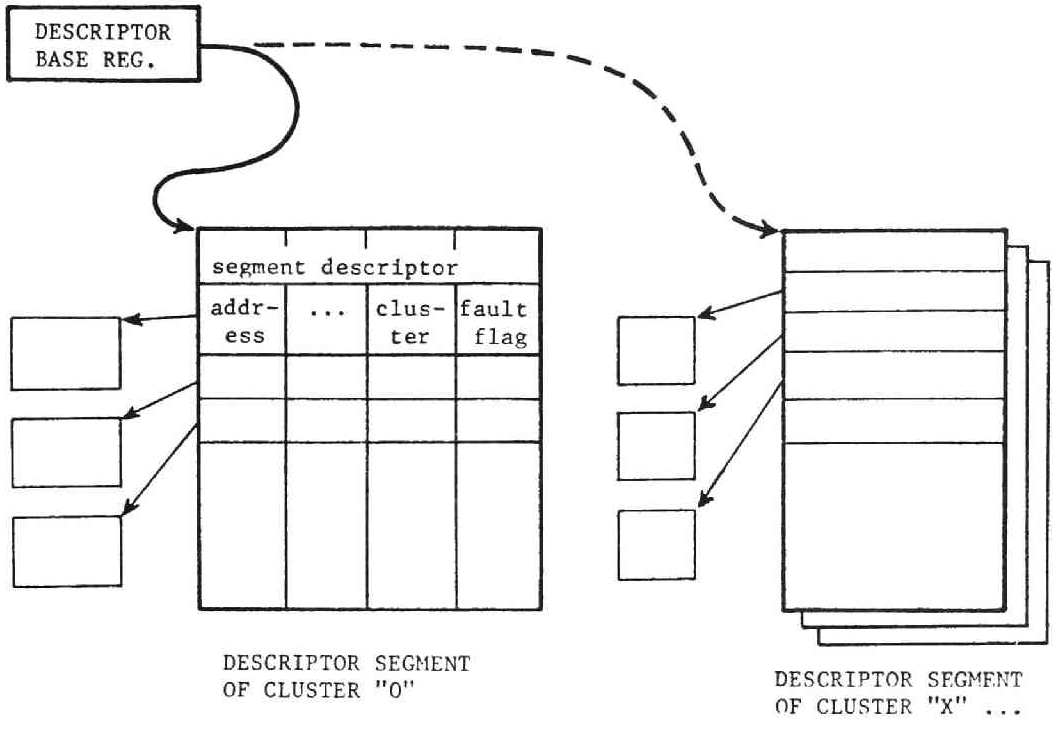


Figure 7.4 Constitution of clusters. One descriptor segment is associated with one cluster. One cluster constitutes the ring domains, which are controlled by the ring bracket fields in segment descriptors. Indication of cluster is placed in the cluster field as shown in Figure 7.3.

Separate data segments are required to execute a process for each cluster. These data segments are linkage segments, stack segments and other data segments. When a procedure call is executed, the stack frame is pushed down. The old stack frame and the new one are chained by two pointers. If a procedure call which requires the cluster switching occurs, the stack frame is also pushed down, but in this case into a separate stack segment

because the called procedure is executed in a different cluster where a separate set of data segments is used. The push-down sequence is executed by the gatekeeper routine when a cross cluster fault is detected.

When a return from the called procedure is taken place, the stack frame is popped up and the execution status of the caller is tried to be restored, using the chaining pointers. If a cluster crossing has not been taken place, this pop-up will be executed successfully in straightforward manner. If it has, access to the older stack frame is forbidden and a cross cluster fault occurs again. The gatekeeper can intercept this fault and switches the cluster back. The following figures show the constitution of clusters and scheme of the cluster switching.

The segment identifiers which designate a segment are assumed to be identical in each cluster. In a multi-programming system segment identifiers, which are used by the processes in the system to designate the same segment, are independent.

The ring protection mechanism is still working in this extension. Hence, calls are restricted to those which do not change ring or call inward, and this ensures that no contradiction is caused even if the original cluster is entered recursively.

The problem of arguments accompanied by a call is troublesome. Generally speaking, arguments placed in the caller's cluster cannot be referred to in the called cluster. Then, it is necessary to place them in common areas which can be referred to both in the caller cluster and in the called one. Further, the validity check of arguments should be undertaken by the procedures which use them. This check is carried out by the called

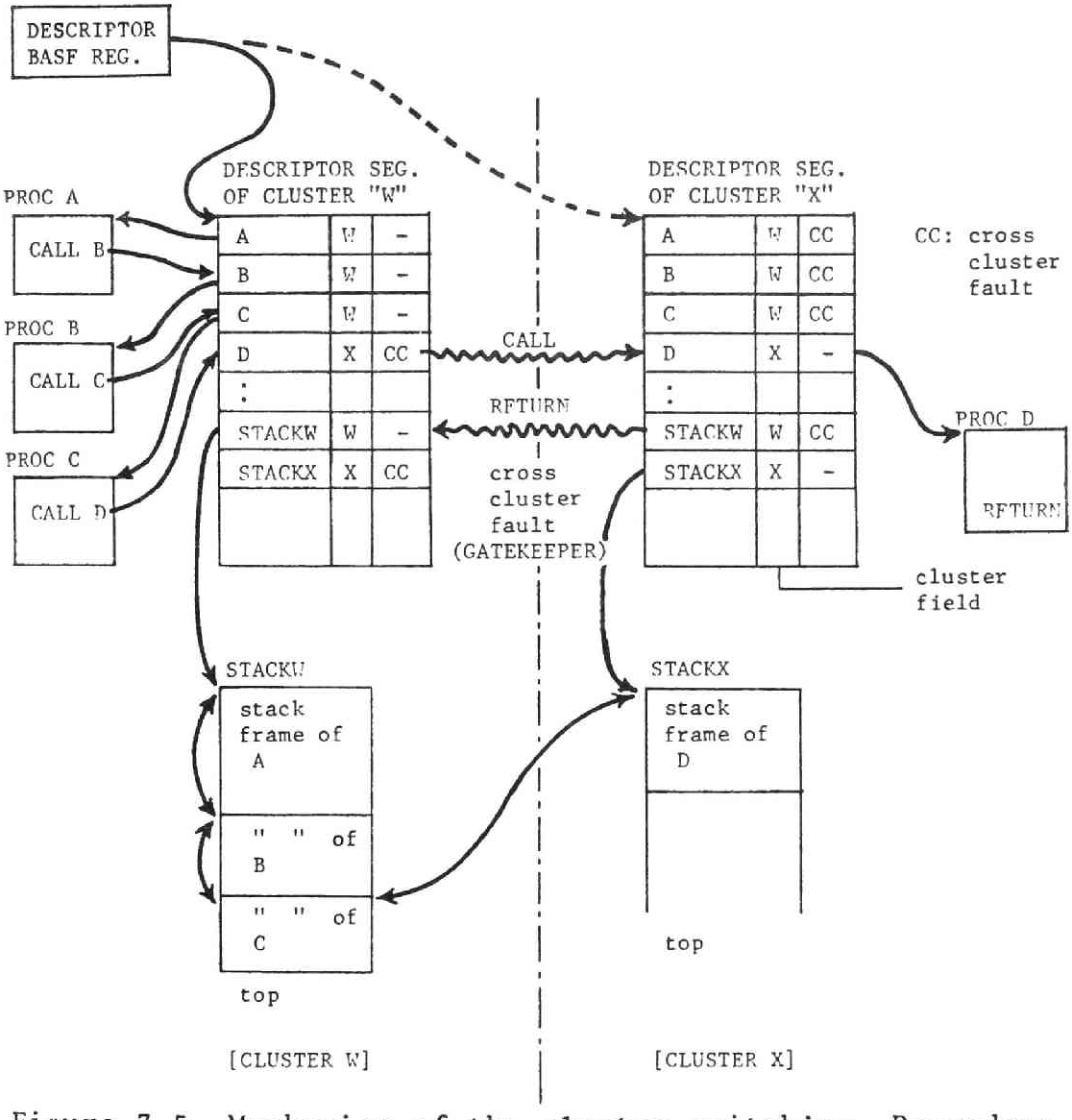


Figure 7.5 Mechanism of the cluster switching. Procedure A calls procedure B, B calls C, and so on. Procedure A, B and C are executed in cluster "W" and procedure D should be executed in cluster "X". The cluster switching is caused when procedure D is called. The swapping of the descriptor segments, the stack segments, etc. is undertaken by the gatekeeper, which also chains the stack frames. Thus, when a return to C is tried, a cross cluster fault is caused and "calls" the gatekeeper again in order to complete the cross cluster return.

procedure in the inner ring (more privileged one) when the ring protection mechanism is incorporated. The problem of constituting common areas is discussed in the next section.

5. Clustering of Domains

Next problem is where to put each segment. The access control information of a segment is stored in the directory entry for this segment. The access control information consists of a list of user names to whom access to this segment is allowed and their access rights. When the ring protection mechanism is used, access rights are designated by access flags which denote the kinds of access permitted and a ring bracket. In order to create independent clusters of domains, information for segment clustering is needed. The simplest way is to add subsystem identifiers to lists of access rights. Usual segments are not given such subsystem identifiers and are usually put into the cluster, say, "0". Those segments that are given subsystem identifiers are put into other clusters each of which correspond to a subsystem identifier, and fault conditions are set in the domain of the other clusters so that the cluster switching condition will be notified. A cluster of domains is assigned dynamically as a new subsystem identifier is encountered.

The problem of constituting common areas for argument passing is rather easy. To do this, the identifier of its own cluster is set in the cluster field of the segment descriptor for a common data segment (see Figure 7.6).

This scheme seems to work, but is still

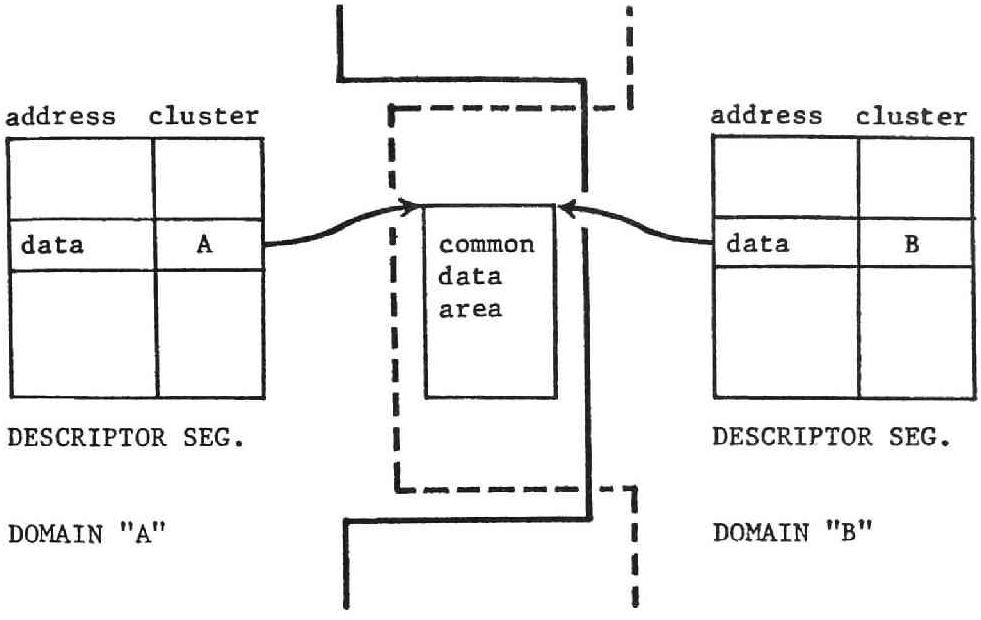


Figure 7.6 Common area to pass arguments. Common data segments may be utilized to pass arguments between procedures which belong to different clusters each other. To do this, the identifier of its own cluster is set in the cluster field of the segment descriptor for a common data segment.

insufficient. If all segments which are needed in a computation are properly sorted and given subsystem identifiers beforehand, above mentioned scheme would work successfully. In usual cases, however, many segments are borrowed from system libraries or other persons' directories, in which case, if it is wanted to create independent domains, additional information is needed to identify subsystems. Such information could be stored separately in linked entries to the directory entries which hold the attributes of the borrowed segments (see Figures 7.7 and 7.8).

Linked entries for this purpose cannot be created automatically. If segment clustering is not indicated at

user name	permitted access	ring bracket	subsystem identifier
JHON	RW	0,5,5	
JACK	R	0,5,5	
BLACK	none		

Figure 7.7 Constitution of access control list (ACL).

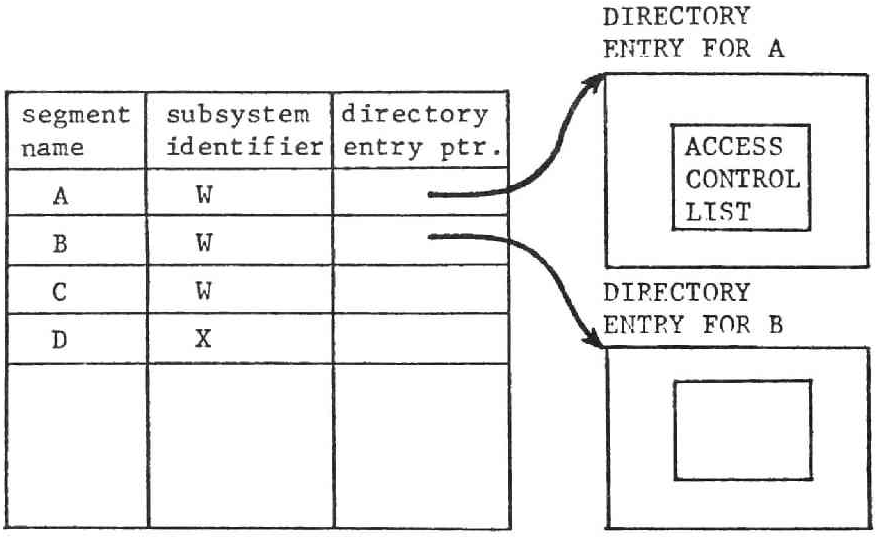


Figure 7.8 Linked entry to hold subsystem identifiers.

all, programs will be executed in the usual ring protection environment.

6. Capability of Ring i State

In ring i it is able to:

1. Change ring number i to j where j is not less than i,
2. Create segments which can be executed in ring j,
3. Create segments which can be read in ring j,
4. Create segments which can be written in ring j,
5. Create directory segments which can be searched for in ring j, and
6. Create directory segments which can be modified in ring j,

but it is not permitted to create a segment which can be executed in a ring whose number is less than i. That is, the ring brackets of segments which can be created are

(r_1, r_2, r_3) where $r_1 \geq i$, $r_2 \geq i$, and $r_3 \geq i$.

It does no harm in itself to create such segments whose read or write brackets are less than i ($r_1 < i$, or $r_2 < i$), but this will have the following side effects:

1. The execute bracket becomes (r_1, r_2) by convention, and this is not permitted.
2. It would be meaningless to create a segment which cannot be read or written by the creator himself.
3. It would cause troubles in a computing system to create a segment which the creator cannot control. Thus, it is usually recommended that the read and write brackets should include ring i.

The initial ring number of a user whose authorized rights are the level of i is i.

7. Additional Comments on the Ring Protection Mechanism

Privileged instructions can be executed only in ring 0. The processor status which controls access rights can be handled only in ring 0. Supervisory procedures can be created only in ring 0.

Users who have ring 0 rights can be registered only from the system console.

It is asked to show a special password to register users in the system. This password can be changed by the user himself freely after he has been registered in the system. The password table should not be stored in its original form but should be stored cryptographically.

CHAPTER 8
CONSTITUTION OF AN ADDRESS SPACE

This chapter discusses the problem of constituting an address space in relation to the information space of the system and to the structure of an address space.

1. File System

On-line information is essential for a computer utility. Information is registered and managed in a directory in a file system, treating a segment as a unit. Thus, the structure of the file system affects greatly the characters of an address space in a computer system.

A directory itself is a segment, and as a matter of course it is registered in its "parent" directory. If a directory segment is registered in a directory in the same way as a non-directory segment, the structure of a file system inevitably becomes a tree structure [DAL1], [RAP1]. The origin of this tree structure, that is, the first directory is called the "root directory". It is supposed that the root directory always "exists" and its location is known.

The structure of a file system generally becomes a tree structure as stated above, and moreover, this structure determines the characters of all elements which are necessary for the file management. That is, space and access are controlled by the hierarchical structure. One segment can be uniquely identified by a route in a file system which starts from the root and reaches the

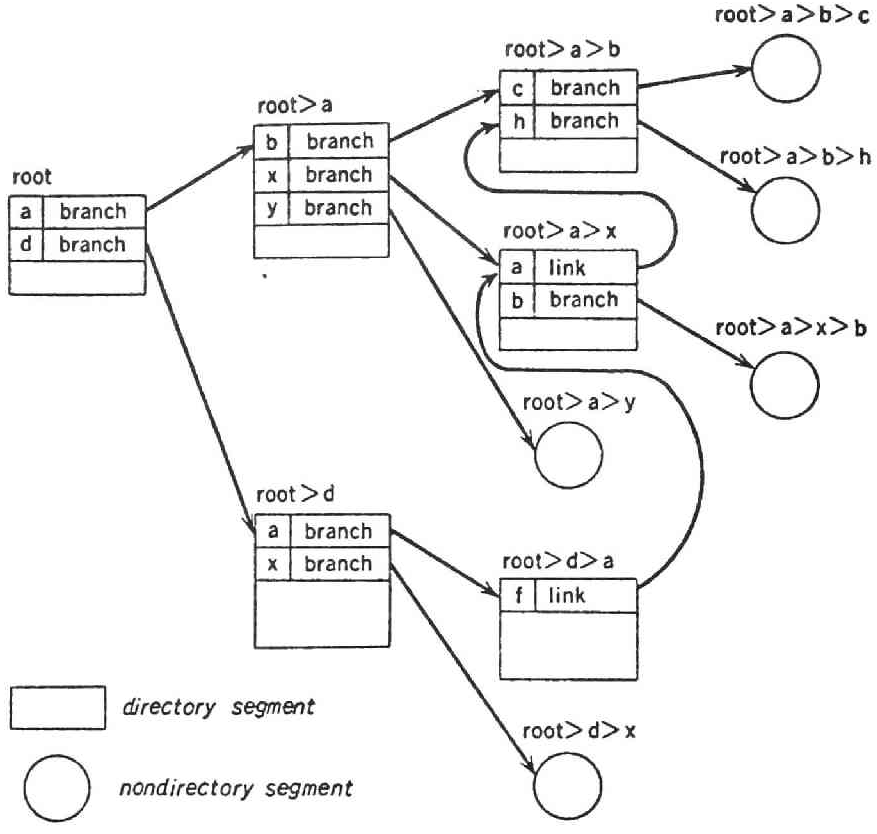


Figure 8.1 Hierarchical file structure. Two kinds of directory entries are shown in the figure, one is a branch which describes a segment, and the other is a link which points at a directory. The problem of aliases is resolved by using links. ">" connects node (directory or non-directory) names to form the path-name of a segment.

directory in which the required segment is registered, showing a sequence of directory names which are the nodes on the route.

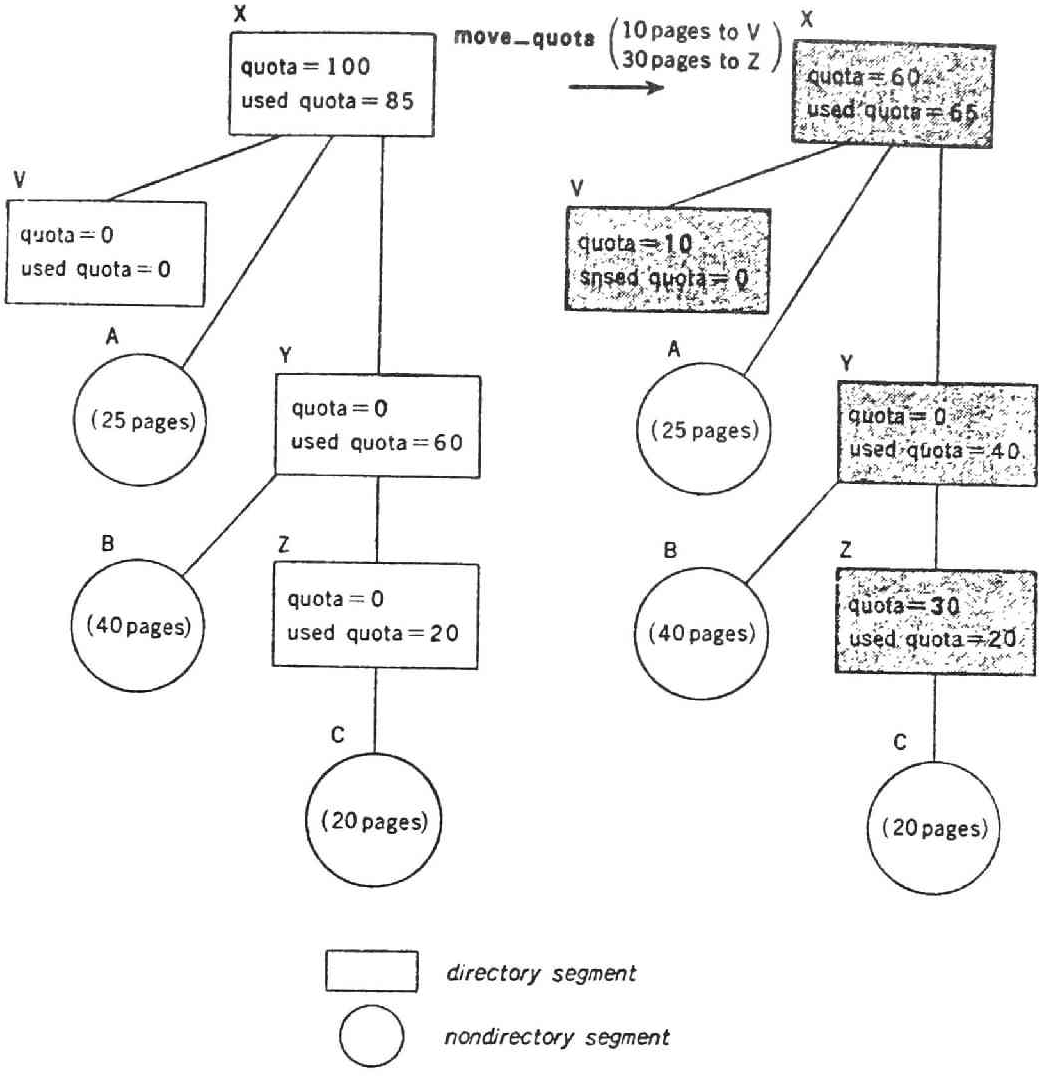


Figure 8.2 Management of the file space. This figure shows that the management of the file space is also enforced hierarchically under the hierarchical file structure. At first, all the quota - file space - is given to the root directory. Quota can be moved to lower directories. A directory which is given quota takes charge of the file space management of files under this directory except those which are managed by some directories under this one.

Thus, there doesn't occur a problem like a homonym in a general file system. A problem about an alias is nominal (see Figures 8.1 and 8.2).

2. Connection of an Address Space and the Information Space

Generally, there are more than one directory in which segments contained in one address space are registered. In order to specify a segment required in a "computation" uniquely in a directory hierarchy, it is necessary to show a path-name. As the most direct method, this condition is satisfied with executing all the references of a segment in a "computation", showing a complete path-name. However,

1. A user doesn't sometimes want to express clearly the location of information which he requires,
2. He sometimes can't express clearly even if he wants, and
3. It is not necessarily caused by rational reasons that one segment is registered in some directory.

Thus, this method is not proper for our purpose.

The logic of a program should be independent of the way of file search. For this reason the method in which one appoints directories and an order of directory search is generally used. One can completely locate a necessary segment by this method, grasping sufficiently a directory in which the required segment is registered. If one uses this method carelessly, there may happen a case in which one can't specify the necessary segment. As it occurs mainly in a case that one wants to use a part of special libraries in combination with his own library and system libraries, one must be careful in selecting directories

and an order of directory search.

3. Dimension of an Address Space

We defined in the previous chapter that an address space is the collection of programs and data to which one process refers in a computation. Moreover, we stated that information becomes a "group" called a segment according to attributes and such segments are gathered to constitute an address space.

The next problem is the logical structure of a segment. It can also be said from another viewpoint that the logical structure of a segment represents the method of addressing of information in a segment. Let us suppose, in the following, that address or location means a logical entity unless specified otherwise. That is, address or location in the discussion of this section has no relation to the physical storage location.

As an address space is a representation of the space for consideration it is necessary to analyze the logical structure of the space for consideration when one discusses the structure of an address space. An address space also exists even in conventional computers which pay little notice about the constitution of "logical address space". But in this case, a distinction between the program part and the data part is ambiguous and only linear addressing is often used. Such an address space is one-dimensional. In case that programs and data are divided into separate segments, an address space becomes two-dimensional at least. How many dimensions of "an address space" do we manage on earth in human activity of consideration, now? For example, a dictionary is an one-dimensional array whose element includes a key word

and its meaning. If a key word has more than one different meaning, each item is regarded as one-dimensional array and an array with such one-dimensional array elements is regarded as two-dimensional space. If a literature is regarded as being only a string of words, it is considered to be one-dimensional space, and if it is regarded as an array of statements, it is considered to be two-dimensional space. As a picture, image and a table are two-dimensional, collection of these things is three-dimensional.

The information space in a computer, which is composed of a complex of one or more than one subsystem, is considered to form the following tree structure (see Figure 8.3).

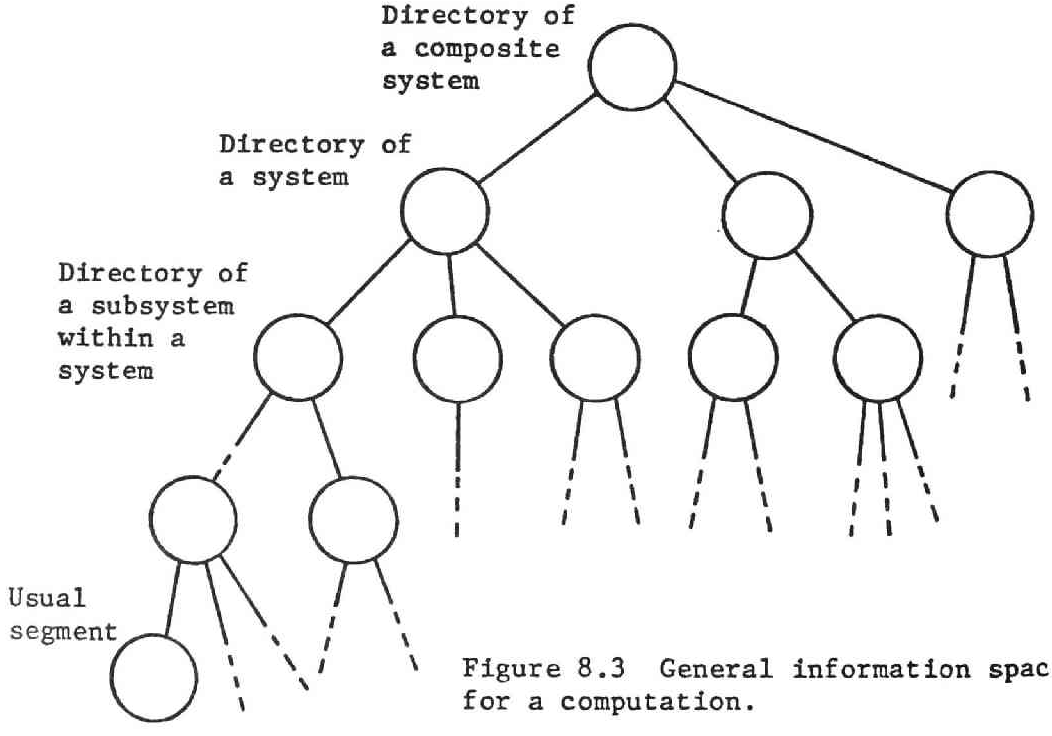


Figure 8.3 General information space for a computation.

The subject which refers to one unit of information (bit, byte or word) is a process and a process executes instructions on a processor. The object to be referred to is either an instruction or data. Various methods are considered in the methods of addressing, but each of array, structure, block structure and stack requires a characteristic addressing method to refer to their unit information.

4. Recursion and Block Structure

When recursion occurs, information which belongs to the domain of the previous level becomes invisible from the present level [ORG2], [BUR1]. A stack is generally used to realize such domains, constituting a domain to place all the automatic variables, including the return pointer and current values of registers, as one element of the stack, which is called a stack frame.

A variable placed in a stack can be referred to by using the stack pointer that points at the base of the current stack frame which includes the required variable. By providing a stack pointer which is accompanied by an upper and a lower limit register, domains which belong to the previous levels are made invisible. (These limit registers are redundant because such limit registers are included in the mechanism of segmentation as shown later.) In case that there are nested domains created by the block structure, a process can get access to the domains below the current level (Such domains correspond to the levels in static meaning which appeared previously). Each currently "active" level is pointed at by a display register [RAN1] (each relative location of the most recently appeared static-levels which are found

in the active dynamic levels).

5. Mechanism of Segmentation

Segmentation is employed in order to realize a two-dimensional address space and to make the followings possible [RAP1], [GLA1], [VSY1], [MCC1], [GIB1]:

The different access controls for each segment,

The dynamic change of segment size,

Dynamic linking, and

The efficient utilization of the memory space.

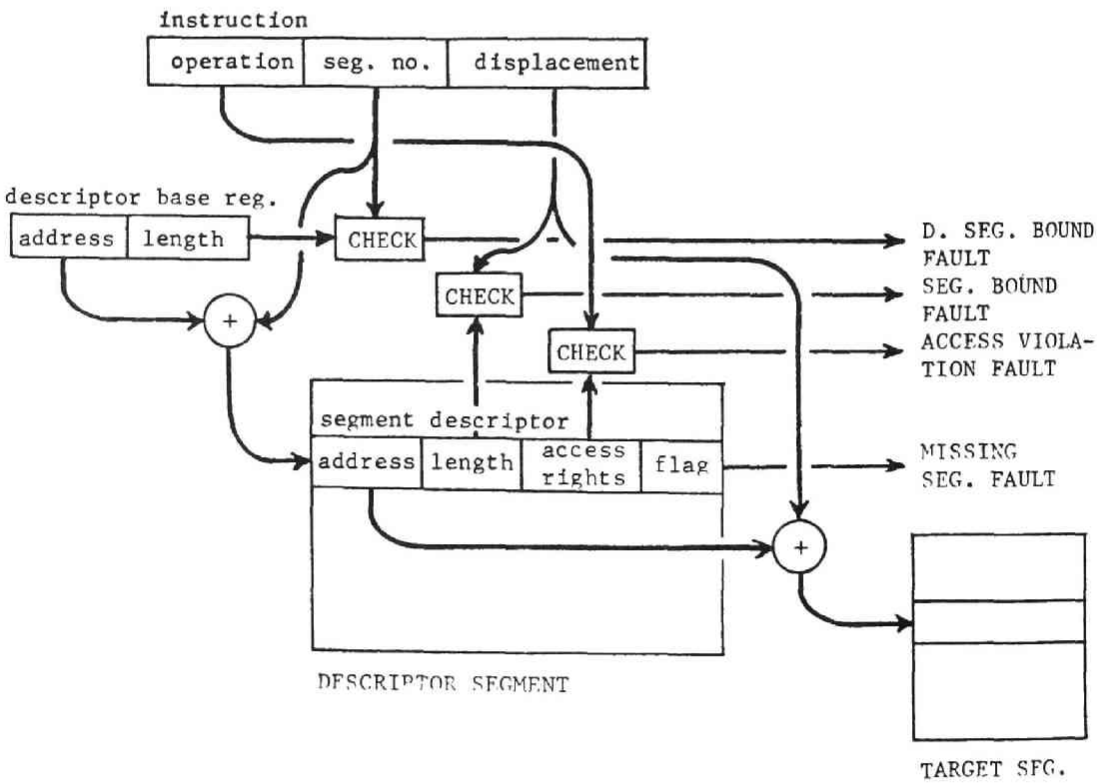


Figure 8.4 Mechanism of segmentation.

The mechanism of segmentation requires the following functions:

To map fragmentary spaces to the contiguous memory space,

```
/* The following lines illustrate the address formation operation
of segmentation in PL/I like notation. */

mechanism_of_segmentation: proc;

dcl 1 instruction based,
      2 operation,
      2 address,
         3 segment_number,
         3 displacement;
dcl 1 base,                        /* descriptor base register */
      2 address,
      2 length;
dcl 1 descriptor_segment based (base),
      2 segment_descriptor (array),
         3 address,
         3 length,
         3 access_rights,
         3 flag;
dcl ( seg_des_pointer,
      target_pointer ) hardware_working_register;

if base.length < address.segment_number
/* check bound of the descriptor segment */
   then descriptor_segment_bound_fault;
   else do;
      seg_des_pointer = addr(base.address + address.segment_number);
      if seg_des_pointer → flag ≠ incore
      /* check missing segment */
         then missing_segment_fault;
         else if seg_des_pointer → segment_descriptor.length <
               address.displacement     /* check segment length */
            then segment_bound_fault;
            else if seg_des_pointer → access_rights ⊅
                  instruction.operation   /* check access rights */
               then access_violation_fault;
               else target_pointer = addr(seg_des_pointer →
                     segment_descriptor.address +
                     address.displacement);
   end;
return;                            /* finished */
```

Figure 8.4 Mechanism of segmentation (continued).

To enforce access control (validate access rights) at each and every reference,

To detect access to a segment which doesn't exist (activated) in the main memory, or

To detect access to a part of a segment which has not been placed yet in the main memory.

The hardware mechanism shown in Figure 8.4 is used in a large computer system.

When segment descriptors are placed in the main memory,

"seg_des_pointer -> segment_descriptor"

implies that one memory reference is taken place. Therefore, in a segmentation mechanism, two memory references are at least required in order to refer to the object information. (Here we do not touch a paging mechanism.) For this reason, associative memory is used to realize fast access [SCH1], [AND2], [HON1]. In a rather small scale computer system in which communications between the CPU and memory is executed synchronously and an indirect addressing mechanism is not provided with it is impossible to place segment descriptors in optional location in the main memory.

We are trying a method adaptable in a middle or small scale computer system. The basic principle of this method is the same as what we stated before, but the segment map, which is logically equivalent to the descriptor segment, is placed in special high-speed memory (a group of registers) and its address is fixed in the system. Thus, the base register which points at the base of the segment map is not needed. As we use a fast memory separately from the main memory, extra clock is not necessary for address mapping and a word in a segment can be referred to within the time of one memory access cycle. For this reason, the address mapping mechanism is

entirely transparent from the viewpoint of memory access time. Of course, functionally it does segmentation distinctly.

There are some computer systems which have segmentation mechanism whose segment number is only of the order of sixteen to thirty two. Such systems cannot realize useful segmentation but only do "segment overlay" which is prepared and bound beforehand. This is because a big software system incorporates a great many segments whose number is the order of one hundred. Thus, the length of the segment number field should be long enough to match such a requirement.

6. Descriptor Segment and Descriptor Base Register

The descriptor segment is an array of segment descriptors [GLA1], [ONI1]. It is used as a segment map to map fragmentary spaces to the contiguous memory space. That is, it is indexed by the segment number, and a segment descriptor in it is referred to by the address formation mechanism of the CPU. A segment descriptor has an address field for this purpose, which holds the start address (of the page table, if paging is used,) of the segment that corresponds to the segment number of this descriptor. The segment number is assigned at the linking time as described in the earlier chapter.

A segment descriptor holds some flags, one of which is used to detect whether or not the required segment is incore. The dynamic loading function of segmentation owes to this flag. Usually, the initial value of this flag is set to the not-incore condition.

A segment descriptor is set when a process requires access to the segment whose incore-flag denotes that the

required one is not incore. This condition is notified by a (missing) segment fault, and the segment fault handler "connects" the segment in the information space in the system to the address space of the process. This is done by searching in the file system as described in the earlier section.

The descriptor segment is not only used for segmentation but also used for access control as well. A segment descriptor is referred to each and every time the target segment is referred to, thus the check of access rights is easily undertaken by placing and verifying the access control information in the segment descriptor. The access control information, which is obtained from the directory entry of the target segment and set at the "connection" time, might include the followings:

Segment length and segment limit,

Kinds of access permitted,

Range of access permitted - ring bracket, cluster, gate of entry, etc. -, and

Kind of segment - privileged, non-privileged -.

To sum up, the descriptor segment plays the central role in segmentation as well as in information protection. It is an array of segment descriptors, and a segment descriptor has such fields as address, length, incore flag, access control flags, and other access control information.

7. Three-Dimensional Address

In a procedure which has the block structure, information is located by a static lexical level and a relative location in this level. In case that such allocation is used, "displays", as shown in the following

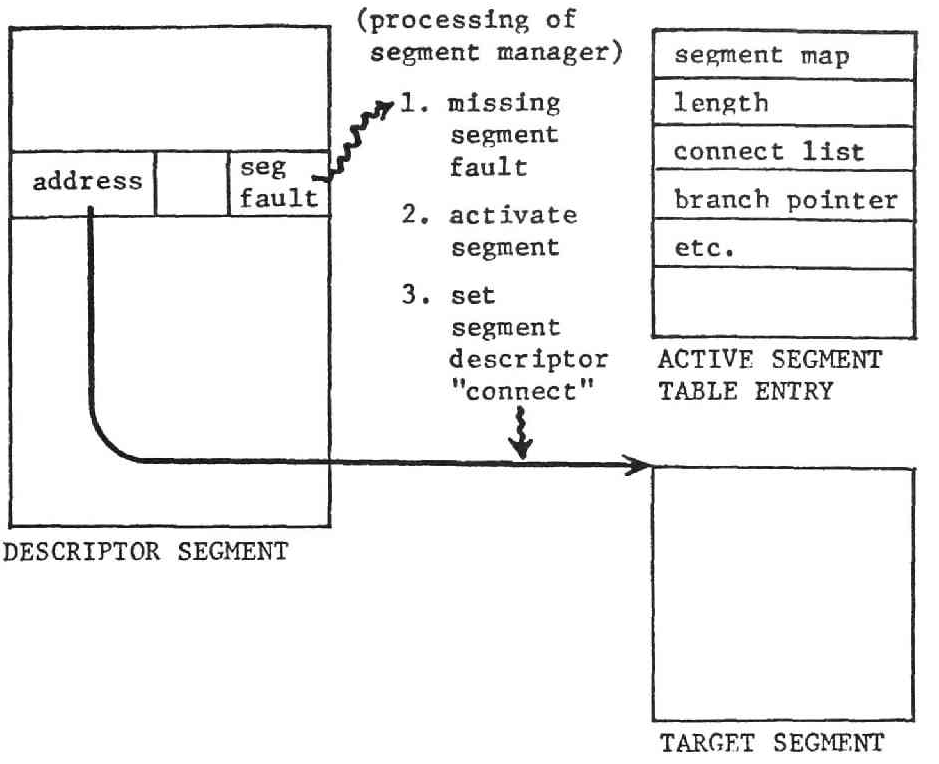


Figure 8.5 Connection of an address space to the information space of the system. The information space is managed by the hierarchical file system. Those segments which are currently beeing used are made active (open) and their directory entries are copied into the active segment table in the main memory. (1) When a process wants to refer to a segment for the first time, a missing segment fault will be caused. A missing segment fault is also caused by other reasons. (2) The segment manager looks up the active segment table if the required segment has already been active. If not, it makes the segment active. (3) Then, the segment manager "connects" the required segment to the address space of the process by setting the segment descriptor that caused the missing segment fault. (4) Now, the process can refer to the segment.

picture, are necessary in order to indicate directly the dynamic execution status of a procedure as it is. Moreover, one additional dimension is needed in order to denote a segment in the environment of segmentation.

address = (segment_number, relative_location)
= (segment_number, lexical_level,
relative_location_in_a_level)

A process is always accompanied with data segments. If one can assign this data segment a fixed segment number, there is no necessity to use a mechanism of general three-dimensional addressing. The problem is whether or not it is necessary to switch stack segments, that is, to make an environment of three-dimensional space and to use it. Then, is there any necessity to support a general multi-dimensional space by hardware? The conclusion is no.

A directory hierarchy constitutes a general multi-dimensional space as stated before. Therefore, there has already been no problem about the constitution of a multi-dimensional space itself. Design issues to be taken into account are as follows:

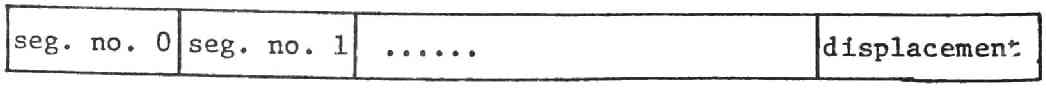
Overhead of hardware which supports multi-dimensional segmentation will be big.

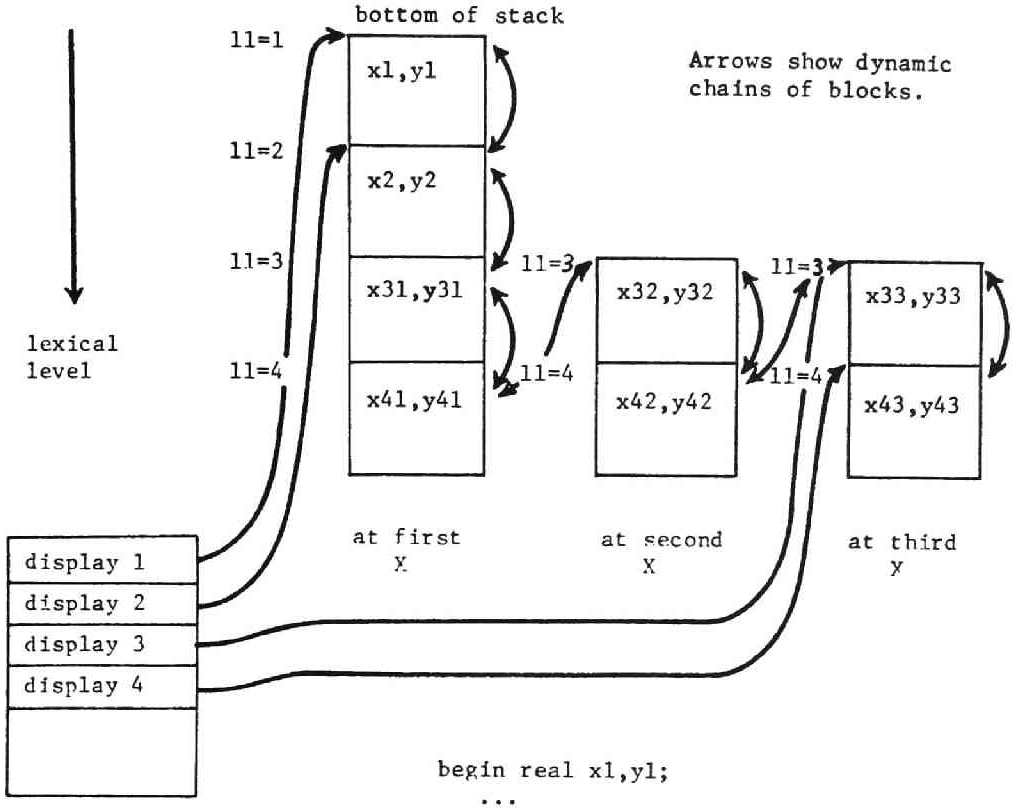
Hardware constitution to support a general multi-dimensional space will become as the following picture.

Addressing in a program:

(seg_name_0, seg_name_1,..., displacement)

Translation to a link:






```
begin real x1,y1;
   ...
   begin real x2,y2;
      ...
      proc Q3;
      begin real x3,y3;
         ...
         begin real x4,y4;
            ...
         X: zzzz
            Q3;
            ...
         end;
      end Q3;
      ...
      Q3;
      ...
   end;
end;
```

Figure 8.6 Block structure and lexical levels.

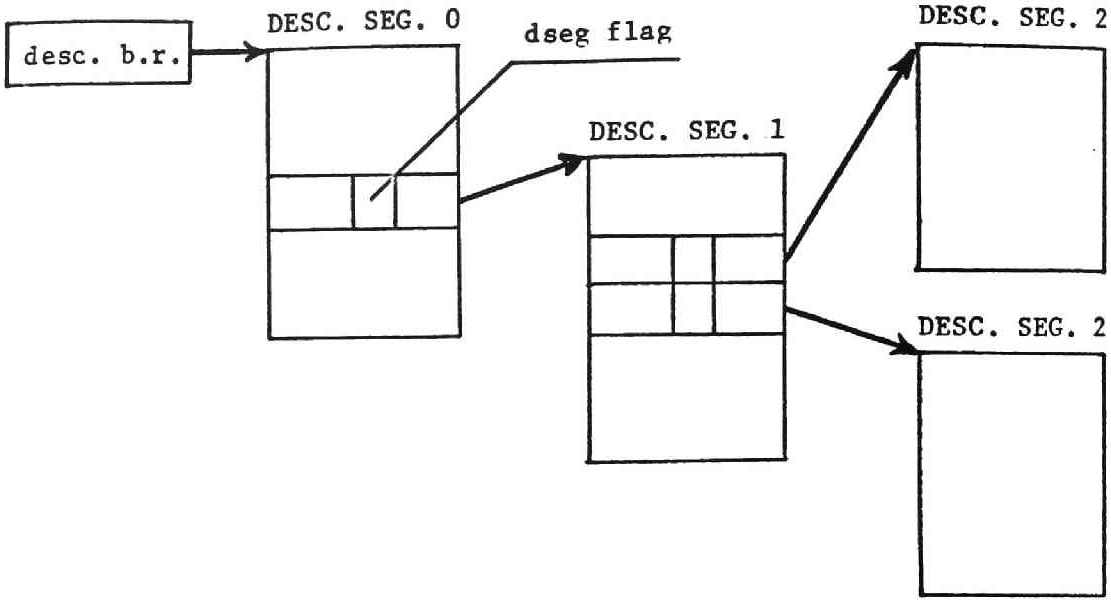


Figure 8.7 Realization of multi-dimensional space. The dseg flag in a segment descriptor indicates whether or not the segment which is described by this descriptor is a descriptor segment (dseg), thus address formation operation continues until a descriptor whose dseg flag is off is encountered.

The problems are:

The segment number 0 is determined dynamically as stated above.

How are the segment Numbers 1, 2,..., ?

How are they determined?

What is meant by the fact that they are determined dynamically?

How to provide the space in which links are put?

It is the case of a dope vector of a multi-dimensional array that the segment descriptors 1, 2, 3,..., are fixed "statically". It is nothing but the hardware which supports a multi-dimensional array. In case that each level of segment fields corresponds to the directory in the hierarchical file system, the mechanism of

multi-dimensional space directly maps the general hierarchical information space.

However, there are the following defects in an implementation of a multi-dimensional space by hardware:

A long address field is required in an instruction.

The degree of indirections increases as the number of dimensions increases.

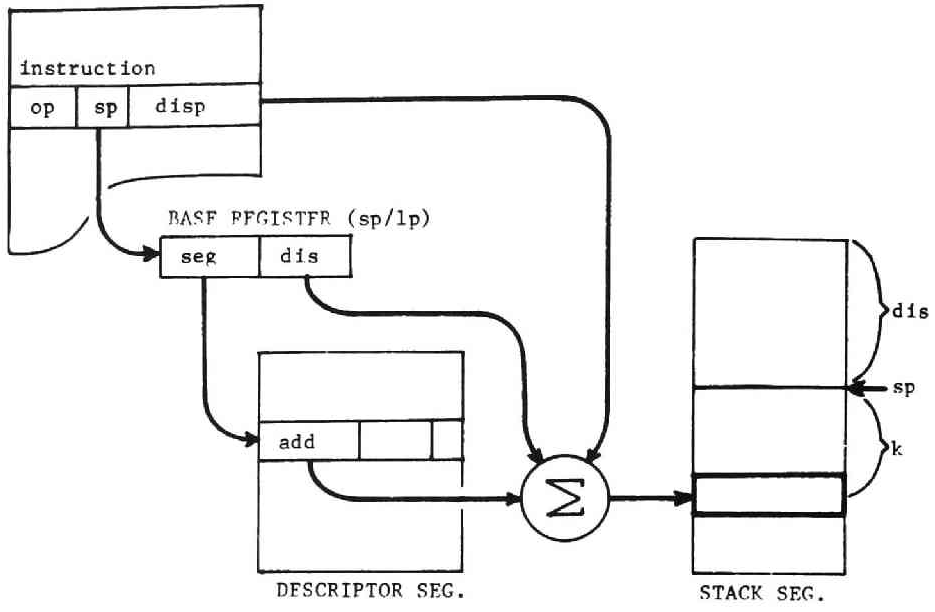
The number of active segments will be a few for each descriptor segment.

Overhead to maintain a number of small descriptor segments will be big.

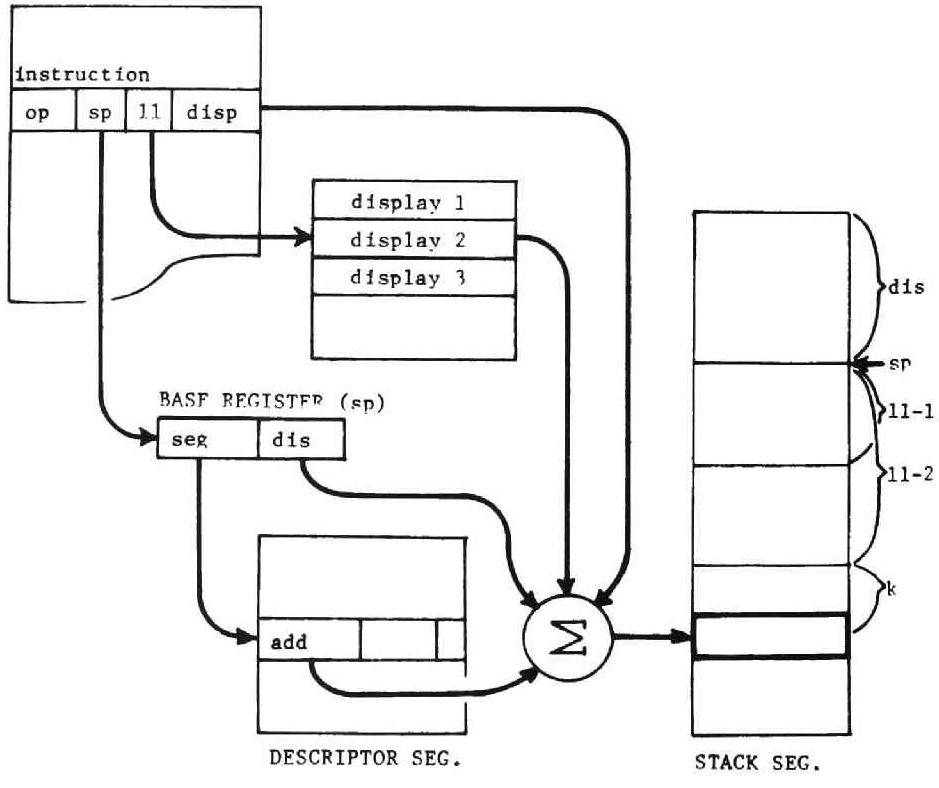
One can utilize the hardware circuitry of segmentation by putting only active segments in the mechanism of usual segmentation. The necessity to constitute hardware circuitry which realizes general multi-dimensional segmentation doesn't seem to be caused for the moment.

8. Support of Lexical Levels

In order to refer to information in one segment in a system where the block structure as ALGOL or PL/I is employed, it is not enough merely to give a displacement but it is necessary to give a lexical level [ORG1], [BUR1], [RAN1] and a displacement. It is for the activation records that requires such an access method, and a general three-dimensional segmentation mechanism is not necessary because the activation records of a process are stored in the limited number of data segments. But a special addressing mechanism becomes necessary in this part as to update the contents of display registers.



(a) Single indexing.



(b) Double indexing.

Figure 8.8 Support for lexical levels.

CHAPTER 9
ESTABLISHING AN ADDRESS SPACE IN A COMPUTER SYSTEM

This chapter discusses the conditions of establishing an address space in a computer system.

1. Conditions Which Specify an Address Space

In this section conditions which specify an address space will be discussed. This argument makes it clear what information should be reserved when an address space is established or process switching is taken place.

1.1 To Show a Process in the System

In order to establish an address space for a process it is necessary to show the existence of this process explicitly in the system. Usually, there is an active process table in a system, in which an entry is assigned to each active process. The minimum information which will be needed to establish the address space of a process is stored in an entry in the active process table.

1.2 The Minimum Information

The minimum information required to establish an address space is:

A. The pointer to the descriptor segment (assuming that the descriptor segment has already existed),
B. The stack pointer (assuming that the stack frame has already contained the status of the process),

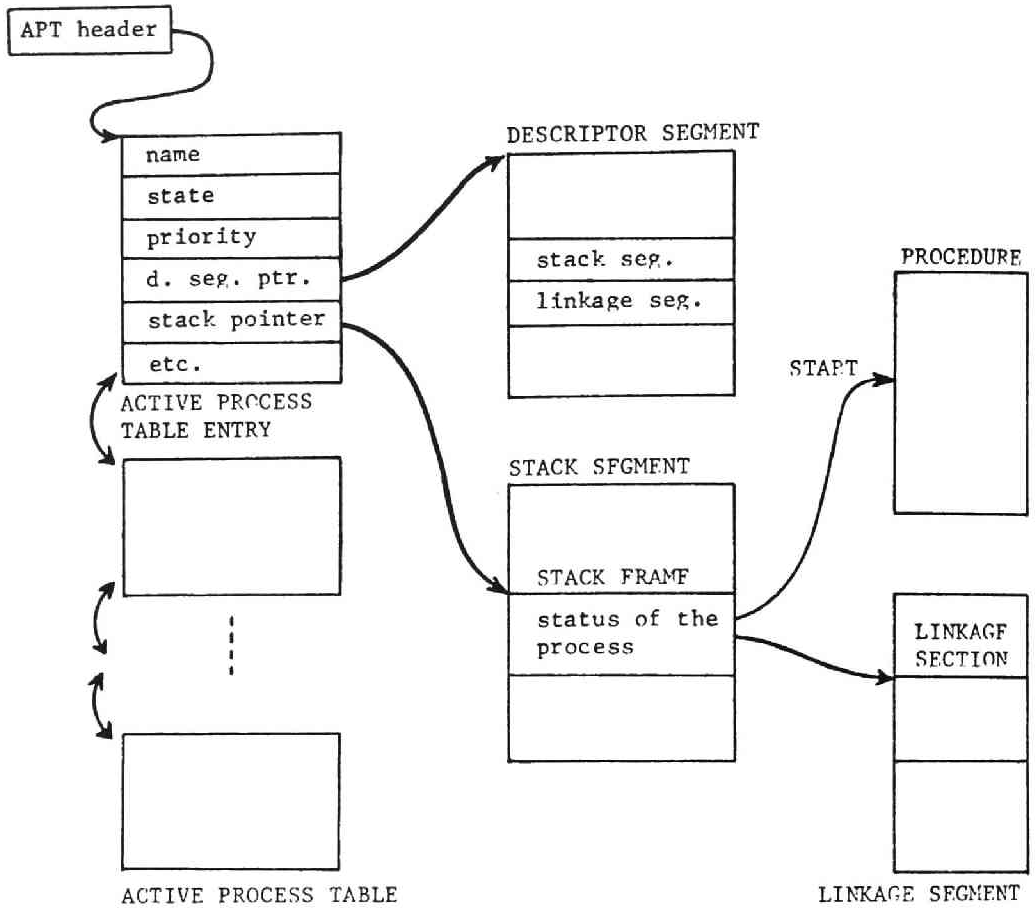


Figure 9.1 Active process table. Active process table is logically an array of active process table entries. Each active process table entry holds the information which is required to select a process to run, to establish the address space and to start it.

1.3 Information Required to Execute a Process

Additional information is needed to execute a process:

C. The instruction pointer to the instruction to be executed,

D. The linkage pointer to the linkage section, and

E. The ring number.

These can be restored from the stack frame upon return,

given the stack pointer, in which case only the information A. and B. are enough to (re)start execution of a process.

1.4 Dispatching a Process

When a process is dispatched from the ready state to the running state, it is enough to set the information shown in B. and to execute a "return". These sequences are executed by the swapping procedure when a process "returns" from the block or the wait routine.

1.5 Initiation of a Process

The following is the direct consequence of the earlier discussion:

> When a new process is created, it is required that initial values for the process are set in the stack frame, and then this process is registered in the system in the blocked state and is made alive by "wake up".

2. Protection of Pointers and Data Segments

The ring brackets of pointers which specify an address space and of data segments of a process are considered as follows:

1. The descriptor segment and the pointer to it (the descriptor base register):

   They are common to all rings. Their ring brackets are (0, 7, 7). They are set only in ring 0 by privileged instructions and are referred to only by the address formation mechanism.

2. Linkage segments and the linkage pointer:

   There are two ways to manage the linkage

segment. One is to use a linkage segment common to all the rings, and in this case ring brackets of linkage segments and the linkage pointer are (2, 7, 7). And the other is to use a dedicated linkage segment to a ring, in this case their ring brackets are (r, r, r) where r is the executing ring number. Here, we assume that the procedures which relate to the protection mechanism are executed in ring 0, and that the other supervisory procedures are executed in ring 1.

3. Stack segments and the stack pointer:

   Each stack segment is dedicated to a ring and its ring bracket is (r, r, r). The segment number should be common to all rings and is added to the ring number to be able to get access to the stack segment in any ring if it is necessary to use separate segment for each ring. This strategy simplifies the calculation of the segment number of the stack segment every time the ring changes, and makes the protection independent of a computing algorithm.

4. Static data area and the static area pointer:

   Each area is dedicated to a ring and its ring bracket is (r, r, r). The relative location of a data area in a static data segment which corresponds to a procedure is found in the linkage offset table that is stored in the header of this static data segment.

When a linkage segment is common to all rings, a stack pointer, a linkage pointer and a static area pointer are needed to specify an address space. When a linkage segment is dedicated to a ring, only a stack pointer and a linkage pointer are needed.

3. Address Space Switching

The descriptor base register (dbr) holds the pointer to the descriptor segment which specifies the address space of a process, that is, the dbr points at the starting location of the descriptor segment and it is implemented as a hardware register in the CPU. In this case the descriptor segment can be placed in any location in the main memory. The dbr is set by a privileged instruction which switches the CPU from one address space to another.

In some system the location of the descriptor segment may be restricted or fixed to a specific place, in which case, if address space switching is needed, the descriptor segment of the required process must be moved into this fixed place. In this case the dbr doesn't (or needs not) exist physically.

Address space switching is executed by the process exchange procedure "get work". In case of a system whose location of the descriptor segment is restricted, this procedure handles the descriptor segment in the absolute addressing mode. This procedure emulates dbr swapping.

To run a process in the newly switched address space the stack pointer in ring 0 (where address space switching is taken place) is set, and the machine conditions such as instruction counter (ic), the linkage pointer (lp) and other working registers are restored from this stack frame. The stack pointer is stored in the active process table entry.

Process switching needs such an instruction as

SWAP_PROCESS (register_set_block_address).

However, there remain some problems:

1. Conventional computers don't possess such an

instruction, hence several instructions are used to accomplish process switching.

2. To minimize the size of an active process table entry, only the values of the dbr and the stack pointer are stored in an active process table entry.
3. Other values to run a process are stored in the stack frame.

Thus, the procedure is coded like this:

```
SWAP_PROCESS:    PROC;
   LOAD_DBR      dbr(apte);
   LOAD_SPR      stack_ptr(apte);
   LOAD_REG      reg(sp);
END;
```

where

apte denotes the pointer to the entry of this process in the active process table,

dbr and stack_ptr are the fields for the pointers to the descriptor segment and to the stack frame in the active process table entry of this process respectively,

sp denotes the pointer to the stack frame set by the pointer in the entry in the active process table, and

reg is the field from where the values of registers are restored in the stack frame.

CHAPTER 10
INTRA-PROCEDURAL COMMUNICATIONS

It is required that a procedure is pure as one condition for information sharing. For that reason, a process requires its own impure data segment for the execution of a procedure. There are two kinds of information which is stored in this segment. They are:

Static variables and links whose allocation is static, and

Activation records and automatic variables whose allocation is dynamic.

In addition, a procedure should be able to share even itself in order not to impose an improper limitation on the expression of an algorithm, thus recursion is often resulted. Variable areas which have lexical levels in case of the block structure are also necessary. Then, it is required that a dynamic data segment is constituted as a stack in order to have such an ability. That is, a procedure can refer to variables relative to the base (lp) which points at the area provided statically and to the base (sp) which points at the area provided dynamically in the execution time (, and to the base (sp) and displays (ll_1, ll_2,..., ll_n) in case that lexical levels are created). One element of a stack which we consider here is not one unit cell but a storage area which contains all of the activation records and the dynamic status of a process that relate to one procedure or a lexical level of a procedure. This unit is called a

stack frame. Dynamic status of a process is a kind of automatic variables as will be explained later.

The status which is concerned with the information protection should not be changed improperly. Apart from discussing the protection of the status information, we can discuss the problem of intra-procedural communications in a well formulated manner if we consider that all the current status of a process are stored in one stack frame.

1. Management of Process Data Segments

In this section the management of process data segments is discussed in relation to their access rights. In principle it is required to isolate information whose access attributes are different. But the processing is rather "contiguous" in logical sense unconcerned with the discontinuity of access rights. Data segments which a process uses can be grouped into static data segments and dynamic data segments. A static data segment is used to hold static variables and a dynamic data segment is used to hold automatic variables.

1.1 Stack Segment

Automatic variables are the activation records of a process and it is desirable to store them in stack segments [BOC1], [BOB2]. Each time access rights of a process change, it is necessary to switch stack segments (as a rule) lest the activation records of a privileged state should be altered in a non or less privileged state.

It is, however, not necessary to do so if there is the inclusion relation among protection domains as the

ring protection mechanism has. This is because domains are generally switched when less privileged procedures require more privileged processing executed in a more privileged domain, and the inverse case happens in the most limited situations that the caller doesn't use the results of the processing for its decision.

Examples of such a case are found when the system creates a new process and wakes it up, and when a teacher runs and marks programs of students in a less privileged and confined domain. The former is the case that the system gives control to a newly created process by "returning from the block procedure of the process exchange module" [STE1] which is executed in the privileged domain. However, this case can be reduced to "the return to the imaginary caller which called the block procedure before the process becomes alive". The latter is the very exceptional case which needs to prepare a separate stack by the software intervention.

Each stack frame is connected with its predecessor and successor by pointers and is maintained its logical contiguity even if it is placed in a separate segment. When a return instruction is executed and the previous stack frame pointer points at the caller's frame in a separate stack segment, this condition can be notified, as is previously mentioned, by an access violation fault, and then the gatekeeper, the domain switch handler, is invoked.

In case that there are stack frames which are created in different rings, care must be taken to restrict the scope of searching for a signal handler within the stack frames which are created in the same ring. The ring number is one of the activation records and stored in every stack frame, so such a check would be

carried out very easily. The discussion on the condition handling and a signal handler is left behind.

1.2 Static Data Segment

Static data are as follows:
Variables which are allocated statically,
Accounting records for a procedure/entry,
Core of a pseudo random number, etc.

Their area size is pre-assigned when they are declared. Static data do exist during the life of a process which created them, and data inherent to each ring exist concurrently. Thus, more problems arise concerning with their management, as compared with the management of a stack segment. It has already been mentioned that data whose access rights are different should be separately stored.

For static data inherent to a process, the number of data segments depends upon the clustering strategy of access rights. Static data inherent to a process are:

1. Data written and read by a procedure,
2. Data read by a procedure (read only), and
3. Pointers which are used to refer to other segments (read only data for a usual procedure, written by the linker).

Data 1. is managed per ring basis. The size of a data area for a procedure is determined at the time this procedure is created, thus dynamic allocation is not necessary for this area. A data area for a procedure is allocated in a data segment prepared for the current ring at the first time this procedure is executed. The offset value which shows the location of the data area within this data segment is registered in a table indexed by the segment number of the procedure associated with this data

area.

A (pure) procedure refers to its data area, setting the pointer register the offset value within the data segment from the offset table.

Data 2. is usually included in the procedure body itself except less frequently referred data as a symbol table for debugging purposes, which is stored in a separate segment.

Such data may be shared in other rings as a procedure body itself.

There are several methods to manage pointers (links) of data 3..

1.2.1 In Case that the Pointer is Invariant in Every Ring

There is a case that different segments are required from the same procedure in different ring, but such a case is rather rare, that is, the pointer is invariant while the access rights are variant. In this case only one linkage segment is needed and the management of such a data segment is simplified, but

1. It is necessary to execute the linker in the supervisory ring 0 or 1,
2. As mentioned above, the target segment is invariant even if ring changes, and
3. A procedure needs a separate pointer to refer to links in addition to the pointer to the data area.

   The common linkage segment should not be altered in non-privileged rings by 1.. But in order to realize more reliable protection it is better to minimize the size of the supervisory kernel which relates to the protection mechanism. Method shown in this section violates this policy. The problem of removing the linker from the security kernel is

discussed in the later chapter.

1.2.2 In Case the Linker is a Non-Privileged Procedure

In case that the linker is a non-privileged procedure and is executed in the faulted ring, links will be made in various rings, therefore linkage segments should be prepared for each ring. It is also necessary to prepare linkage segments for each ring if different links are needed in different rings. (To make the linker a non-privileged procedure is discussed later in Chapter 12.)

In such a case a linkage section related to a procedure can be merged with a read and write data section related to this procedure, and the management of a data segment can be included in the procedure for data 1., that is:

1. Only one pointer is needed by a procedure to refer to both the data area and the links,
2. Can minimize the layer of the protection, and
3. In many cases links are the same in different rings, but linking is required in every ring where the object segment is referred to. However, rather a few segments are referred to in different rings and this overhead can usually be neglected.

2. Constitution of a Linkage Segment

A linkage section is small and may result a big page breakage if each linkage section constitutes a segment. By contrast, if linkage sections in the same ring are combined into a combined linkage segment, those linkage sections whose related procedures become disused cannot easily be eliminated from this linkage segment.

CHAPTER 11
INTER-PROCEDURAL COMMUNICATIONS

Inter-procedural communications are accomplished by a call to a procedure, a return from a procedure, a non-local go to from a procedure, and an implicit call caused by an interrupt or a fault.

One of the reasons which make the program logics needlessly complicated is that several entirely different forms of procedure invocation are employed in today's computer system. In addition to a usual procedure call, SVC (supervisor call instruction), interrupts and faults are, in fact, invocation of a procedure in their primary functions, and sometimes protection issues are raised by the contents of their processing. So far, these two points have hardly been separated, and each of them has been processed in an odd manner. And the size of the security kernel has unduly been big.

By unifying the various sequences of procedure invocation, the structure of software system would be much clarified, and the size of the security kernel would be minimized. Thus, the integrity and the reliability of the system would be improved much.

1. Call and Return

A call and a return are discussed in the early section about the ring protection mechanism. This section only discusses the following cases:

A call and a return in the same ring, and

An inward call and a consequent outward return. A call is accomplished by a call instruction and a return is accomplished by a return instruction.

Automatic variables which are referred to in a procedure are allocated in a stack. The area of stack allocated to the variables for a procedure in the current activation level is called a stack frame [ORG1]. The status and the contents of the registers in the CPU are all automatic variables. By saving all the automatic variables in the stack frame upon call and by restoring them upon return, it is able to support the constitution of reentrant or recursive procedures by the fundamental framework of the system instead of a function of individual language processors. The call stack in ALGOL [RAN1] corresponds to the stack frame. A few large computers are equipped with such an instruction that performs both stack push and procedure invocation. SKB instruction in HITAC 8700/8800 is an example of such an instruction [SIM1]. By saving or restoring all the status of the CPU in one instruction with stack push/pop and call/return, the sequence of call to or return from a procedure is much simplified, and the execution speed would be improved much.

1.1 Call Instruction

The followings are the sequences executed by a call instruction:

A. Save all the status of the process into the stack frame. The status to be saved are:

1. Values of automatic variables including the contents of the status registers and the general or working registers in the CPU,
2. The return pointer to the caller's procedure,

3. The stack pointer, and
4. The linkage pointer.

All are the activation records of the caller which are needed to be restored when control is returned to the caller. The minimum information required to restore the activation records is only the stack pointer, provided that the address space is set for this process.

B. Push the stack frame.

This will set the stack pointer to the new stack frame to be used by the called procedure. If a call is to cross domain, switching the stack segments might also be needed.

C. Transfer control to the called procedure in the proper domain (ring).

The target domain (ring) is determined by the access control information shown in the segment descriptor of the called procedure. The check of the gate whether a correct entry is selected or not is also undertaken.

D. Set the pointers to execute the called procedure.

In order to execute the called procedure, the stack pointer and the linkage pointer must be set.

1.2 Return Instruction

The followings are the sequences executed by a return instruction:

A. Pop the stack frame.

This restores the stack pointer to the previous stack frame which has been used by the caller.

B. Restore all the activation records.

It is necessary to restore all the activation records except those that cannot be changed in the called domain of execution including the linkage pointer and the contents of other working registers from the popped stack

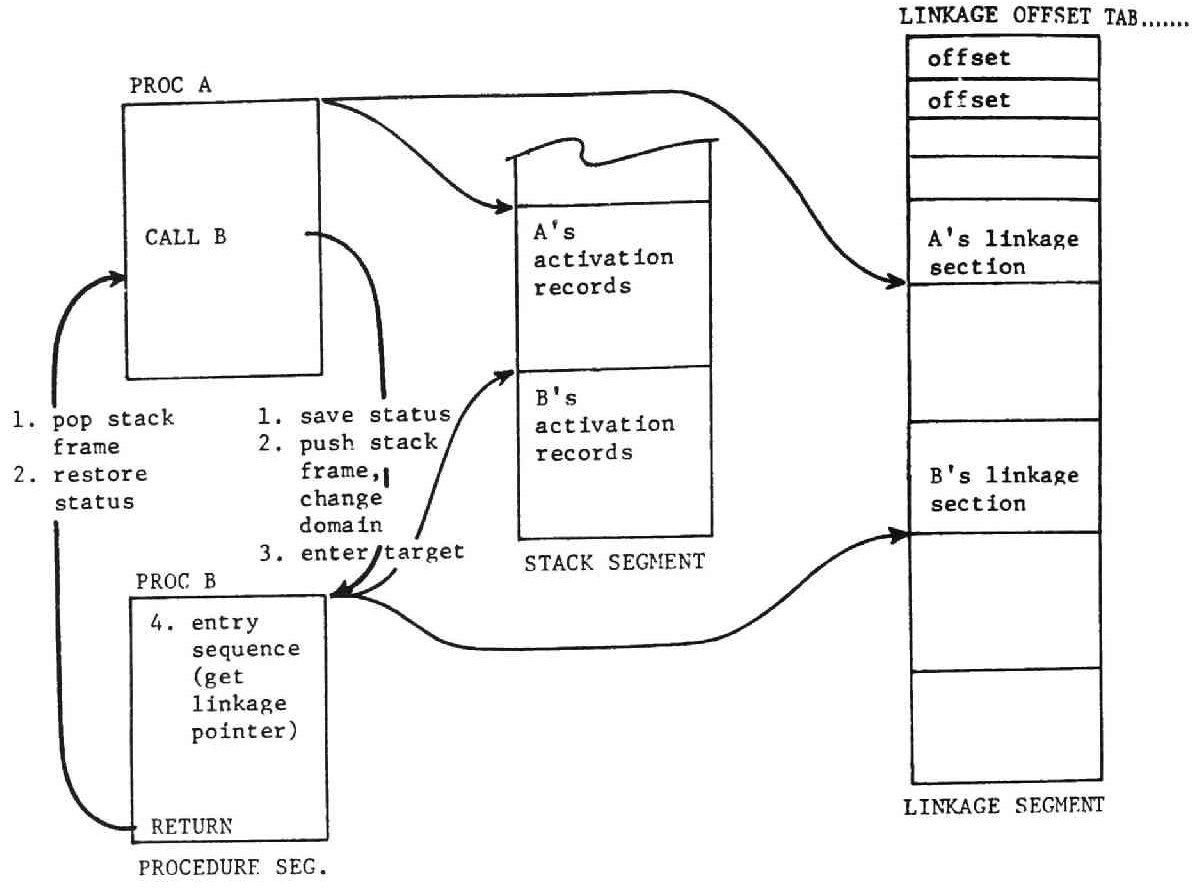


Figure 11.1 Call and return

frame.

The information for protection control in the process status saved in a stack frame is essentially the one which the most privileged procedures can process. Thus, such information cannot be restored without any restriction. Restoration of the process status is validated by imposing the following conditions:

1. Only return to the caller within the same domain or in the less privileged domain is permitted, and return to the caller in the more privileged domain is needed to be processed by the software

intervention.

2. The domain returned to is determined by selecting less privileged one between the current one and the one indicated in the process status.
3. Restoration of the process status is confined to the information which can be processed in the called domain.

And there would be no fear that the protection status would be impaired unduly. Of course, the saving of surplus information wastes time and space and should be avoided. Thus, there is a difference in the amount of the saved status between a usual call and an interrupt or a fault. (Here, more privileged means that the domain has some access privileges which others don't have, and less privileged means that the access privilege of the domain is proper subset of others.)

2. Elimination of SVC

A supervisor call (SVC) instruction switches protection domains and calls a privileged procedure. The domain of protection should be determined and switched by the domain switching mechanism according to the access rights of the target segment in the course of the execution of an instruction which transfers control such as a call or a return. In a computer system which is not equipped such a domain switching mechanism, programmers are obliged to use such instructions as SVC and LPSW (load program status word) to switch domains according to the contents of the processing. Thus, program logics are affected by protection issue. SVC can be eliminated by connecting the function of transferring control with the domain switching mechanism (refer to Chapters 6, 7 and

16).

By employing such a domain switching mechanism, programmers are freed from the business of switching the protection domains, which has nothing to do with the algorithms of programs, and the structure of programs is made clarified. All procedure invocation including one of supervisory procedures can successfully be executed by a usual call/return instruction.

3. Condition Handling

"Interrupt handling" facilities in software can be found in the condition handling of PL/I [IBM3], [COR3], [MSPM]. These facilities are to declare a handling procedure for a predefined "condition" and to execute this handling procedure when the specified condition is notified, using a signal statement, or is detected by the hardware circuitry.

As a handling procedure is searched for in the dynamic descendant manner, handling procedures might be registered in the current stack frame when they are declared, and then signalling might be accomplished, searching for the required condition handler in stack frames from the top to the bottom when a signal statement is executed or a signal condition is detected by the hardware circuitry.

This handler searching must be carefully controlled lest protection violation should be caused. Condition handlers might happen to be registered in the stack frames whose access privileges are higher than those of the procedure currently executing a signal statement. Such condition handlers cannot be executed, because if they can the result is that more access privileges are

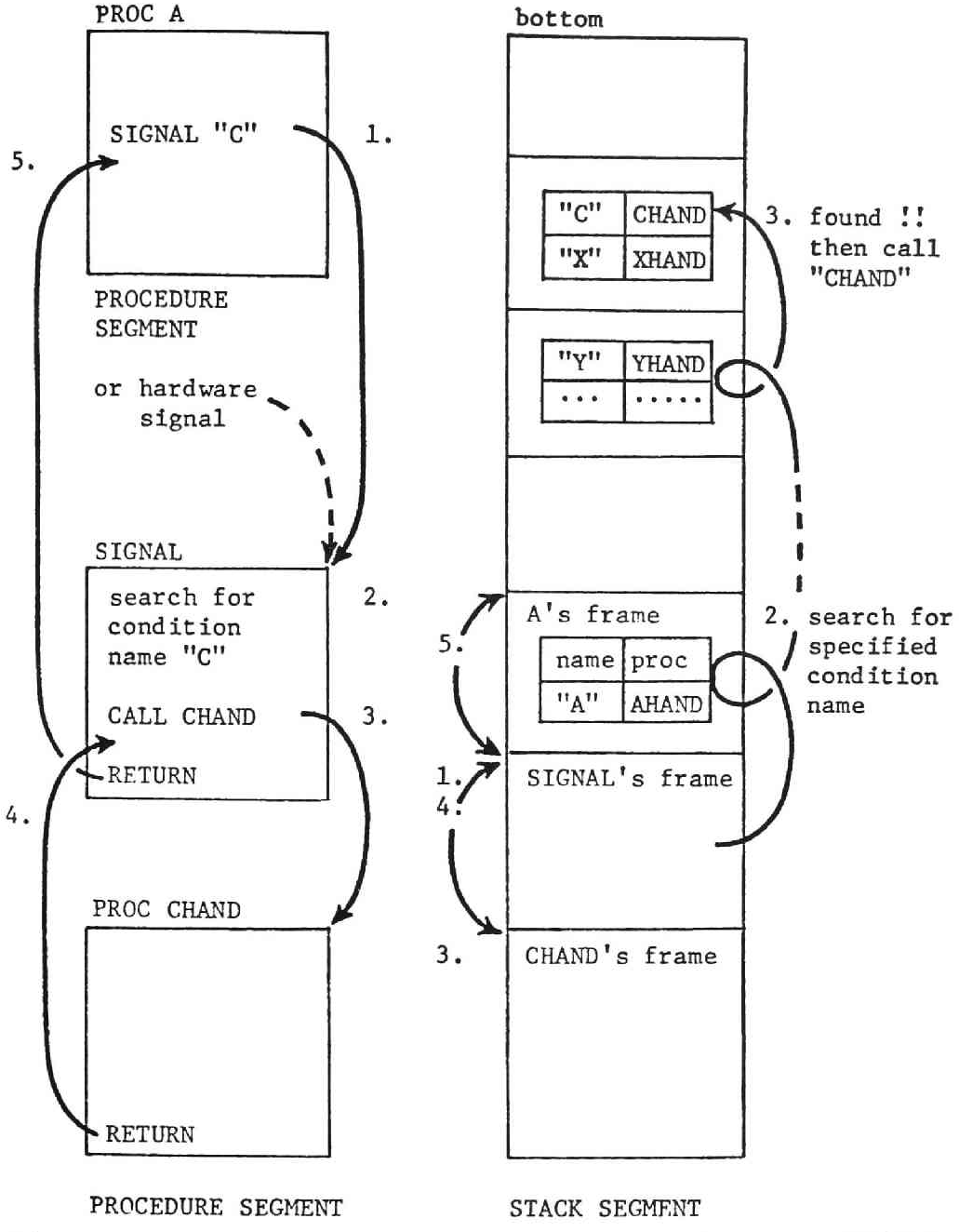


Figure 11.2 Condition handling. When a SIGNAL statement is executed or a hardware signalling is required, (1) SIGNALling handler is invoked, (2) the condition name is searched for in the stack frames, and (3) the handler procedure found in step (2) is invoked.

given without any check of the validity.

In a system where only inward calls and outward returns are permitted, such cases do not happen because no more privileged stack frame than the current one exists.

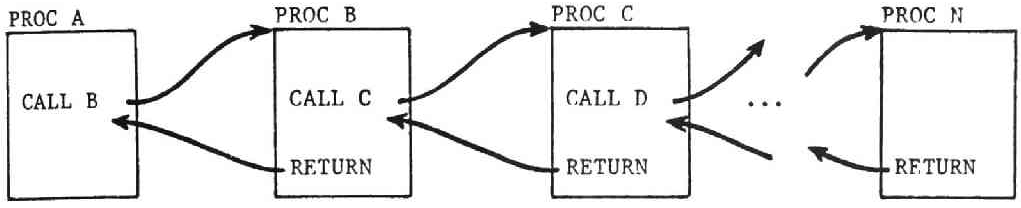
Further, condition handlers might happen to be registered in the stack frames whose access privileges are lower than those of the signalling procedure. Such condition handlers should not be executed, because more privileged procedures should not rely on the results of less privileged ones and more privileged rings should prepare their own signalling environment.

It will, then, be reasonable to impose such a restriction that a condition handler should be searched within the same domain. That is, signalling should be accomplished without changing the domain. This restriction from the standpoint of the protection will not affect the program logic unduly.

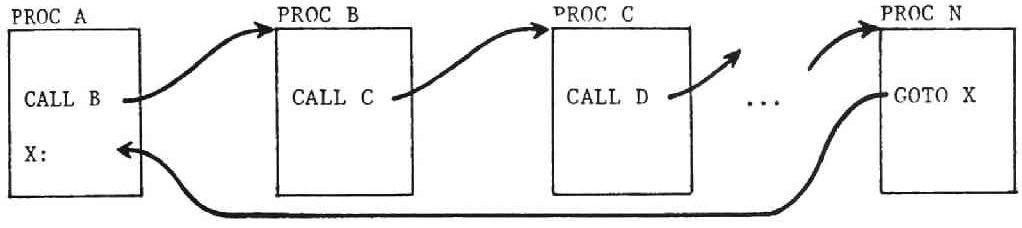
4. Non-Local Go To

When a return instruction is executed back tracking the chain of calls, the stack frame previously pushed down by the call instruction is popped up.

It sometimes happens that control is returned to another place other than the normal return point in the caller's procedure or to the procedures on the call chain, skipping the normal return sequences by a reason such as job aborting. A call is only permitted to enter a brand new procedure or a brand new level if recursive call is executed, and this non-local go to is a kind of returns, that is, control is returned to a procedure which previously called other procedure and then



(a) Normal call and return.



(b) Non-local go to.

Figure 11.3 Normal call / return and non-local go to.

transferred control to it and to which control has not yet been returned. In contrast to a usual return which returns control to the point where the call is executed, this non-local go to returns control to a place other than the normal return point, and in this sense, it is sometimes referred as an "abnormal return".

A return point of a non-local go to is also designated, using a two-dimensional pointer. The problems to carry out a non-local go to are:

1. A non-local go to may skip the sequences to disengage those segments which some previously called procedures have created and used temporarily as data segments instead of using the stack segments, and such segments may remain as vagrants.
2. A non-local go to may leave some locks for common resources locked.
3. It is necessary to pop up settings of condition

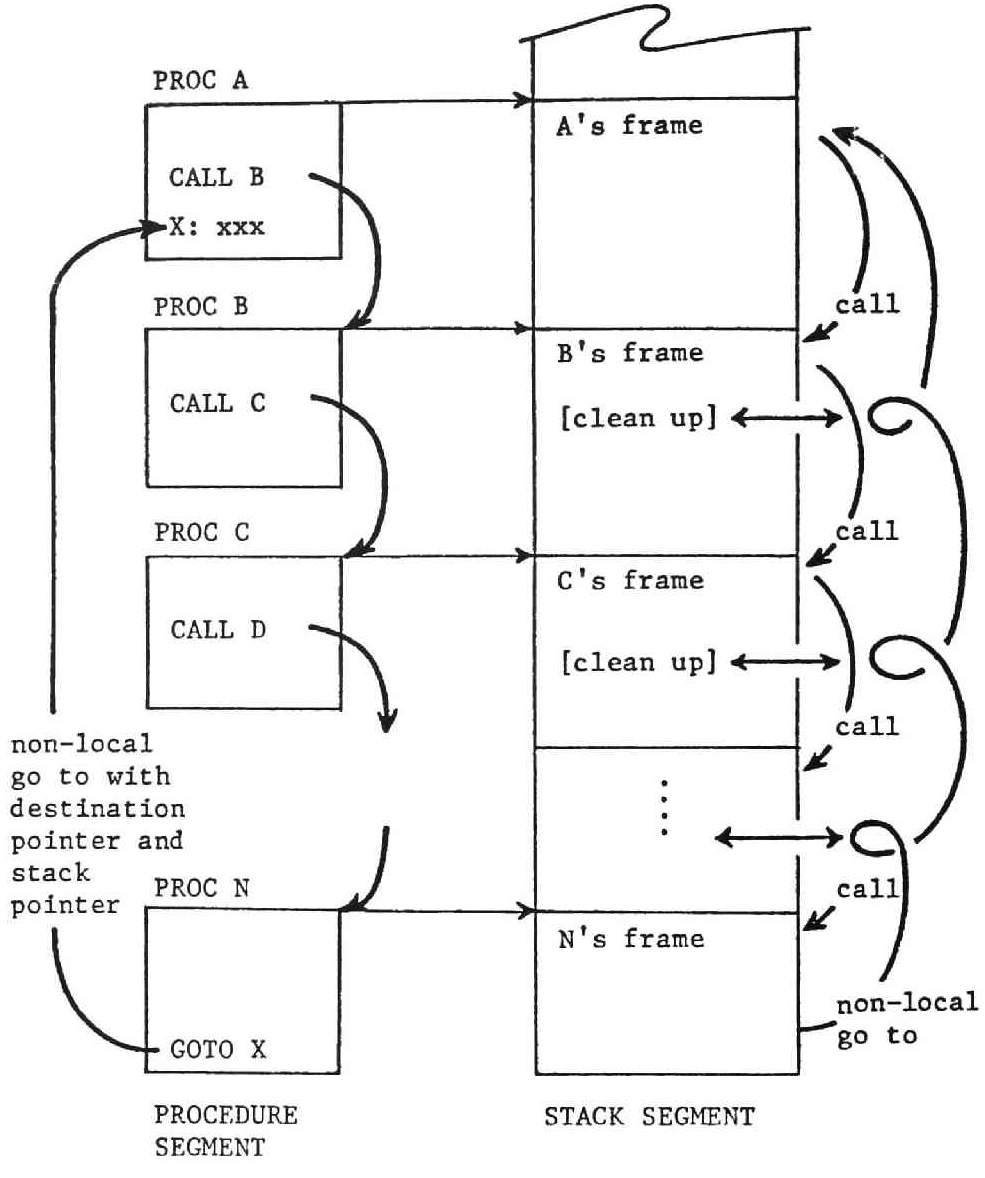


Figure 11.4 Processing of non-local go to.

handlers.

To solve these problems it is necessary to execute post processing sequences, popping up the stack frames which are to be skipped by this non-local go to one by one.

Post processing procedures to disengage data

segments or to unlock locks may be set in the stack frame of a procedure which needs such processings. On a non-local go to stack frames are popped up one by one, executing post processing sequences designated in stack frames, until the intended frame is reached.

Thus, in order to accomplish a non-local go to, not only the destination pointer but also the stack pointer which points at the stack frame been used when control is returned are needed. Further, if a system incorporates lexical levels, the value of level and the set of display registers are also needed.

As has already been mentioned, all the necessary status to resume execution of the returned procedure can be restored, using this stack pointer. Resetting (or reverting) condition handlers is automatically accomplished by popping up stack frames, so no more problems remain here. [ORG1]

5. Implicit Call

There are another kinds of calls of which programmers are not explicitly conscious but which affect the structure of inter-procedural communications. They are interrupts and faults, whose handling is the most complicated and mysterious part in the modern, and especially large computer architecture. And it is true that this affects the structure of operating system greatly. The purpose of this section is to clarify the logical structure of them and to make the structure of operating system easier to constitute and to understand.

An interrupt is the hardware mechanism to notify an event relevant to asynchronous processing or an asynchronous action to a process. The interrupted

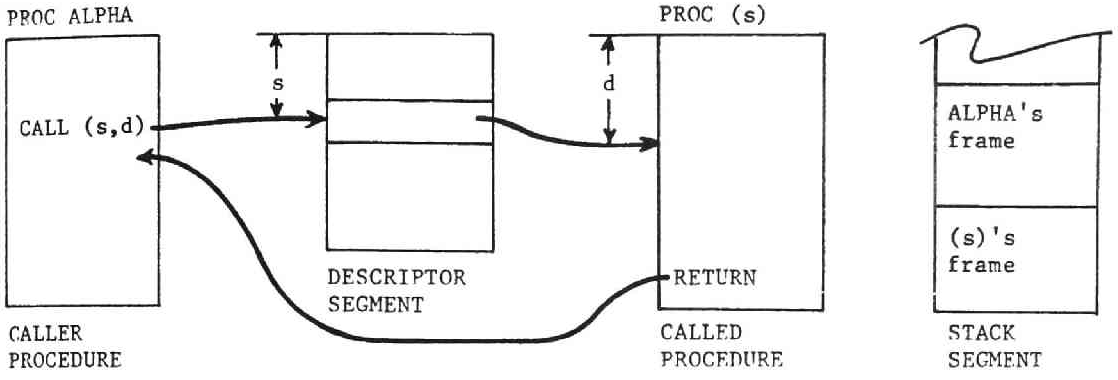
process generally has nothing to do with the event which caused the interruption, while the status of the process related to the event need to be changed. The alteration of the process status is executed by the process currently running on the CPU.

An interrupt is a hardware implemented "call" [ORG2] to a procedure which alters the process status, and upon completion of this processing control is "returned" to the interrupted place. Sometimes, control is once preempted by other processes, and then returned. Such a call and its consequent return are entirely transparent to the computation of a process which is interrupted.

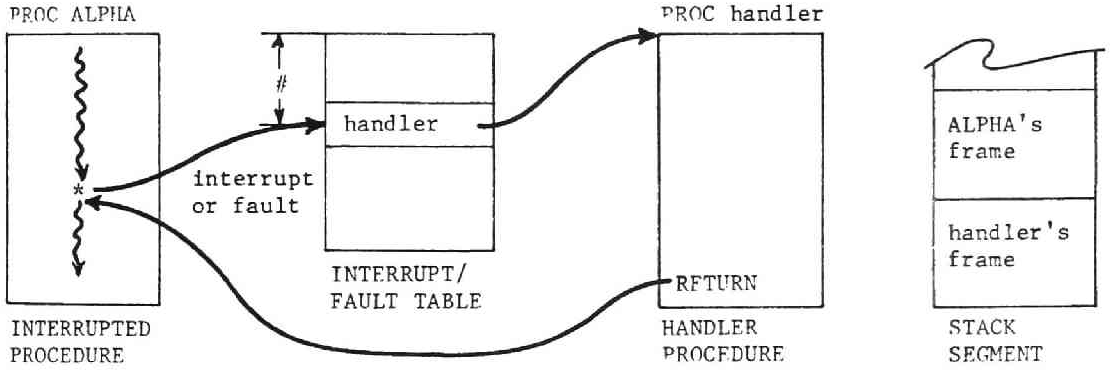
A fault is the hardware mechanism to detect economically a condition or an event which is caused by the execution of some operation and is difficult to detect economically by software methods, or is caused by the internal status in the CPU or the memory. That is, a fault is logically a hardware implemented call [ORG2] to an event handling procedure by the currently running process itself, otherwise the detection sequences of these events need to be programmed. Slightly different point from other calls is that there are two types of event handling procedures, one is statically defined and the other is dynamically defined at the time of fault detection. The details are left behind.

Anyhow, an interrupt or a fault is, by their nature, a call to a procedure, and may result in a (normal) return or a non-local go to.

Therefore, pushing and popping a stack frame just the same way as a usual call and a return respectively, processing of an interrupt and a fault can be accomplished in the same manner as a usual procedure invocation [ORG1].



(a) Usual call and return.



(b) Interrupt and fault handling.

Figure 11.5 Normal call and interrupt or fault handling.

So far, interrupt and fault handling has indiscriminately been executed by privileged procedures in many computers. This method is not adequate in order to make the structure of operating system clear and to improve the integrity and the reliability of the system by minimizing the size of the security kernel. Based on the basic rule of information protection - need to know [SAL3] - , only the matter which has relation to the protection issue which can only be processed in the privileged domain should be processed in that domain. For

example, a linkage fault used in dynamic linking has been processed by the supervisory procedure. Links are, however, the data which belong to the domain where the process is currently running. That is, as it is obvious that they are processed in the usual non-privileged domain when they are linked by the static linker, there is no problem which has relation to protection, and no privileged processing is needed.

Anyway, by interpreting that both interrupt and fault are invocations of procedures without any exception, one can simplify and unify the structure of inter-procedural communications and further the structure of software systems.

6. Invocation of Interrupt and Fault Handlers

When an interrupt or a fault occurs and its relevant handler is invoked, it needs some considerations to carry out Steps 1.1 A. to D. executed on a usual call. They are modified as follows:

A. To determine the target handler, and
B. To acquire appropriate status to run the target handler.

6.1 Determination of an Interrupt or a Fault Handler

There are two cases to determine an interrupt handler. One is given a pointer to an interrupt handler for an interrupt. The other is given a queue of processes which require to be notified an occurrence of an interrupt.

There are also two cases to determine a fault handler. For an event which the system is responsible for handling, a pointer to the handler is given. For an event

which a process is responsible for handling, it is necessary to search for a handler in the place relevant to this process, given an event name as a key [MSPM].

These are principles; however, there are some exceptions. A quit signal from a console operator, for example, is an interrupt, but its meaning is that "an external cooperative process" of human being wants to cause a fault condition. A linkage fault is a process fault, but needs the processing on the system-wide conventions, thus the pointer to the linker is given.

In case that a pointer to an interrupt or fault handler is given beforehand, pointers may be listed in the interrupt and fault handler table.

If a queue of processes which require to be notified when some interrupt occurs is associated with its interrupt, such an interrupt is also handled in the same way as a usual procedure call, designating a queue handling procedure in the handler table described above.

Handlers for many process faults are often changed during the execution of a process. This is just the same situation that a condition handling procedure of PL/I is defined for a condition name by using an ON statement, and at another point such a pre-defined procedure is invoked by using a SIGNAL statement. That is, process fault handling is the generalization of the condition handling mechanism, and such fault handling is accomplished, designating a fault type as a condition name (see Figure 11.6) [MSPM], [COR1]. The fault type is one of the process status and is saved in the stack frame of the faulted procedure.

The procedure "signal" searches for and invokes a handler registered in association with a condition name in the faulted domain.

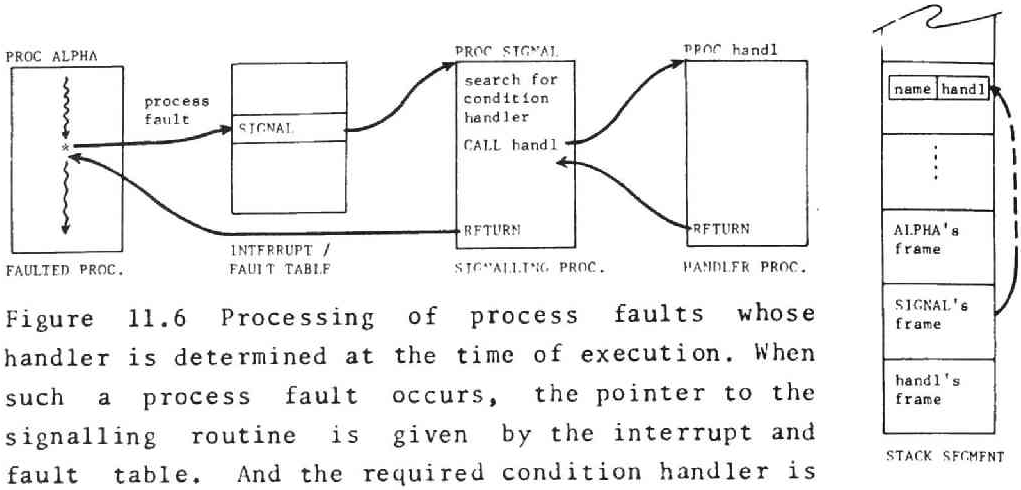


Figure 11.6 Processing of process faults whose handler is determined at the time of execution. When such a process fault occurs, the pointer to the signalling routine is given by the interrupt and fault table. And the required condition handler is searched for in the stack frames of the faulted process in just the same way as a SIGNAL statement.

Thus, putting the pointer to the signal procedure in the handler table, these faults can be handled just the same way as other interrupts and faults.

It may be considered that the interrupt and fault table is a fixed table except in a special case, thus only one table is enough even if the system is running in multi-programming operation. On the contrary, one descriptor segment which describes the address space of a process is needed for each and every process.

To get the required handler from the interrupt and fault table is logically equivalent to the situation that a usual procedure call is accomplished by referring to the descriptor segment. That is, the conventional segmentation mechanism can be applied for the

implementation of interrupt and fault handling by preparing an additional descriptor segment for interrupts and faults and a base register which points at the beginning of that table, and by switching the base registers as the occasion demands. (See Figure 11.5.) The interrupt handler table prepared in the stack trunk of Burroughs 6700 is one example which realized this function.

6.2 Status for Handler Execution

It is also able to apply the decision function of the execution status and the domain in the segmentation mechanism for the execution of an interrupt of a fault handler. In this case, the usual descriptor segment and the one for interrupt and fault handling are pointed at by the separate base registers, and there is no fear to misuse these tables. Thus, the access rights can be assigned independently each other. Care must be taken from the viewpoint of protection not to invoke a less privileged handler when a process is interrupted or faulted.

In case that the ring protection mechanism is incorporated, the status to run an interrupt or fault handler is determined by the ring number r_x and the kind of procedure, supervisory or non-supervisory procedure. The problem is the relation between the faulted or interrupted ring number r_y and the target executing ring number r_x. It is assumed that only inward calls are permitted. Therefore, the condition

$$r_y \geq r_x$$

is needed. Otherwise, a less privileged interrupt handling procedure is "called" from a more privileged ring. This is not agreeable from the viewpoint of the information protection.

The domain where a handler associated with some interrupt is executed is essentially independent of the priority of that interrupt; however, there would be no obvious inconsistency if an interrupt of higher priority is considered that it has higher protection level. All of the interrupt handling is executed in the supervisory mode in many systems, and there is only one protection level in this case. The above condition reduces the system to one level at the most limiting case. In case of an input and output interrupt which usually needs prompt attention, the protection level is also high because of the information protection. And there would be no obvious inconsistency when the value of r_x is associated with the priority of an interrupt. Even if interrupts whose ring number of execution are greater than the current ring number ($r_y < r_x$) are inhibited, it would not cause any inconvenience.

Fault handling is a procedure invocation which is essentially executed by the current process itself and, therefore, it is rather natural that the current ring is not less than the target ring, that is, $r_y \geq r_x$.

To summerize, the condition $r_y \geq r_x$ can be assumed for most interrupts and faults or can be imposed without inconvenience, and the problem of the executing ring is solved. Registering ring brackets in the handler table

just the same way as the descriptor segment, one can control the domain switching in the same way as the usual domain switching.

Here are some comments about interrupt and fault handling.

1. The interrupt and fault handler table is referred to by the address formation hardware only when an interrupt or a fault is detected. Therefore, if a different ring bracket from the descriptor segment is given, it cannot be improperly used for other purposes.
2. Usually, a call bracket in a ring bracket of an interrupt or fault handler includes the maximum ring number of the system. Thus, whatever executing ring is interrupted or faulted, it is able to "call" the required handler. Further, the ring bracket in the descriptor segment is different from the one in the handler table, but there is no danger to be improperly used.
3. The signal handler is a utility procedure whose ring bracket is (0, 7, 7), and is executed without any ring change. This is natural because process faults are logically just the extension of usual processing of the currently running process.

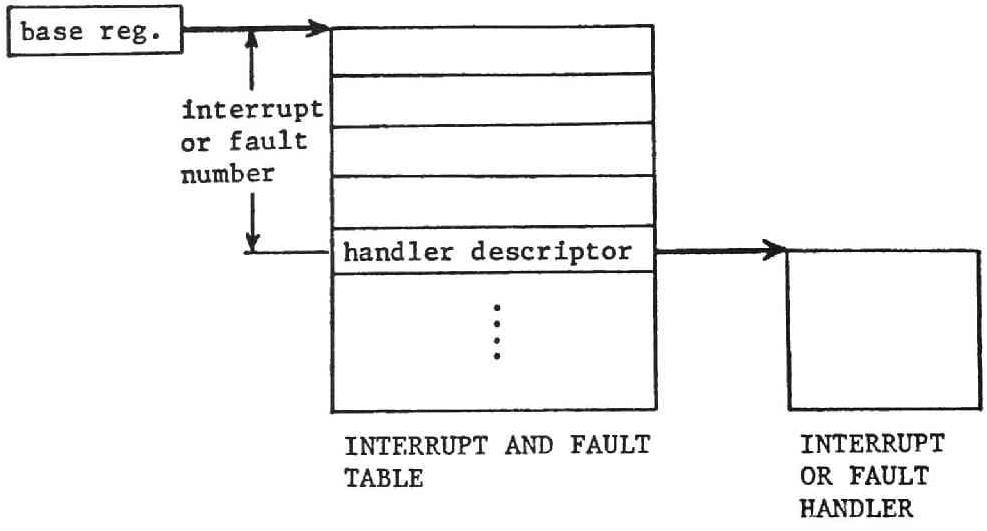
7. Interrupt and Fault Table

The interrupt and fault table acts as the descriptor segment only when an interrupt or fault handler is "called". This table is indexed by interrupt and fault number assigned to each interrupt and fault. Each entry of this table holds the following items:

Pointer to the handling procedure,

Segment length,
Access privileges, and
Type of the handling procedure including the addressing mode.

The following figure shows the construction of an interrupt and fault table.



pointer	length	access cont. information	type

HANDLER DESCRIPTOR

Figure 11.7 Interrupt and fault table.

A supervisory procedure can only be run in ring 0, which can execute such privileged instructions as doing input and output or changing the ring number.

The absolute addressing mode is used to form address directly without using the descriptor segment in such special processing as machine malfunction or initiation fault handling. Switching to the two-dimensional addressing mode from the absolute addressing mode is taken place by executing a mode switching instruction, or

by an interrupt or a fault which is processed in the two-dimensional addressing mode.

8. Masking Interrupts and Faults

It is necessary to inhibit and mask another interrupts and faults while the process status is reserved. In case that the hardware status reservation and stack pushing mechanism is incorporated, such inhibition is not necessary. But such an interrupt or a fault whose executing ring number is greater than the current ring number, can not be processed, because they would result an outward call which must be inhibited. Such an algorithm is implemented by hardware or firmware better than by software.

9. Processing of Interrupt and Fault Status

There are two kinds of processor status, one relates to usual processing and the other relates to protection control. Table 11.1 summarizes this relation. The saving of the contents of the instruction counter, the pointer registers, the general registers and the domain register is required upon a usual procedure call, and the restoration of them under the condition not to give surplus privileges is required upon return.

There remain other status in the CPU. They are the status of instruction execution, address formation and pipeline operation, the mask for interrupts, the interrupt register and so on. The larger the computer becomes the more delicate the control becomes. These status should also be saved and restored in case of interrupt and fault handling.

	USAGE		CALL/RETURN		INTERRUPT and FAULT		COMMENT
	PROCESSE'S OPERATION	PROTECTION	SAVE	RESTORE	SAVE	RESTORE	
INSTRUCTION COUNTER	*		*	*	*	*	
POINTER REGISTER	*		*	*	*	*	
OPERAND REGISTER	*		*	*	*	*	
INSTRUCTION REGISTER	*				*	OCCASIONAL	
DOMAIN REGISTER	VERIFY ONLY	*	*	CONDITIONAL	*	CONDITIONAL	
DESCRIPTOR BASE REGISTER	*	*	*	PRIVILEGED MODE	*	PRIVILEGED MODE	
EXECUTION MODE		*					INCLUDED IN SEGMENT DESCRIPTOR
EXECUTION PHASE		*			*	CONDITIONAL	CAN RETRY
ADDRESS FORMATION PHASE	FAULT HANDLING	*			*	CONDITIONAL	CAN RETRY
DOMAIN SWITCHING PHASE	FAULT HANDLING	*			*	CONDITIONAL	
PIPE-LINE CONTROL					(*)	(*)	MAY DISCARD
INTERRUPT MASK	INT./FAULT HANDLING	*			*	PRIVILEGED MODE	
INTERRUPT REGISTER	INT./FAULT HANDLING	*			*	PRIVILEGED MODE	

Table 11.1 Saving and restoring of the status in the CPU.

9.1 Point of Issue for the Saving of the Status

It is required to save the status of the CPU in a stack frame of the current process. In case that demand paging is employed, this saving might cause a page fault, and therefore, it is needed to take some countermeasures such as saving the status in the hardcore stack first which belongs to the process but is concealed from it [MSPM], and then moving them to the stack frame, or allocating the stack in a contiguous area lest a page fault should caused.

Reception of an interrupt and a fault has been processed in the privileged domain so far because the most sensitive information that controls protection is also included in the status to be saved. This part is regarded as the interface between the hardware of the CPU and the system software. Assuming that the ring protection mechanism is employed, the ring bracket of this part is (0, n, 7), where n is the target ring number that the target handler has. Execution of this part begins in ring 0, the most privileged domain, saves the status, determines the target ring, and invokes the target handler in the target ring. Thus, this is an exceptional part from the viewpoint of usual procedure invocation. Such processing is permitted only because it is executed in the privileged domain; however, this violates the system convention of procedure invocation. Thus, this part should be regarded as a part of the processor, and then there would be no contradiction in the previous discussions.

9.2 Point of Issue for the Restoration of the Status

If an interrupt or a fault which doesn't require the privileged processing is processed in the non-privileged

domain, how can the sensitive status be restored then?

One way to perform this is to ask a privileged procedure to restore or to validate them [JAN1] [MSPM], but this method results in that the protection issue affects the program logic, and that the return sequence of handlers is also changed. These things are not favourable for our purpose. An interrupt or a fault handler processes within the privileges of the executing domain, and is permitted to restore the status within the scope of its ability, that is, it is not permitted to restore status which give more privileges to the process. On the contrary, the handler has the right to give less privileges to the process.

If such sensitive status which need more privileges to restore don't exist in the current domain, return from the handler would be performed successfully by the usual return sequence. We will study this situation individually in the following:

Input and output interrupts:

These process the raw information, and in many cases, it is required to alter the process state. These processing should be performed in the privileged domain.

Operator interrupts:

The requirement of response time is not strict, thus it is able to delay such interrupts until no sensitive status remain.

Machine check faults:

Usually, they can be processed only in the privileged domain.

Faults in the course of address formation:
(linkage fault)

It is only needed to retry the faulted operation.

(Segment/Page fault)
They are processed in the privileged domain.
Access violation faults, and domain switching faults:
They are processed in the privileged domain.
Faults caused by arithmetic or logical operation:
They have nothing to do with protection.

Thus, return from the handler may be performed by the usual return sequences.

10. Necessary Faults

Programs cannot efficiently be executed if the checking of all the conditions is needed in programs each time a statement is executed. Hardware implementation of such checking is necessary to ease programming and to execute programs efficiently.

This hardware facility is referred as a fault. This section discusses the kinds of faults and the processing policies which are desired to form an address space.

10.1 Faults Caused by Errors in Hardware and Software

Faults caused by a machine malfunction, an illegal instruction, or an illegal operand imply that there happened an errorneous operation and it is meaningless to continue the current processing.

These faults would need no further explanation.

10.2 Faults Caused by Arithmetic Operations

Some faults are caused in the course of arithmetic operations in the CPU. These faults would only need a short explanation.

Faults caused by:

Accumulator overflow or underflow by add, subtruct, multiply, and divide operations,
Loss of digits caused by floating point add, subtract operations, and
Zero divide

are process faults, and it is needed to execute a procedure which is designated by the current process or a default handler which is designated by the system.

10.3 Faults Caused by Address Formation

Some faults are caused in the course of address formation carried out by the CPU when an instruction is fetched or an operand is referred to. These faults are caused by the following reasons:

A. Access to a link which has not been snapped yet (linkage fault)

This is a process fault but needs processing on (sub-)system-wide conventions. (See the section of dynamic linking.)

B. Access to an inactive segment (segment fault)

This is a system fault and it is necessary to activate this segment and place (the page table of) this segment in the main memory.

C. Access to a missing page (page fault)

This is a system fault and needs page loading.

D. Access to static storage which has not been established yet (static storage fault) [JAN1]

This is used to initiate and establish a static storage space needed to run a newly called pure procedure. This is a process fault but needs processing on (sub-)system-wide conventions. Its processing is to copy the template of the static area relevant to the procedure into a static data

segment, and set the private pointer table.

E. Access to the stack bottom or the bottom of the queue

10.4 Faults Caused by Access Control Functions

The following faults are caused by the access control functions which relate to the access rights of the target segments:

A. Access violation

This is a system fault, which requires to deny the access and to halt the process execution.

B. Wall crossing for which software intervention is necessary

This is a system fault which requires to switch the domains. No SVC instruction is needed because the domain switching is taken place when the call instruction is executed.

C. First time / every time reference fault

This is used to detect first time or every time a specified link is referred to.

10.5 Faults Intended to be Used by a Process

Some faults are caused by a process itself intentionally in order to be used for its processing such as obtaining a check point dump in debugging. They are sometimes called simulated faults, which are detected as an exception of a certain condition such as a segment number.

11. Where to Set Fault Conditions

Some faults are able to be detected when a pointer which needs to cause a fault is used. The followings are

a list which shows where to set fault conditions for various kinds of faults:

A. In a segment descriptor
    Segment fault
    Access violation fault
    Crossing wall fault

B. In a link
    First / every time reference fault
    Linkage fault

C.1 In a pointer in a linkage offset table
    Static storage fault

C.2 In a pointer in a page table
    Page fault

C.3 In a pointer in a queue
    Bottom of queue fault

CHAPTER 12
MECHANISM OF DYNAMIC LINKING - IMPROVED ALGORITHM -

The algorithm of dynamic linking has been discussed in Chapter 5. One of the reasons why it is difficult to implement dynamic linking is that it has been necessary to constitute the linker as a privileged procedure which has been related to fairly big parts of the supervisory kernel. However, as described in the previous chapter, it is desirable to execute the linker in the faulted domain in order to make the security kernel small and to improve the integrity of the system. That is, the processing of linking doesn't need the privileges of the security kernel, and the security kernel doesn't need the linker for its functions. This is obvious from the fact that conventional linking is performed in the non-privileged state by a service program such as a linkage editor. Therefore, dynamic linking should not be executed in the privileged state but in the non-privileged state. Here, the algorithm of the linker is developed to meet this requirement.

1. Removing the Linker from the Security Kernel

It has been shown in Chapters 6 and 7 that problems with regard to information protection arise when a process gains more privileges than the current one.

A linkage fault which requires to switch domains is, in fact, caused only by a domain crossing call to a procedure which is required to execute in a different

domain. The kinds of access to a usual segment are read, write and execute. Read- or write-access to a segment to which it is not permitted to get access in the current domain is nonsense. And it is obvious that no linkage fault occurs when control is returned to the caller.

In case of the ring protection mechanism, an inward call is the only thing that may cause a linkage fault which needs special handling. That is, read- or write-access to a segment whose read or write bracket includes the current ring has no problem. The linker can refer to such a segment in the faulted ring. Further, an outward transfer of control is restricted to an outward return, which doesn't cause a linkage fault because the segment returned to is the "caller" itself.

Now, let's consider the case that causes a linkage fault when a segment is called crossing the domain walls.

When a linkage fault is caused, the linker is invoked in the faulted domain. The linker, then, obtains the segment name of the target procedure using the pointer in the link, and asks the file subsystem to search for this segment in the directory hierarchy.

Next thing to do is to get the entry offset looking up in the global symbol table of the target segment. The problem arises here. Because the intended call is a domain crossing call, the called segment cannot be referred to by the linker executed in the caller's domain. The most easy going way to solve this problem is to execute the linker in the privileged domain which has the capability to get access to the information which can be referred to in both the faulted domain and the target domain. It is, however, desirable to minimize the security kernel in order to improve the integrity and the reliability of the system. Thus, it is required that the

linker itself should be executed in the called domain.

The problem of removing the linker from the security kernel is discussed by Janson [JAN1], but the linker in the faulted domain calls the linker in the called domain via the gate prepared for this purpose in each domain in his method. However, taking account of the following facts:

1. Only a privileged procedure can determine the target domain, and
2. The function that determines the target domain has already been included in the mechanism which controls the domain switching associated with transfer of control,

there is entirely no necessity to implement the same supervisory function for the linker over again. The program logic of Janson's linker is affected by the protection issue, and is unduly difficult to understand. Moreover, preparation of gates is needed.

The scheme proposed here utilizes the domain switching mechanism that the system has already had, and clarifies the algorithm of the linker further.

Here, the linker is divided into the following two functional modules:

1. Segment linker
   - A. To get the target segment name by the pointer in the faulted link.
   - B. To ask the file system to search for the target segment. And to assign a segment number to the target segment.
   - C. To set only the segment field in the link.
   - D.1 In case that an entry name is given, to set the condition in the link to cause the second linkage fault.

D.2 In case that the value of entry displacement is given, to establish the link.

E. To return control to the faulted place.

2. Entry linker

A. To get the entry name by the pointer in the faulted link.

B. To get the value of entry displacement that corresponds to the entry name by looking up in the global symbol table associated with the target segment.

C. To set the displacement field in the link.

D. To return control to the faulted place.

When the first linkage fault occurs, the segment linker is invoked in the faulted domain. The segment linker executes the steps of 1., and then, control is returned to the faulted place. In case that an entry name is given, reference to the target segment via the link causes the second linkage fault because of step 1.D.1.

This time, however, the target segment has already been determined, and address formation process proceeds to the stage to determine the target domain and to switch the domains. Thus, if the second linkage fault is caused in this state, it results in that the entry linker is invoked and executed in the required domain to refer to the target segment, and there is no inconvenience to execute the steps of 2. That is, the required domain switching to refer to the target segment and to get the entry displacement is realized by causing the second linkage fault.

A linkage fault is considered as a hardware fabricated call [ORG2] to the linker, thus a link which causes a linkage fault corresponds to an instruction which invokes the linker. That is, the first linkage

fault invokes the segment linker, and the second linkage fault switches to the target domain and invokes the entry linker.

The return from the entry linker (2.D) switches back to the original domain, and resumes execution of the faulted operation again. Figure 12.1 shows the outline of this scheme, and Figure 12.2 shows the stack frame manipulation, assuming that call stack like ALGOL is used.

If the overhead of the domain switching is big, step 2.D can be modified as the following because the domain has already been switched to the target domain:

2.D.1 In case of a procedure call, to go to the entry and to complete the required call.

2.D.2 In case of read- or write-access to a data segment, to resume the execution of the faulted procedure.

It is, however, favorable to constitute the linker procedure just in the same structure as the usual closed subroutine, which is invoked by a call instruction and returned by a return instruction, from the viewpoint of system integration.

The scheme described in this chapter is natural, and the algorithm of the linker is clear, as the domain switching problem is resolved by using the domain switching function included in the protection mechanism. To cause two linkage faults for a link might result in inefficient operation so long as the conventional fault handling is employed. This fact, however, stems from the defects in the current fault handling mechanism, and the fault handling mechanism discussed in the previous chapter resolves this problem.

2. Processing of the Entry Linker

Links to which a procedure refers and their link definitions belong to the domain where this procedure is executed. The entry linker is sometimes executed in a separate domain from the one where the segment linker is executed, thus the problem arises about the reference to parameters for the processing of the entry linker.

In case that the ring protection mechanism is employed, such domain switching is required only when an inward call is executed. And there is no problem about the reference of parameters because all the information that belong to the outer ring can be referred to from the inner ring.

In case of a general protection mechanism, it is not permitted to refer directly to the information that belongs to other domains. This problem, however, can be

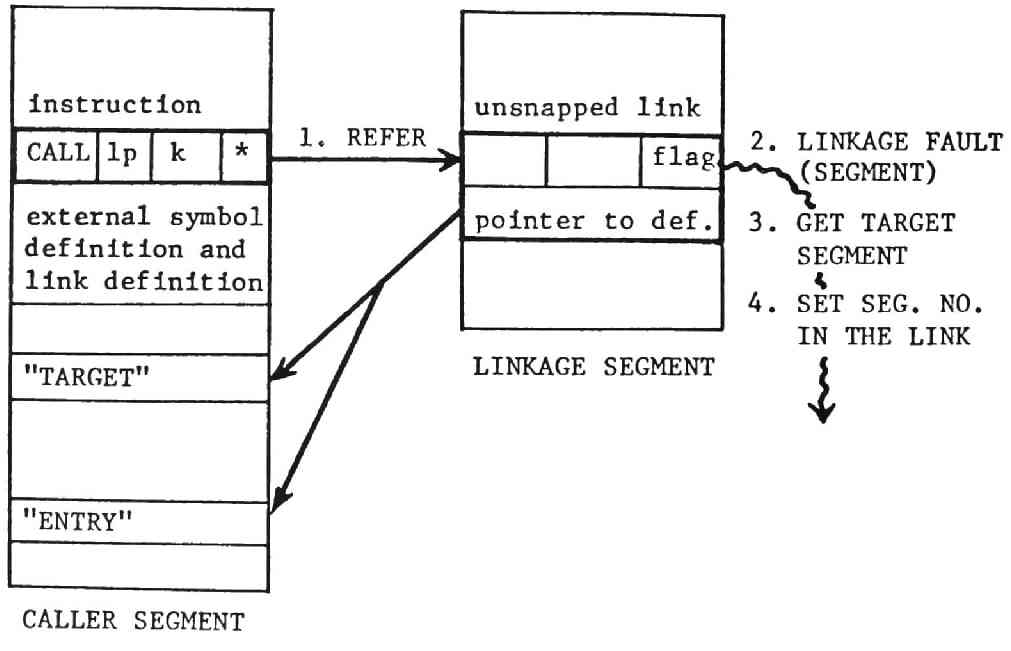


Figure 12.1 Mechanism of the linker which executes in the faulted domain.

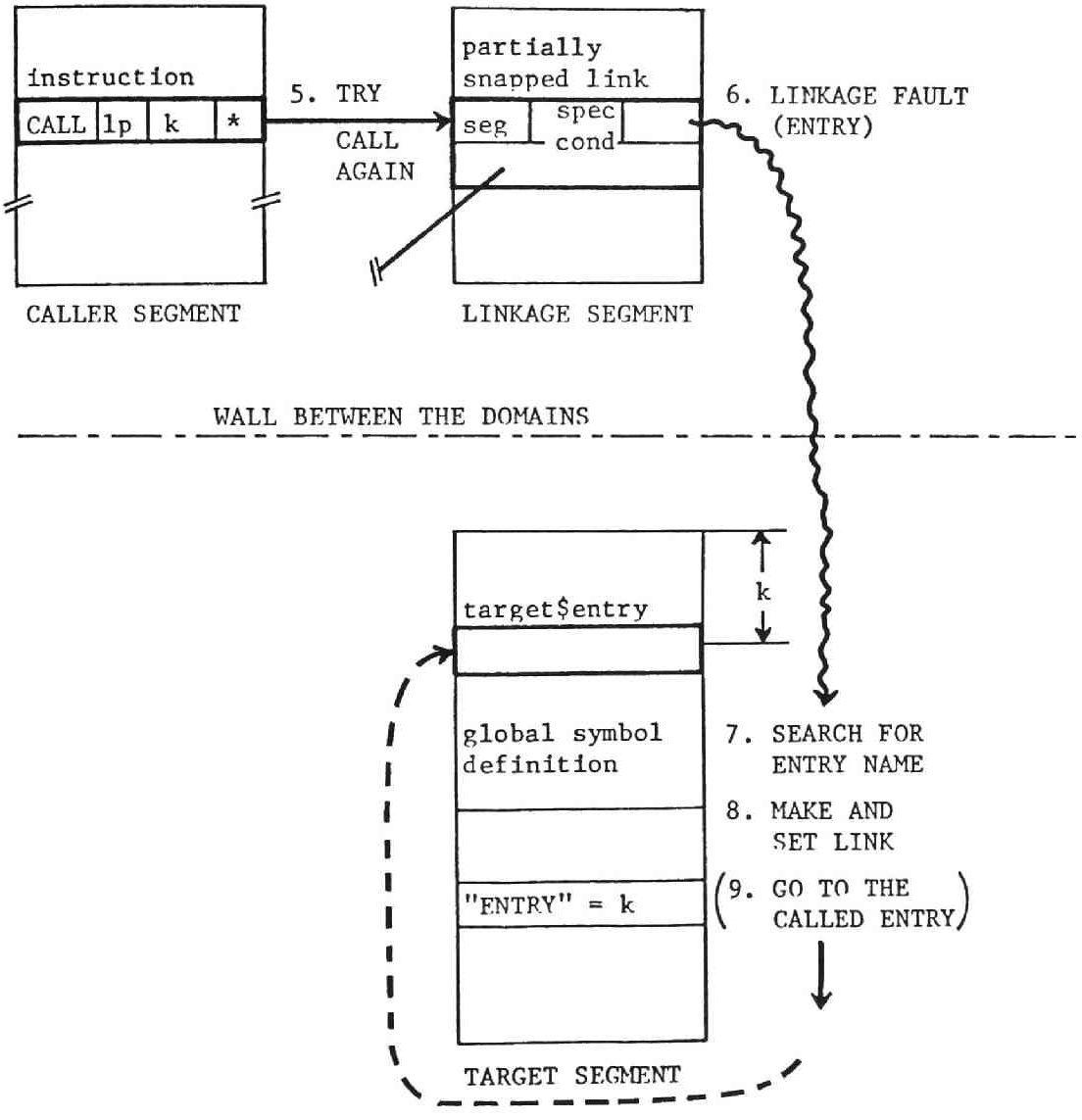
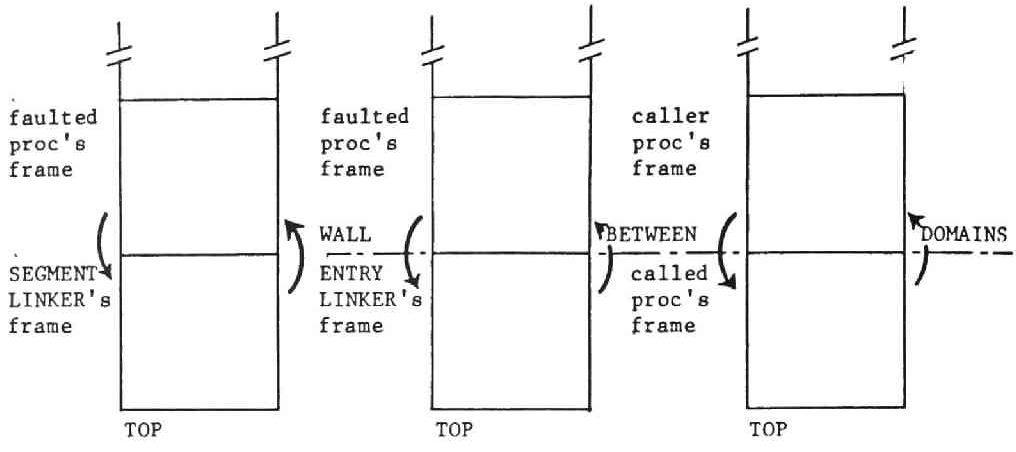


Figure 12.1 (continued)



(a) Invocation of the SEGMENT LINKER (b) Invocation of the ENTRY LINKER (c) Invocation of the target procedure

Figure 12.2 Picture of stack frames. Three stages of dynamic linking in case of domain crossing call.

easily resolved by the help of a supervisory function which simply transfers the necessary access capability for parameters. This problem is discussed later as the problem of the capability for the reference to the arguments.

3. Consideration on Performance

In Multics dynamic linker occupies 5% of program steps, 5% of the number of program modules, and 11% of gates of the kernel [JAN1]. Gates are the entrance into the security kernel and, thus, are the most sensitive and directly threatened part of the kernel. Reduction of program size of 5% and number of gates of 11% is a significant improvement for the reliability and the insulation of the security kernel.

The reasons why dynamic linking is not popular are the difficulty in the implementation and the batch-oriented method of system evaluation. Performance of many batch-oriented computer systems is evaluated solely by the executing speed of executable binary programs, and few of the excellent features that dynamic linking has are duly appreciated. On-line use of computer utility, however, enhances the excellent features that dynamic linking has.

To cause two linkage faults for a link might result in inefficient operation so long as the conventional fault handling is employed. This fact, however, stems from the defects in the current fault handling mechanism. Essentially, fault handling should be as handy as usual procedure invocation. This is the problem of the constitution of the interface between hardware and software rather than the problem of this scheme. These days, kernels of operating systems are sometimes tried to be implemented by firmware, and improvement of efficiency, especially in the part of hardware-software interface, can be expected in near future.

The excellent features of dynamic linking are never cancelled by the inefficiency of fault handling. The batch-oriented evaluation of computer systems gives little to on-line users. Dynamic linking excels static linking in the efficiency of program development and the efficiency of memory space. Dynamic linking delays linking of external references until they are actually needed to do so, thus it is able to realize flexible reference to the required information. Further, dynamic linking offers the direct addressing method independent of the physical storage location, and makes the program logics independent of system configuration, removing all

of the file I/O instructions from programs. External references which need to be linked with other segments occupy only a small portion, comparing intra-procedural references. And the execution speed via a snapped link needs only one extra memory reference. Thus, performance degradation due to dynamic linking is nominal. Dynamic linking is strongly hoped and recommended to become widely used.

CHAPTER 13
STRUCTURE OF SUPERVISOR

This chapter summerizes the method of constitution of an address space from the standpoint of the supervisor structure. Most of the early operating systems consisted simply of one big program. As system became larger and more comprehensive, this "brute force" approach became unmanageable. Eventually, the extended machine approach (Donovan) might be applied, that is, a computer system is considered as a layered computer. Each layer is the extension of the machine beneath its layer. The method of arguments is top-down, exhibiting layer by layer from the outermost to the innermost. The following is the list of the layers:

1. The address space manager
2. The segment manager
3. The memory space manager
4. Physical i/o subsystem
5. The process manager
6. The processor manager
7. The bare hardware interface
8. The bare hardware

The outermost layer accepts programs which refer to information symbolically and executes them directly. This layer is the very layer where programmers express their algorithms and operators require to the computer what they want. This layer is called the address space manager. The address space manager searches for the required segments in the file system, registers them in

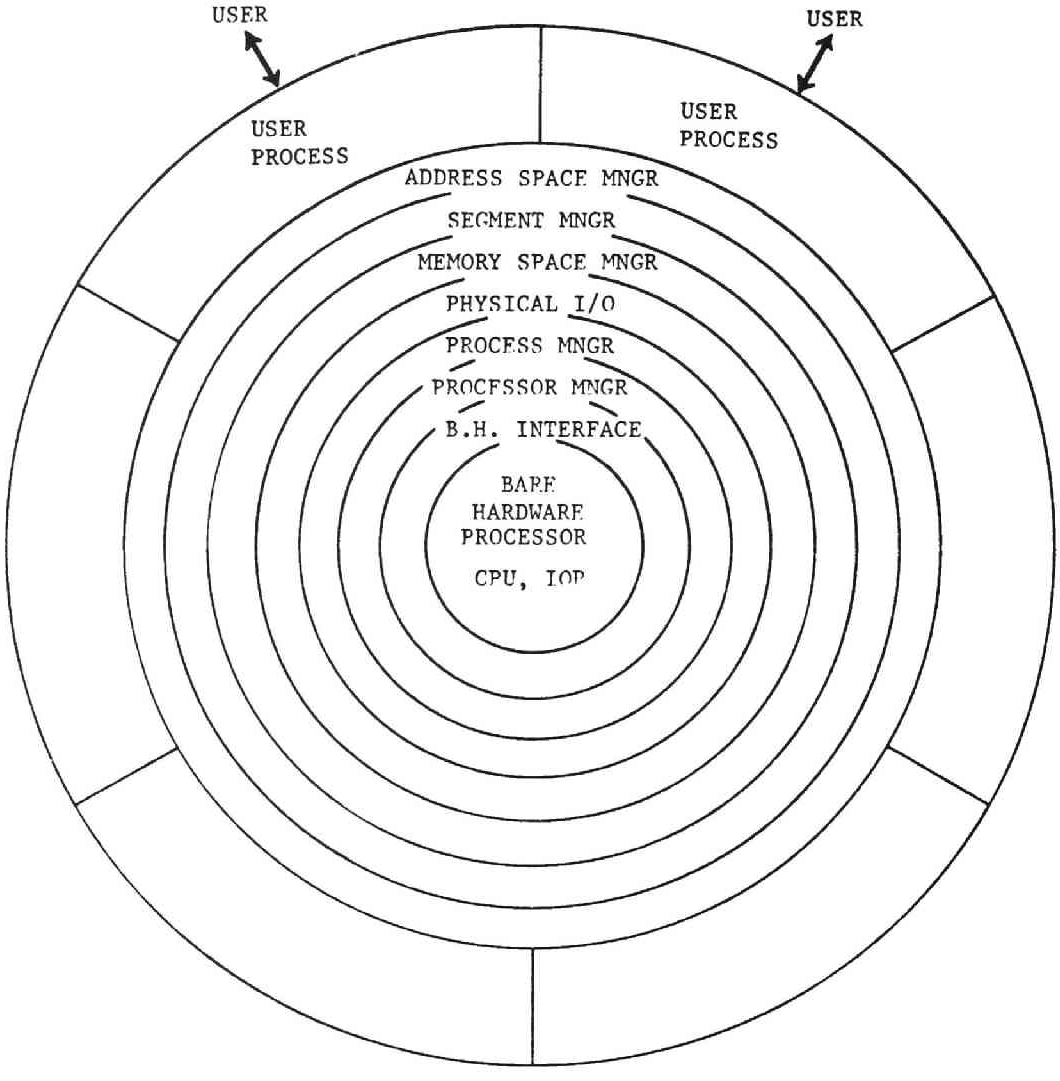


Figure 13.1 Layered computer system.

the address space of a process and links them to resolve references done by symbolic identifiers.

Next layer is the segment manager. When a required segment is located, next thing to be done is to open (activate) this segment to place this segment in the main memory and to validate address formation of the segmentation hardware.

When a segment is made open, the pages which are required are not immediately placed in the main memory. As the result of actual access necessary space is allocated in the main memory and/or in the secondary storage. This layer is called the memory space manager. The memory space manager manages the memory space in the main memory and in the secondary storage.

When all the conditions to execute a process are satisfied, the state of this process may be set from the ready state to the run state. This layer is called the process manager. The process manager has charge of all the state changes of the processes in the system lest there should be any contradictions.

Processes in the ready state are put into the ready queue, and one process is selected by appropriate criterions (mostly on the priority basis) and allocated the CPU to run. This layer is called the processor manager.

The innermost layer is the interface between the bare hardware and software. While the process executes the sequences described beforehand in the procedures, there would not be any extra problems. However, there happens such a situation that needs a special intervention of a software handler by an occurrence of an external asynchronous interrupt or an internal fault caused by an abnormal condition. The contents of such an intervention greatly differ according to the hardware used, but it is the common thing that at the time such an intervention is needed this layer touches directly the hardware status and then interfaces a handling procedure.

1. Address Space Manager

Programmers express their algorithms using symbolic references, and operators issue commands to execute their jobs to a computer in this layer. This layer consists of the following modules:

The dynamic linker,
The known segment manager, and
The directory manager.

1.1 Dynamic Linker

The dynamic linker is invoked by a linkage fault. Given a symbolic name, it invokes the known segment manager to check if the segment required has already been known to this process. If this segment has not been known yet, it invokes the file subsystem to search for a directory entry, assigns it a segment number, and makes a link to enable the CPU the required address formation. The dynamic linker is an execute-only procedure and has the ring bracket of (0, 7, 7).

1.2 Known Segment Manager

The known segment manager determines whether or not the segment referred to by a symbolic name has already been linked logically to this process, that is, a segment number has already been assigned to this segment . If a segment number has not been assigned to this segment yet, the known segment manager requires the directory manager to search for a segment in the directory hierarchy.

1.3 Directory Manager

The directory manager

1. Searches for a segment in the directory hierarchy, given a path-name of the required segment which uniquely identifies a segment in the file system,

and returns the location of the required directory or the directory entry,

2. Creates a directory entry in a directory, and
3. Deletes a directory entry from a directory.

As the directory manager handles all the information in the system including the one of the top secret whose ring bracket is (0, 0, 0), the ring bracket of the directory manager must be (0, 0, 5).

This manager is invoked from the procedures which search for, create or delete, a segment, and from the linker. The directory manager has nothing directly to do with the physical input and output operations.

2. Segment Manager

The segment manager activates a segment and connects it to the address space of a process, given a directory entry of a segment. The ring bracket of this manager is (0, 0, 0). The segment manager is invoked mainly when a segment fault is detected, and it is sometimes invoked by special modules of the directory manager in the address space manager.

After a segment is connected to the address space of a process, the memory manager is invoked. The segment manager itself may be put in the paging environment. It is determined from the standpoint of operation efficiency of the system whether the segment manager is placed in the main memory permanently or is put in the paging environment.

This module uses the active segment table as its data base and the active segment table includes page tables. Entries in the active segment table except the page table may be placed in the paging environment.

A page table cannot be paged because the decision whether or not a page is missing is undertaken, using the page table itself. Furthermore, a page table must be physically contiguous in the main memory because there is no mapping mechanism for the page table reference. Hence, it is rather simple and efficient to make all entries of the active segment table resident in the main memory. The active segment table and a page table in the active segment table are addressed in the two-dimensional addressing mode when they are processed by the supervisory procedures. Further, a page table is referred to in one-dimensional absolute addressing mode by the hardware address formation mechanism.

3. Memory Space Manager

Today's computer system utilizes various kinds of memory devices which range from very high speed one at high cost to very slow speed one at low cost. If memory space allocation is scheduled well, memory space of very big capacity is realized at fairly low cost [KAR1], [COR2].

The memory space manager takes charge of the dynamic allocation of various kinds of memory spaces, and multiplexes the lacking main memory space among processes in the system. The memory space manager includes several memory managers, but only one of which is discussed in this section.

The main memory manager controls the resource allocation of the main memory, one of the fundamental resources in a computer system. The main memory manager uses the memory map, which is created when the system is started, as its working database, and selects a page

swapping candidate according to the page replacement algorithm.

The actual page transfer from the secondary storage into the main memory or vice versa is left to the physical i/o subsystem beneath the main memory manager. The main memory manager invokes the process manager in order to minimize the idle time of CPU and to make maximum utilization of system resources.

The transfer of control between the main memory manager and the process manager must be efficient enough because this part is run quite frequently. The ring bracket of this manager is (0, 0, 0). This manager is invoked by a page fault and by special modules of the segment manager. The modules of this manager must be resident in the main memory because no manager except this one can resolve a page demand.

4. Physical I/O Subsystem

The physical i/o subsystem transfers a record from the main memory to the secondary file devices or stream devices and vice versa, given two record addresses or a record address and a device address of both the source and the destination. Physical data transfer is executed by the i/o processor of the input output subsystem, and the supervisory module of the physical i/o subsystem acts as an interlude to the i/o processor. The ring bracket of this module is (0, 0, 0).

The physical i/o subsystem is invoked by the memory space manager. This module must be resident in the main memory because no manager except the physical i/o subsystem can transfer a record from the main memory to the secondary storage and vice versa.

5. Process Manager

The process manager controls all the state changes of the processes in the system and maintains the ready list and the active process table. The process scheduling algorithm, which dispatches a ready process, is independent of the process managing algorithm but these two are often implemented together because of the system efficiency. The process manager assigns the processor to a process which is selected by the scheduling algorithm, invoking the processor manager. The ring bracket of the process manager is (0, 0, 2). The entries of the process manager may be invoked directly from the supervisory modules, and indirectly from non-supervisory procedures via the gate procedures. The process manager is invoked by a process, by a timer interrupt and by an i/o completion interrupt.

6. Processor Manager: Get Work

The processor manager gives a ready process the CPU and makes this process run. The ready process selected by the process scheduling algorithm is usually placed at the top of the ready list, assuming that this list is arranged in the order of priority. The layer of the processor manager comes beneath the layer of the process manager. The ring bracket of this module is (0, 0, 0). Let's consider the initiating process to analyze the relevant structure of the processor manager.

In the first step the "pre"-processor manager, which is usually a human operator himself, starts the hardware implemented bootloader routine to place the initiating

module in the main memory. This initiating module is the loader of the system, and it initially controls all the system modules and places all the resident supervisory modules into the main memory. The resident parts of the supervisory modules may be bound beforehand or linked at the loading time.

Any way these modules and a few specific supervisory processes created by the initiating modules are hereafter considered to have already "existed" in the system, and become the absolute root of the recursion. Modules and processes which are considered to have already "existed" in the system are the followings:

The processor manager and the memory area where this manager is placed, and at least one process which can create another process.

Thus, the processor manager may be regarded as the innermost layer except the following interface and the CPU itself.

7. Interrupt and Fault Interface

The interrupt and fault interface, which saves the machine status and invokes an interrupt or fault handler, should be implemented by hardware circuitry, but part or all of the processing are beyond the scope of today's computer hardware, and usually these are executed by software.

The processing of this interface is just an interlude to an interrupt or fault handler, and should be treated as a special "hardcore" interface to the hardware. Hence, the interrupt and fault interface is the layer which adheres closely to the CPU hardware. The processor manager comes outside of this layer.

As mentioned above, the primary function of an interrupt or a fault is a "call" to a handling procedure. And according to the contents of processing there may arise the problem of "protection". So far, these two points have scarcely been separated, and this has made the logics of programs complicated and inefficient needlessly.

It is necessary to unify and clarify the program structure no matter what a call, a call to supervisory procedures (so far a SVC instruction is used), a hardware implemented call (an interrupt and a fault), or a call to a non-supervisory procedure, is executed.

When this requirement is satisfied, this layer is not necessary and will vanish at all.

8. CPU

There are several processors such as the CPU and the IOP in today's computer. Usually, processors other than the CPU are hidden from users because they are directly related to the physical input and output. In this section only the CPU is discussed.

The CPU is considered as a set of "micro-procedures" which "emulate" the CPU instructions. And all the instruction executions call finally these "procedures". In this sense the CPU is placed at the innermost kernel.

The procedures in the CPU are execute-only. A non-privileged instruction is considered as a utility routine whose ring bracket is (0, 7, 7), that is, it is executable in any domain without the domain switching.

A privileged instruction has a ring bracket of (0, 0, 0) and has no call bracket.

A "micro-procedure" related to an interrupt handling

has ring bracket of (0, n, 7) where n is determined by the execute bracket of the target interrupt handler. This implies that an interrupt handling instruction is a hardware implemented gate procedure and essentially acts as an interlude to a target handling routine.

The CPU also has several working registers which essentially belong to processe's data segments. The access rights of data contained in the general registers are read and write, and the ring brackets of them are (r, r, r) where r is the executing ring number.

The status registers relate to the protection control of the system and hold such data whose access rights are read and write restricted only in ring 0, that is, their ring brackets are (0, 0, 0).

CHAPTER 14
OTHER ADDRESS SPACES RELATED TO A PROCESS

Even in a computer system where the information management and the memory management are unified and most information are managed as on-line files, there remain some pieces of information which must be transferred through the "channel" of i/o processing.

An i/o process simply relates to a logical segment, and its processing algorithm is also very simple and straightforward. In due course, the realization of an i/o process is much simpler as compared with the host process. However, care must be taken to constitute the address space of an i/o process. The address space of an i/o process consists of a segment which is being processed by the host process, and generally the contiguity of physical location of the segment, and even the existence in the main memory are no longer insured. The desirable constitution of the address space for an i/o process is to share the page table of the target segment in the paging environment, given the location of the page table and the offset in the segment as the arguments. In this case the paging facility may also be shared, that is, page fault conditions may be notified to the CPU and the relevant processing may be undertaken by the CPU. This will be easily realized only with small additional hardware circuitry.

This scheme is also applicable to the i/o process which supports page swapping. In this case, however, no notification of page faults is necessary.

1. External World to the Address Space

We assumed that all the information which was stored on-line in a computer system was able to be directly addressable. What else cannot be addressed directly? They are:

1. Streams which are created or consumed outside a computer.
   Typical examples of streams are:
   A. Users at on-line terminals [SAL2],
   B. Batch processing bulk i/o's,
   C. Other terminals such as real time controllers, and
   D. Remote terminals or remote computers.

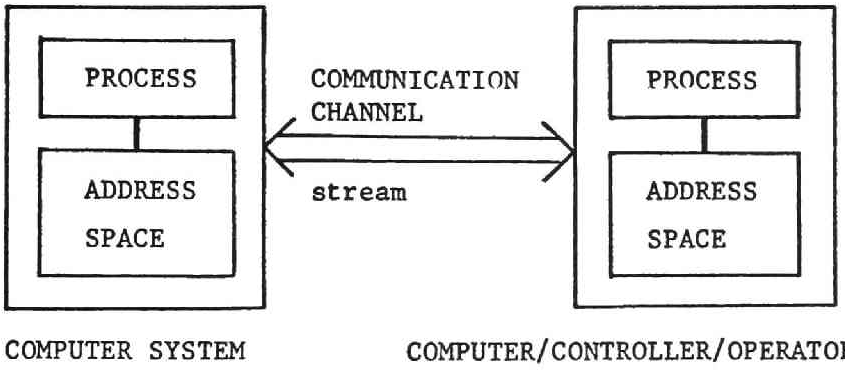


Figure 14.1 Stream i/o is required for the communication between two independent processes. Information is created and consumed at the time when it is created. Human beeing is also a kind of process.

In case of A., C. and some part of D. information is created or consumed at the time it is processed, so it makes no sense to make them directly addressable.

2. Bulk media which are brought to, or from, the other installations.
   Typical examples are:
   MT volume and dismountable disk volume.
3. Such information which is on-line but whose capacity of communication channel is too small to refer to it by direct addressing method.

2. Access of Databases in Computer Networks

It is desirable also in a computer network to unify the information management and the memory management. This section discusses the conditions to realize such a network. The following two things are attained by the unification of the information management and the memory management:

1. The unified and consistent management of information, and
2. Direct addressing to information independent of the physical location where the information is stored.

For the unified and consistent management of information unified control of segments must be enforced as explained earlier. Is it possible to force such control in a computer network? To do so it is necessary to be able to refer to the required information in comparable time and frexibility as if it is referred to in the main memory.

In a single system two or more processes are able to share the same segment. In a computer network also two or more processes which run in separate systems are able to share the same segment if there is common memory accessible to every system in the computer network. However, in such a configuration there is a physical limit in the length of the common memory bus.

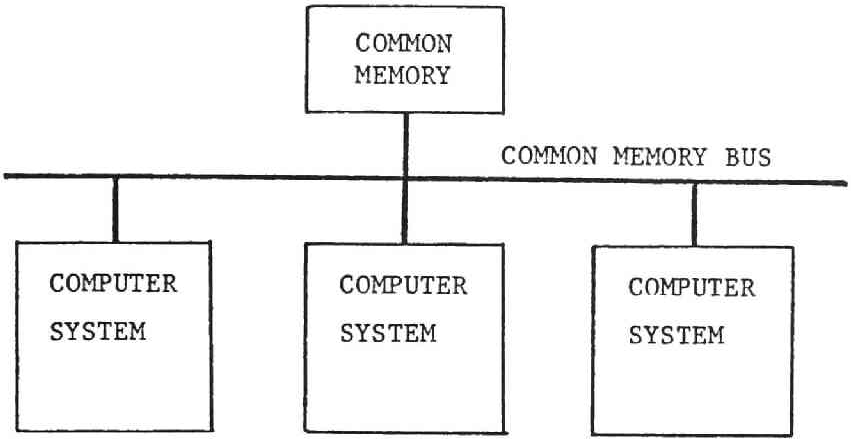


Figure 14.2 Closely coupled system.

In case of a computer network whose distance between two hosts is big, such information control is difficult, but in many cases the periods of modification of information are fairly long. In fact, there are many databases whose lives are almost permanent. In such cases direct addressing is realized by copying the necessary information beforehand. If the capacity of those segments is not so large, this approach can be employed in a computer network whose distance is very big. For example, it takes about twenty to two hundred seconds on forty eight kilo Baud communication line to move a segment whose size is one hundred kilo bytes to one mega bytes, but this movement is needed only once.

Another method is to process the required data in a distant system and to feedback only the results of the processing. This method works well if it is neither the case where two or more databases which are placed apart from each other are needed in a computation, nor the case where the capacity of the results of the computation is

bigger than that of the original database. This method is nothing more than a remote use of computing facilities.

If a computation requires two or more databases which are placed apart from each other or if a computation produces more results than the original database in volume, it is necessary to determine whether all of the information should be transferred or only part of it should be transferred. In the latter case access to the information is dominated by the access method of the target system, and it might unhappily be a traditional and inconvenient one.

CHAPTER 15
APPLICATIONS OF DYNAMIC LINKING

This chapter suggests some application areas suitable for dynamic linking, and gives several design issues for these applications.

1. To Switch Supervisors for Each Process

Sharing concurrently one computer system among more than one user began about 1963. What facility would be required in such a system?

In a system where development of operating systems or control programs is carried out, the bare machine is needed to each user. In a system where concurrent operation of different operating systems are required, the bare machine is also needed.

The virtual machines produced by such a system must simulate every feature of the bare machine up to the very details at the user's console. IBM's VS is such a kind of operating systems and has been contributing greatly to software development. Independent operating systems are used for each user in such a system; as the result, the efficiency of space utilization is not good.

What kinds of virtual machines are needed in a computer utility? In case of a computer utility the following conditions are strongly required:

1. To make it possible to execute an algorithm independent of a system configuration,
2. To minimize the degradation of efficiency due to the

process multiplexing, and

3. To be flexible enough to match with an indivisual application.

In this sense the important supervisory functions are not fundamental resources such as the CPU, the memory or the raw information but the command system and the file system on which user can freely build his own interactive systems or his own database systems.

If a computer system has dynamic linking facilities, these functions are operated under dynamic linking and can easily and arbitrarily be replaced or changed according to individual requirements of a user.

2. Toward System Independent Processing

An algorithm of a computation should be independent of, and should not be unduly influenced from, the system configuration. This section discusses the scheme to construct algorithms independent of a system configuration.

In early day's computers or even in today's small scale computers, supporting systems for programming are poor and algorithms are influenced by system configurations and by the details of devices. In such a system a great part of programming efforts are devoted to managing the format and location of storage rather than to the essential algorithm.

As computer systems and especially high level operating systems and data management systems advance, programmers are freed from a great many part of the problems with regard to the physical location and format when i/o operations are needed. However, problems about the allocation of programs and data in the main memory

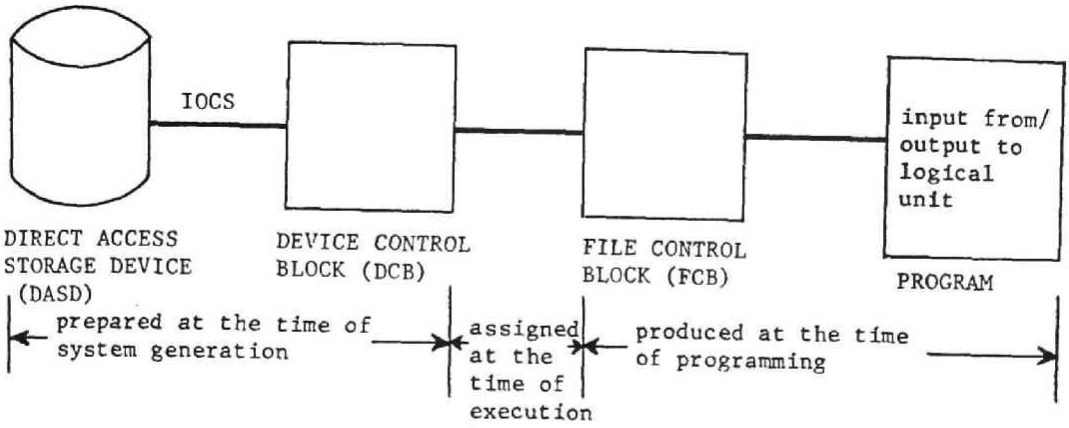


Figure 15.1 Device independent programming for sequential files. Logical units are allocated at the time of programming. Physical device definitions and driver routines are given at the time of system installation. Physical devices are assigned to logical units at the beginning of program execution.

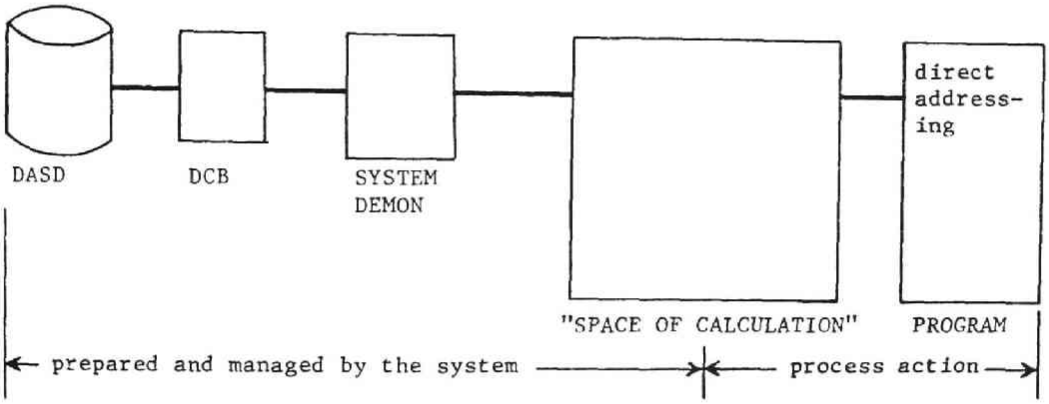


Figure 15.2 Direct addressing method. All on-line files are referred to as if they were placed in the main memory.

are still remained.

Virtual memory systems realized by paging or segmentation have resolved this problem and program structures are made clarified further.

The remaining problems are those which are concerned

with access to various databases and calls to various procedures from a program.

First, we will discuss a problem about access to a database. The most commonly used access method to a file is that a record is copied and processed in a buffer whose format is declared by a structure or array declaration beforehand. The problem of this method is that the linkage between a file and a procedure is not processed automatically, that is, the linkage between a file and a procedure must be "programmed" before the program execution, and it has nothing to do with an algorithm of a program to get access to file records.

Access to a record which has logical relations to an algorithm is the one to a stack and a queue which are basically types of streams with storage spaces.

The most desirable way to eliminate "file access" to a record in a file from the description of an algorithm is to adopt the direct addressing method independent of the physical location of a record. For the realization of this method, it is not enough, as mentioned above, only to realize a virtual memory system but it is necessary to link automatically the reference to the required information. Dynamic linking enables a user to address directly the required information wherever it is placed, unifying the memory management and the information management. It also removes the unnecessary restriction to recursion, which is strongly required not to impose improper restrictions on the expression of an algorithm, as it needs to make procedure pure.

When a program is executed interpretively, problems relevant to recursion and dynamic linking are automatically resolved. This is because:

(Recursion) When an interpreter finds newly declared

variables, it allocates them dynamically, and this is equivalent to pushing automatic variables into a stack, and

(Dynamic linking) "Linking" of procedures are always done dynamically, and this is nothing more than dynamic linking.

However, interpretive execution of a program is not always possible nor preferable because:

Execution speed of a program is slow,

Memory requirement for a program is bigger than that of compiler's object form, and

Appropriate interpreter is not always available.

The stream access method to such storage areas as a stack or a queue has the logical relation to the algorithm itself, so it cannot be eliminated, and it is desirable to prepare operations combined with such an access method to operands, which are placed in a stack or a queue, for arithmetic or logical operations. Especially, they are strongly required in language processing.

3. Application of Dynamic Linking to a Command System

In most command systems the menu of commands is fixed and it is difficult to add, change, and delete, some of them voluntarily. When addition, change, or deletion is required in a system which is constructed by the static linking method, it is necessary to link required procedures before the system is operated. This "preparation" process is called a system generation, which needs both time and labor.

If a system operates under dynamic linking, no such system generation is needed. What is necessary for an

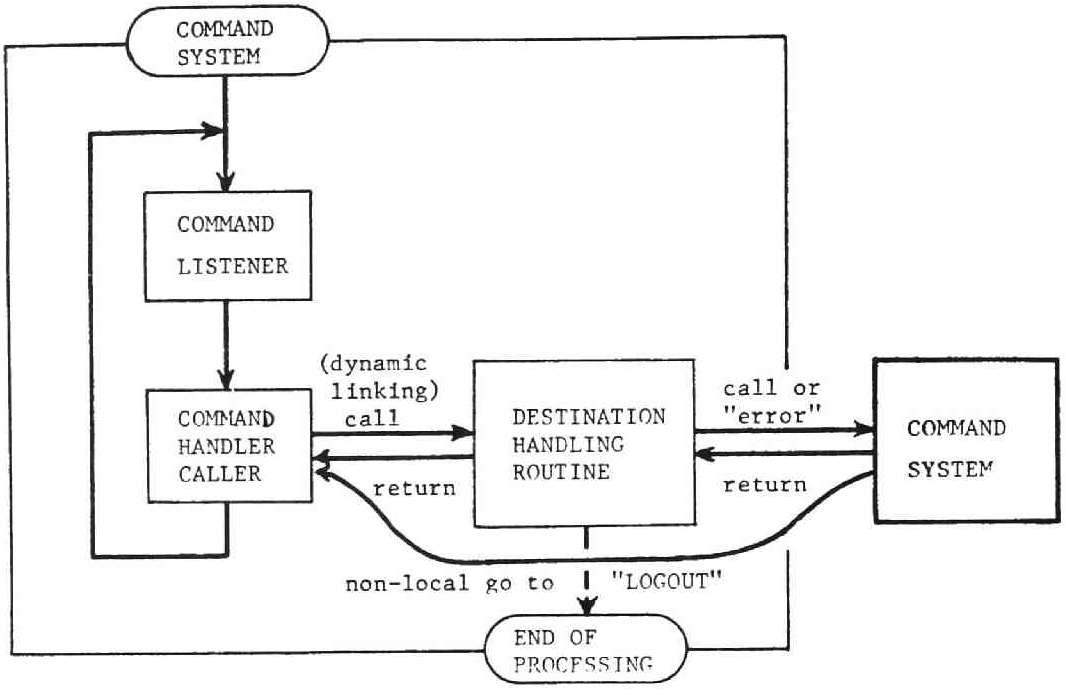


Figure 15.3 Application of dynamic linking to a command system.

operator to execute some program is only to show the system the program name as a command. The command system can execute this "command", just calling this procedure by the given name, and all the necessary linking is processed by the dynamic linking mechanism.

Thus, one subroutine which has just been created could be tested alone without completing all the programming that is required in case of a pre-linking system. Of course, this might cause a linkage fault for a segment which has not been created yet and then might pause execution until it is created.

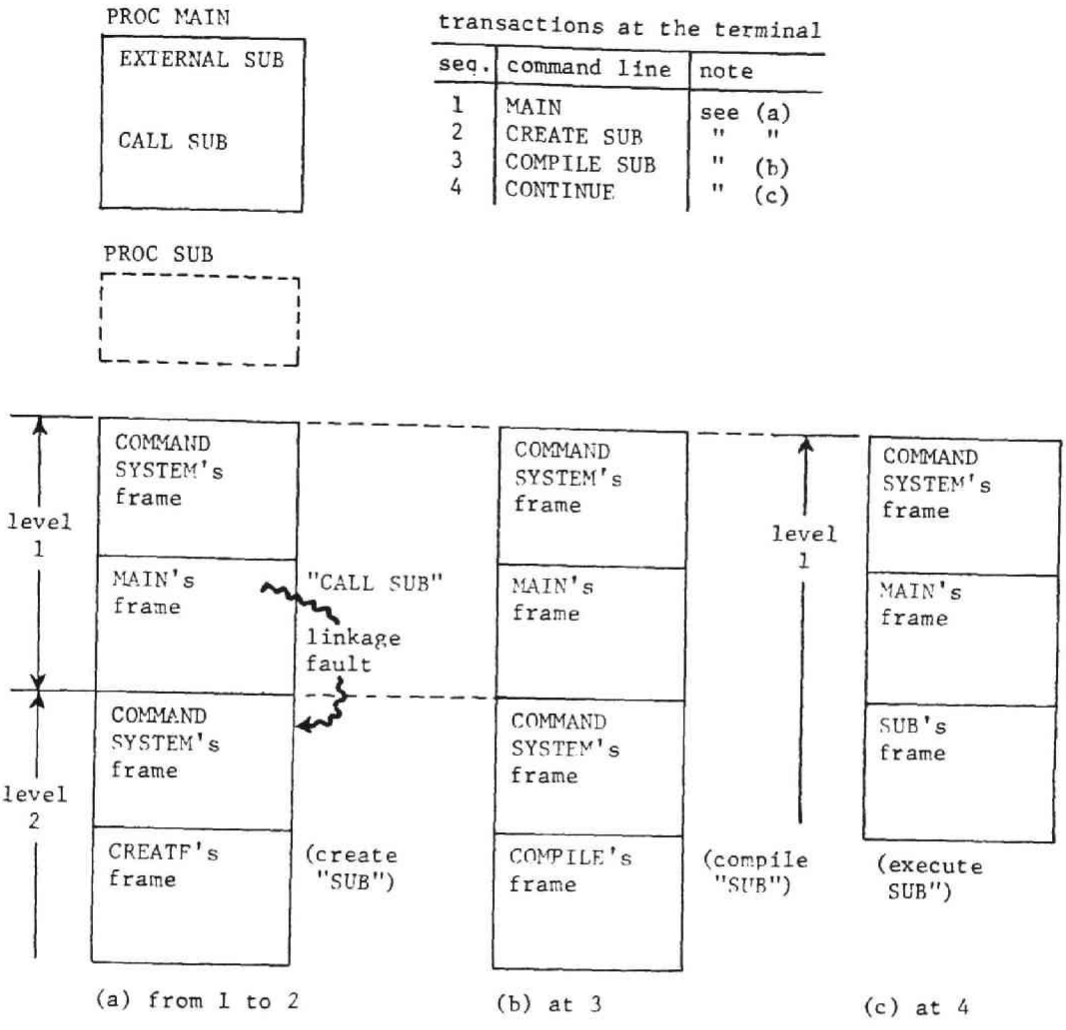


Figure 15.4 Multi-level command system. (a) An operator gives a command, "MAIN", to the command system, and then program MAIN is started. This program calls program SUB which has not been created yet. Thus, a linkage fault is caused and the command system is called again (level 2). It has a conversation with the operator and gets a command, "CREATE SUB". Thus, CREATE routine is executed to create a subroutine named "SUB". (b) Program SUB is compiled and becomes ready to be executed. (c) CONTINUE command returns control at the place where the linkage fault occurred (level 1) and the execution of MAIN resumes. MAIN succeeds in calling SUB this time, and SUB is executed.

Further, all procedures are pure and recursive, so nesting the command system itself does not make any problem and results in a flexible interactive processing system. It would be very convenient that, when undefined data or procedure is required during processing, one can suspend the execution of the process, fulfill the required conditions and then restart the processing.

One way to make such an operation possible would be to invoke the command system recursively from the dynamic linker which is processing the linkage fault which has been caused by requiring undefined data or procedure. Such a system cannot be realized by conventional systems. The only example of such a system is Multics [BON1] which realized dynamic linking.

CHAPTER 16
PROTECTION WITH MULTIPLE CAPABILITY LISTS

This chapter extends the discussions of Chapters 6 and 7, and develops a protection mechanism which uses multiple capability lists.

A process can refer to the required information by the capability [GRA2], [LAM1] that is defined for its state of execution. The scope within which the capability is the same is called a domain [GRA2], [SCH3], [LAM1], that is, a domain is defined by the capability. An address space of a process can be considered the closure of domains.

Usually, one calculation which a user intends is carried out by users own programs, by application programs or by service programs in some appropriately protected domains with proper capability. Generally, all the functions of the computer which these programs utilize are not created by these programs themselves, but some of them have already been prepared by the system modules or by other subsystems. Among these system modules there exist such manager modules that manage the system resources from the higher standpoints of management, and such superior modules in the kernel that supervise and control the state of execution and protection of processes in the system.

There exists the hierarchical relation between the capabilities of some protection domains. That is, the kernel that controls the protection essentially has the capability of the almighty, and can do everything. No

part in the system which the kernel is not able to manage can become the object of processing. The modules that manage the system resources come to the next level. Two layered systems which have had the supervisory mode and the problem mode have been employed to constitute such hierarchical protection domains so far, and the ring protection mechanism has been contrived and implemented in some computers as the generalization of the two layered system.

There also exists the exclusion relation between the capabilities of some protection domains. Lampson showed a case of two mutually suspicious subsystems [LAM1]. And proposals for realizing a protection mechanism which would satisfy such a requirement can be found [SCH2], but few of them are fully implemented actually.

1. Constitution of Protection Domains

1.1 Representation of Capability

The access capability is defined like this: "Subject S is given access privileges X to object O". And such a relation can be represented as an access matrix A [GRA2], with subjects identifying the raws and objects the columns (see Figure 16.1).

The entry A[S, O] contains the access privileges X held by subject S to object O. As there are usually numerous subjects and objects in a system, this matrix is too big in size with sparse entries to store and to maintain in a computer.

There are, however, four practical implementations three of which are suggested by Graham [GRA2]:

1. To store the access matrix A by raws, that is, to associate a capability list with each subject S (see

object

subject	FORTRAN	ARCTAN	CURVE	PLOT	SPECIAL	DATA1	DATA2	
JOHN	EX	EX	EX		R	RW		
MARY	EX	EX		EX	R		RW	
SMITH	EX	EX						
PLOT			EX		RW			
⋮								

Figure 16.1 Access matrix.

Figure 16.2).

2. To store the access matrix by columns, that is, to associate an access-control list with each object O.
3. To combine the capability list with the access-control list, that is, to create a capability list by storing entries (O, K) where O is an object name and K a key, and to associate a lock list with each object or the set of objects.
4. To combine the access-control list with the capability list, that is, to create access-control lists as in 2. and then to reorganize them into a capability list with entries (O, X) extracted from the access-control lists.

Method 1 is the straightforward way, but revocation of access privileges with regard to an object is difficult. This is because access privileges with regard to an object are generally held in more than one capability list for an individual subject, and moreover, they are sometimes transferred to the capability lists for other subjects. Thus, there would be a problem of operating

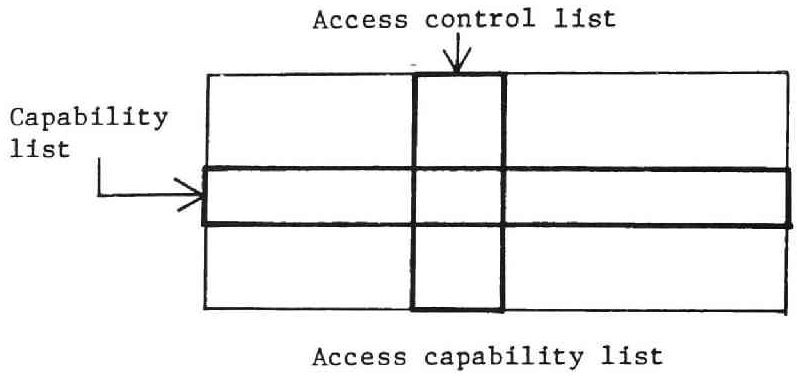


Figure 16.2 Implementations of access matrix.

efficiency if this method is applied for the objects in the file system. This method is, however, efficient for the objects that are active and being processed within the scope of the CPU and the memory. Addressing by the CPU for reference to information is the use of the capability. The concept of capability-based addressing [FAB1] clarifies this situation.

There would be many ways to constitute a capability list in a computer. The segment table used in segmentation can be regarded as a capability list from the standpoint of protection [FAB1].

Method 2 is useful in the environment of a computer utility where many objects exist, and the revocation of access privileges given to other subjects is easily executed. This method is, however, "indirect" one; if subject S wants to refer to object O, it is necessary to scan the access-control list for object O and to check the access privileges. Thus, it is not suitable to employ this method to validate the access privileges every time the object is referred to from the viewpoints of execution speed and memory capacity.

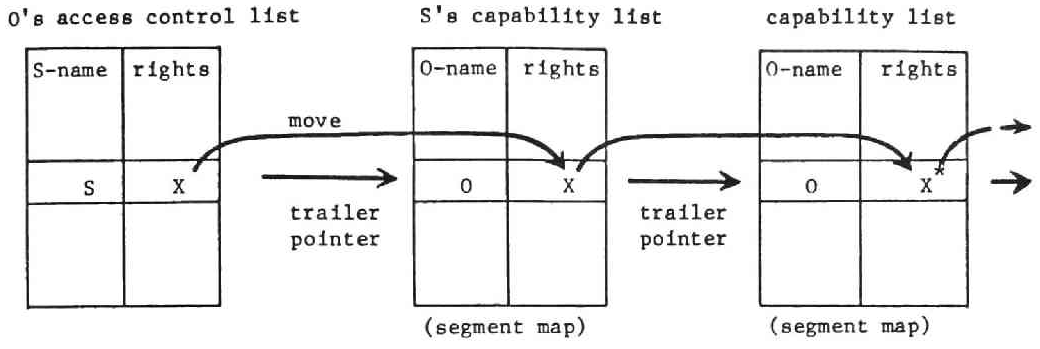


Figure 16.3 Transfer of access capability.

Method 4 is practical in this respect. When object O is referred to by subject S at first time, the access-control list of O is scanned, and then O is opened (activated). Once O is opened, the entry for S in the access-control list is moved to the capability list of S, and the entry in the capability list is chained and maintained the logical connection with the original access-control list of O so that revocation of access privileges is easily undertaken (see Figure 16.3). This method has been employed in Multics [ORG1], [IKE4], that is, an access-control list is used to validate the access privileges at the first time when some object is referred to by a process, and then, the access privileges are moved to the segment table of this process.

1.2 The Ring Protection Mechanism

The capabilities of the domains in the ring protection mechanism [SCH3] have the inclusion relation (see Chapter 7). The capability lists of the ring protection mechanism in the environment of segmentation

can be regarded as an array of segment maps, and in fact, a scheme which switches the segment maps is employed in the early version of Multics [ORG1]. (see Figure 16.4).

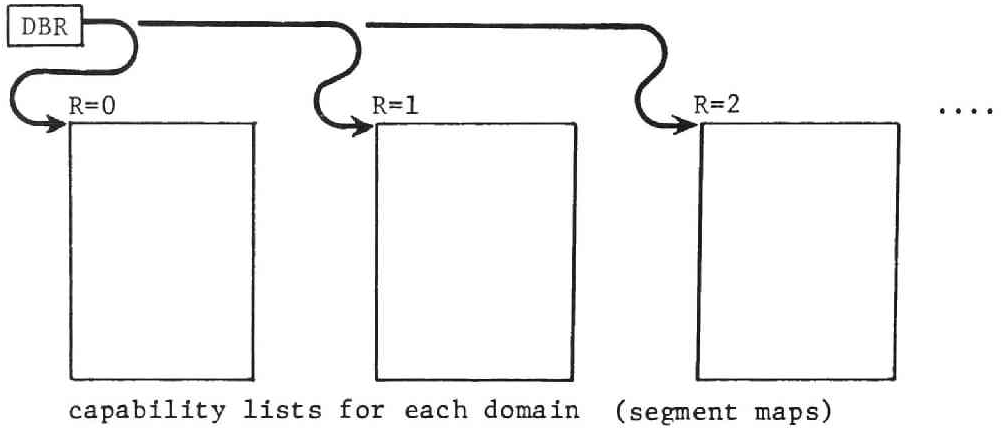


Figure 16.4 Realization of domains.

As the relation among the capabilities of the domains is rather simple, the array of segment maps can be reduced to the simplified mechanism as shown in Figure 16.5 by adding ring brackets in the segment map and a mechanism for limiting the capability. This limiting mechanism acts as a high-pass filter, and passes the access privileges whose ring bracket includes the current ring number.

1.3 Constitution of Independent Domains

Independent domains can be realized by preparing separate capability lists, and this means that it is necessary to prepare separate segment maps in the environment of segmentation. In a computer where multi-programming operation is employed, separate domains are realized by associating one segment map for each process. The relation between two processes is, however, entirely independent of each other, except the limited

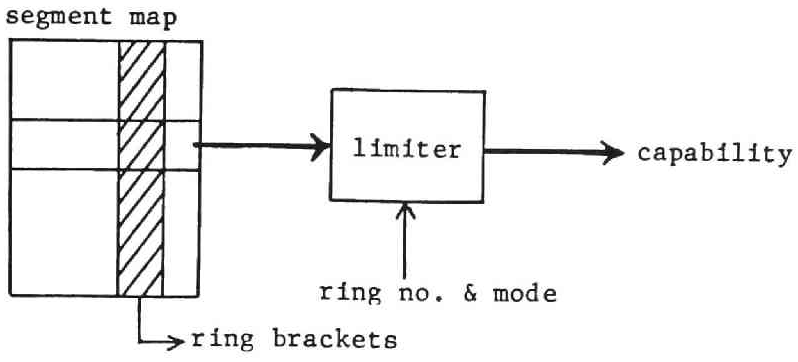


Figure 16.5 Ring protection mechanism.

communications by the inter-process communication.

In order to make direct communication and transfer among independent domains possible, it is necessary to alter the capabilities by switching the domains. Schroeder proposed a scheme to employ an array of capability lists [SCH2].

Fabry showed several classes of schemes to implement capability lists [FAB1]; however, there exists another class which will be shown below. Usually, segmentation mechanism uses one segment map for a process, and defines both the address space and the protection domain. By employing more than one segment map for a process, it is able to constitute more flexible domains, which satisfy more complicated requirements.

1.4 Owner's Capability

The usual segment map shows the capability held by one process in this domain. On the other hand, there is such a kind of applications which permit the reference to the special database D only through the specified program P which is composed by the owner of D himself. This

feature is called the set-user-ID feature in UNIX [RIT1], which gives the specified programs the privileges to use files inaccessible to other users. For example, a program may keep an accounting file which should neither be read nor changed except by this program itself. If the set-user-ID bit is on for this program, it may get access to the file although this access might be forbidden to other programs invoked by the given program's user.

We are going to employ this feature in the capability-based addressing system, which directly refers to objects as the operands of its processing, instead of files. This feature is called "owner's capability" below because the term "user" confuses the subject of capability.

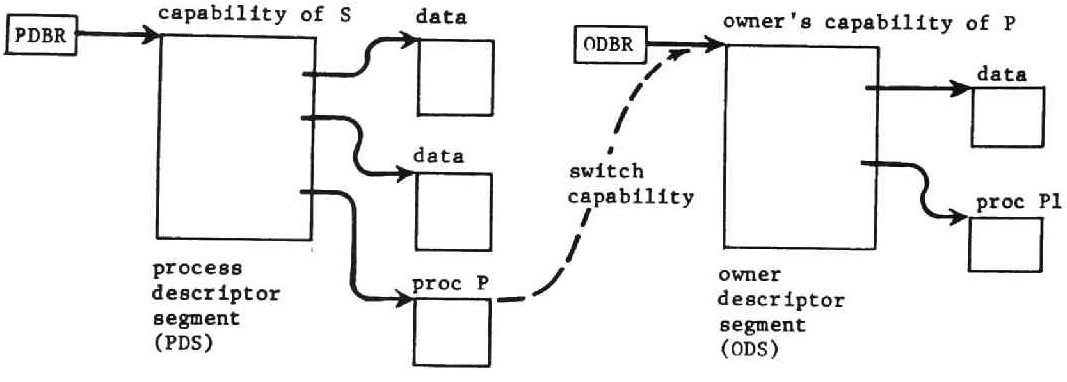


Figure 16.6 Owner's capability list.

It is clear that access privileges X for object D should not be placed in the segment map of subject S because this segment map shows the capability of S, and if privileges X for D is placed there, S can refer to D directly. What is permitted for S is only to execute P.

Access privileges X should be placed in a separate and private capability list for P. That is, process S can use D through the specified program P if the private capability list is used when P is invoked. This situation is shown in Figure 16.6, and hereafter, the capability list for a process is called a process descriptor segment (PDS), and the owner's capability list a owner descriptor segment (ODS), and these tables are pointed at by the process descriptor base register (PDBR) and the owner descriptor base register (ODBR) respectively.

Fabry also showed the use of multiple segment tables, but his scheme is the extension of the main table. The scheme shown in this chapter is not the mere extension of the main table, but describes the necessity for separate capability lists. A scheme which employs more than one descriptor base register is found in the design of ACOS [ACOS], but its distinction between the subjects of the capability is not clear, and the usage of them is different.

1.5 Capability for the Reference to the Arguments

The remaining capability is the one for the reference to the arguments accompanied by procedure invocation. The domain of execution is generally switched upon call to and return from a procedure. Thus, the capability for the reference to the arguments accompanied by procedure invocation is not generally guaranteed, and it is also necessary to pass the capability for the arguments to the called procedure. This capability list should be pushed down into a stack segment lest the generality of procedure invocation should be impaired (see Figure 16.7).

Hereafter, this stack segment is called an argument

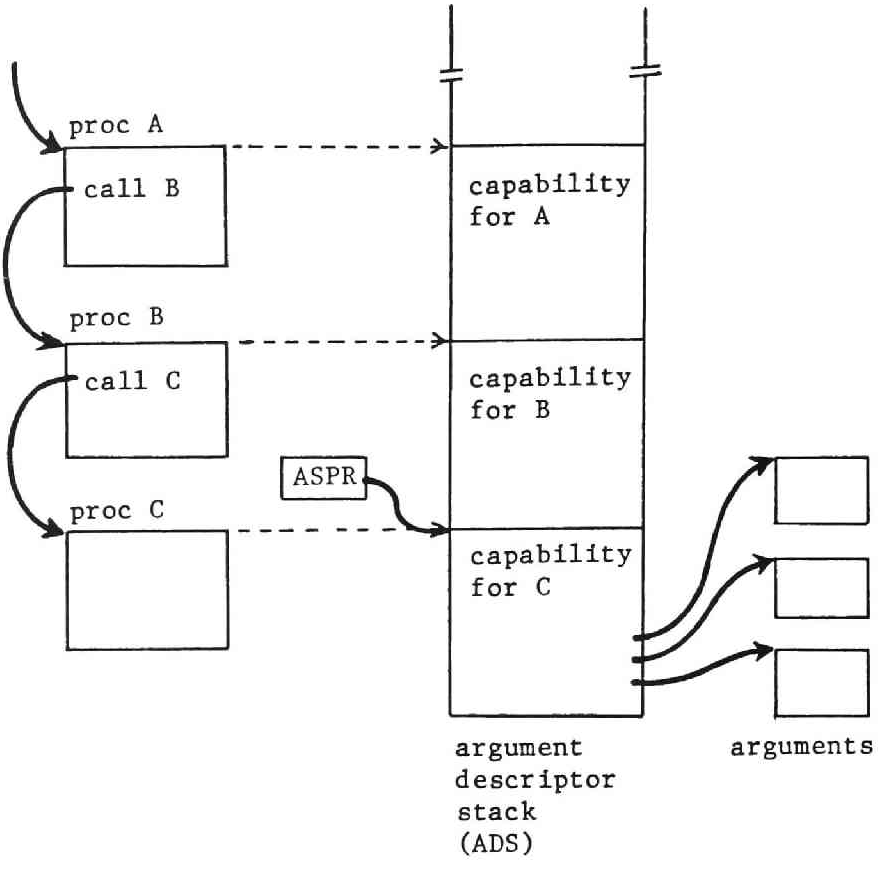


Figure 16.7 Access capability for arguments.

descriptor stack (ADS), and it is assumed that the argument stack pointer register (ASPR) points at the current stack frame. The same contrivance seems to be employed in ACOS; however, it has not been released and its details have not been opened yet.

It might happen that the same segment is entered in different capability lists, and this is often the case for the argument capability list. This makes the management of the trailer pointers, which are used to maintain the logical connection with the original access-control list, complex. Alternative method is to

use the indirection scheme which reduces all the dynamic capability to the static one, but there is no change in the fact that more than one capability list is needed.

1.6 Combination with the Ring Protection Mechanism

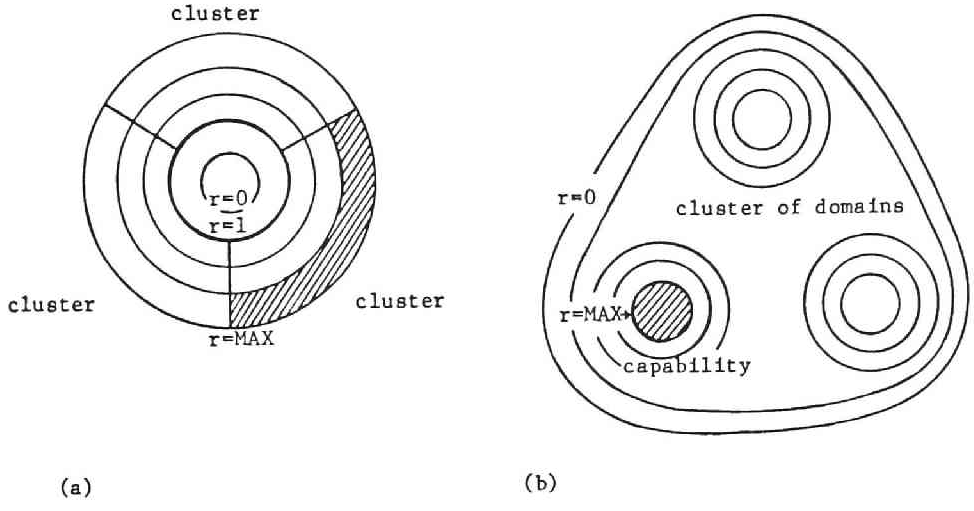


Figure 16.8 Cluster of domains.

The ring protection mechanism is useful though simple to be combined with the multi-segment map mechanism described above. The resulting domains consist of clusters of ring domains as shown in Figure 16.8. The security kernel is placed in the innermost domain, the managers of the system resources are placed in the next domain, and these domains are commonly used by the processes in the system. Figure 16.8a is drawn from the standpoint of protection, that is, inner domains are protected from the outer one. Figure 16.8b shows the same domains from the viewpoint of capability, and these two pictures are equivalent.

2. Use of Capability

2.1 Designation of Capability Lists

This section discusses how to use properly the capability lists.

Usual segmentation mechanism has only one capability list, and there is no need to discriminate two or more capability lists. Only the target segment number is needed to validate the capability.

If there are more than one capability list, it is necessary to differentiate them upon the addressing of an instruction. Two bits information is sufficient in the case of this paper as there are three kinds of lists for the process, the owner and the arguments. The number of bits would be increased if there are more kinds of lists which are required to show the capability in more complicated environment (see Figure 16.9).

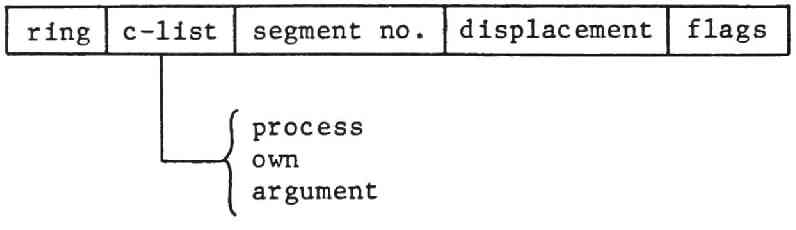


Figure 16.9 Address pointer.

The problem is where and how to include such information. The requirements for this are as follows:

A. A procedure segment should be pure in order to make it sharable.

B. The program logics should be independent of the protection issue.

Thus, it is not generally suitable to include the

indication of the capability list explicitly in the program body. Address pointers and address pointer registers can be used for this purpose. A mechanism is needed which transfers the same indication as the one that is used in the formation of effective address when the pointer is formed. One example of the usage of this function is shown in the section of argument list below. Of course, this indication should be included in some instruction; however, data segments are pointed at by pointer registers, which have the indication of the capability list, thus, it is not needed to include such an indication in usual instructions, and the program logics can entirely be made independent of the protection issue. And this is one of the excellent features of this scheme.

When an external reference is linked to the segment which is placed in the capability list of the process, PDS, the indication in the address pointer is set to denote PDS. If the segment is placed in ODS, the indication is set to denote ODS. The discussion on arguments will be delayed.

The pointers that are used to link to external segments are stored in the linkage segment which is the data segment proper to the process. Address pointers and the contents of pointer registers except those which point at the capability lists are non-privileged data. Static variables can also be allocated in this segment, and of course the linkage segment is a non-privileged data segment. On the contrary, the capability lists and the pointer registers which point at these lists can only be processed in the privileged mode of execution. Thus, there is no fear that the protection violation is caused (see Figure 16.10).

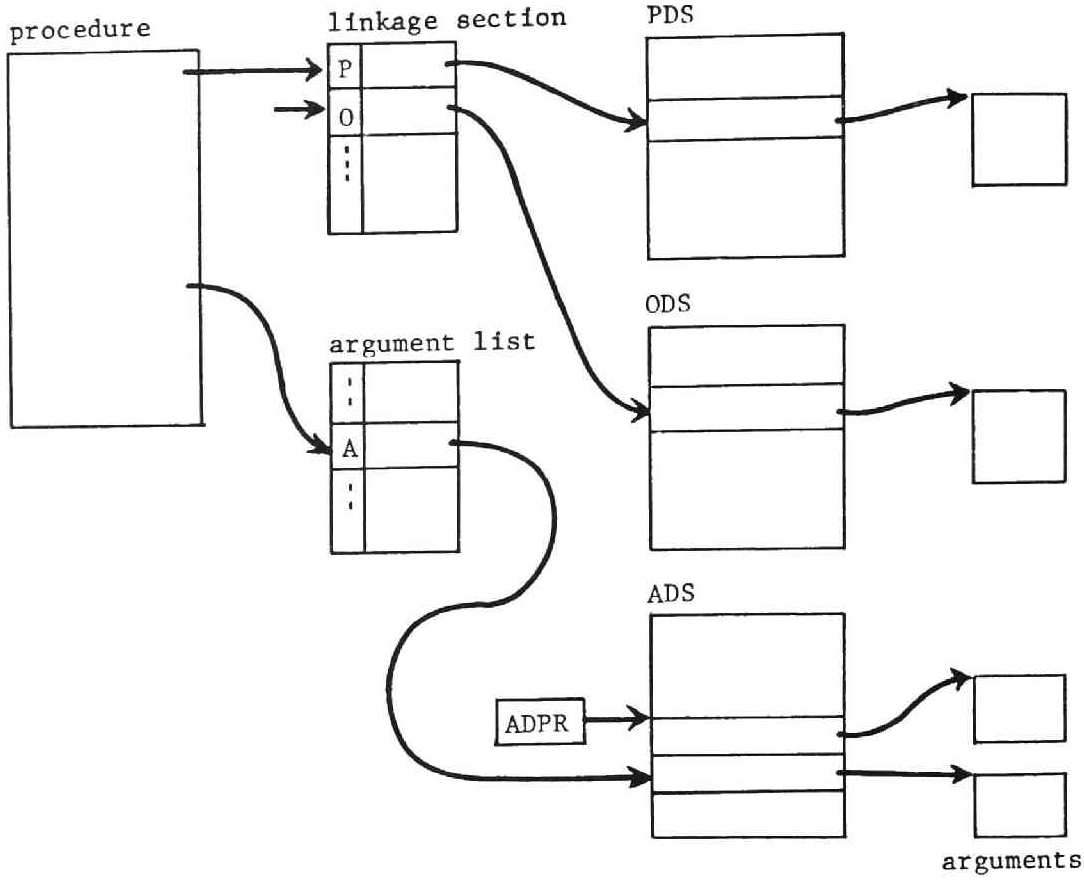


Figure 16.10 Protection with multiple capability lists.

2.2 Reference to the Arguments

Usually, arguments are accompanied with procedure invocation, but information which is placed in other domains cannot generally be referred to from the current domain. One way to solve the problem of arguments is to use common buffers and to copy and to copy back all of the arguments. However, the overhead of this method is big, and further, address pointers cannot be transferred because the references via these pointers are not always permitted.

The method which inhibits arguments is applicable only to limited cases, or the method which allows only those arguments that can be referred to in the called domain also affects the program logics unduly.

In the ring protection mechanism, an inner ring can refer to all the information placed in outer rings, thus, there is no problem with regard to the reference to the arguments if the cross-ring call is confined to an inward call [SCH3].

The method to solve generally the problem of arguments is to give "dynamically" [SCH2] the called domain the necessary capability for the reference to the arguments accompanied by procedure invocation. The following requirements arise here:

1. Not to change the usual sequence of procedure invocation.
2. To use a stack in order not to impose improper restrictions to the program logics.
3. Not to make the program logics complicated, nor to need tricky sequences.
4. Not to make sneak paths.

A procedure creates an argument list when it intends to invoke one procedure. This list generally consists of address pointers to the variables which are passed as arguments. Additional information as to the length and the kinds of access permitted are given besides the starting address. These can be created by the caller's procedure, and this processing has nothing to do with the domain of execution. Currently, it is assumed that the argument list is pointed at by the argument pointer register, and that the list is passed to the called procedure, which can refer to the arguments relative to this pointer register.

The next problem is how to give the necessary capability. The argument list itself is created by the caller procedure, and therefore, it cannot be a privileged database. Thus, it is necessary to create and to pass the capability list by validating the arguments against the capability lists given to the caller's domain if the domain switching is needed upon procedure invocation. This capability list should be created and pushed down into the argument stack automatically by the execution of a call instruction (see Figure 16.11). If the information of the return gate is included in this argument stack frame, all the necessary capability is passed to the called procedure. All the status of the caller is assumed to be saved in the stack frame of the caller, and the return gate implies the capability which is needed to restore the caller's domain and the status.

An alternative way to create the argument capability list is to generate capability descriptors directly instead of validating and converting the argument list into the capability list. For this purpose, an instruction which verifies the caller's capability, creates and pushes down a descriptor into the argument capability stack given the address, the length and the kinds of access of the argument which is intended to be passed to the called procedure is needed. Such an instruction might be executed in any state of execution and creates a privileged capability descriptor; however, the place where this descriptor is stored is pointed at by the privileged stack pointer which the process cannot alter in the usual state of execution, thus there would be no fear that any access violation or trick is caused. And the argument capability list is created safely.

The arguments that can be passed are those for which

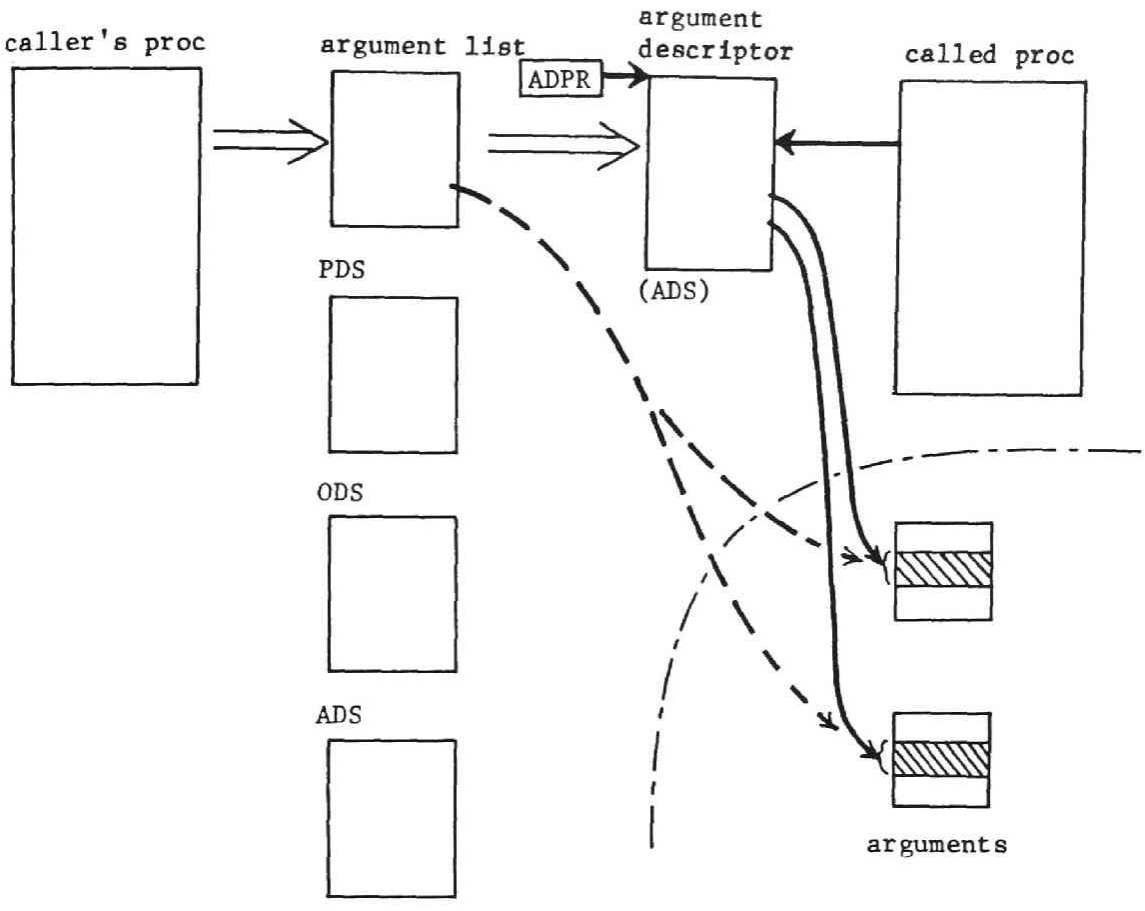


Figure 16.11 Reference to the arguments.

the caller has the necessary privileges to "transfer" as well as to refer. In addition to the usual privileges of read, write and execute, the privilege of transfer is needed. The concept of transfer is described by Graham [GRA2], but there are few implementations.

The validation of arguments can be undertaken by checking the proper capability list according to the pointers in the argument list created by the caller.

If an argument which is going to be passed has been passed as an argument by the previous caller, the pointer

in the argument list created by the current caller has the indication to ASPR as is described in Section 2.1. A part of a segment is often passed as an argument, and the "confinement of the area" can easily be realized by setting the start address, the length and the kinds of access permitted to the descriptor for the argument instead of using the original descriptor in the segment map. It seems that the shortening of a segment in ACOS [ACOS] corresponds to this processing though its algorithm is not opened to the public. Thus, the validation can be undertaken within the capability of the caller's domain and the called procedure can refer to the necessary arguments without any excess and deficiency.

The argument list passed to the called procedure is, in fact, the one created by this algorithm and is not the original one created by the caller itself.

The next problem is the revocation of privileges. It is necessary to chain the segment descriptor to its predecessor every time when it is passed as an argument capability so that the back-tracking procedure can crawl over the transferred capabilities to revoke them.

An alternative method is to pass the capability indirectly and to validate it in the originated domain. Schroeder proposed a scheme to employ a tag bit and a domain indicator which shows the domain that the validation should be executed [SCH2]. This method makes revocation process simple, but creates sneak paths because the validity check is done when the capability is really used not by checking the caller's privileges. If the caller sets the tag on for which the caller doesn't have the privileges in the caller's domain, but if the reference to this argument is permitted in the domain indicated as the originated one, the called procedure can

refer to this argument because the validity check is executed by checking the capability of the originated domain instead of the caller's privileges. This is, of course, improper thing. Then, he added the caller's indication instead of a single bit tag to prevent such a trick. However, the propagation of this indication is costly, and it is a little bit difficult to understand because of the indirection.

The switching of the capability upon call to or return from a procedure can easily be executed by pushing down or popping up the argument descriptor stack [BOB2].

3. Switching of Domains

The switching of domains can be executed by switching the capability lists. This switching is required when control is transferred from one procedure to another, including the case of an interrupt or a fault. Here, such a case that a new lexical level is created is excluded as it is an internal thing within a procedure, and it is assumed that the protection condition doesn't change within a segment (this is one of the definitions of a segment).

One of the requirements as to the domain switching is that the program logics should be independent of the domain of protection. Thus, the call and the return instruction are used to invoke a procedure without exception. Further, the program logics should not be changed even if the domain switching is required. Therefore, a mechanism which is combined the call/return function with the domain switching mechanism is required. SVC instruction is not adequate for advanced computer architecture as described in the earlier chapter.

A hardware mechanism which executes both the domain switching and the transfer of control is realized for the ring protection mechanism [SCH3]; however, there are few hardware mechanisms that can completely control more general protection schemes.

There are two kinds of domain switching in the mechanism of this thesis; one is the switching of the ring domains within a cluster of ring domains, and the other is the switching of domains crossing the wall of clusters. The switching of the ring domains has been described in Chapter 7.

The indication of domain switching can be placed in the flag of a segment descriptor which corresponds to the segment that needs the domain switching, and the domain switching may be executed by a hardware mechanism or by the intervention of software procedure invoked by a fault. The pointer to the new PDS may be placed economically in the address field of this descriptor.

Data segments are used to allocate the variables which are proper to the process, and these data segments should also be switched when the protection domains are switched. If the domain switching is executed within a cluster of the ring domains, and if the cross-ring transfer is restricted to an inward call and its consequent outward return [SCH3], [ORG1] no such switching of data segments is needed. Hereafter, it is assumed that stack segments are used for the data segments.

When a procedure is called, all the automatic variables of the caller procedure and the process status are saved in the stack frame [ORG1], the stack frame is pushed down, and a new stack frame is created and linked with the caller's frame. When a procedure invocation

which switches domains is executed, a new stack frame is created in a separate stack segment prepared in the newly switched domain. This push down sequence is executed with the domain switching operation.

When control is returned from the called procedure, the stack frame is popped up, and the process status is restored. If the domain switching was not caused upon call, this pop-up sequence is completed immediately. Otherwise, the domain is switched back by the domain switching mechanism, and the process status is restored.

3.1 Switching of Owner's Capability

An external reference is linked by a pointer in a linkage segment. Pointers relevant to a procedure constitute a linkage section, whose location within the linkage segment can be obtained by looking up in the offset table [IKE4]. The processing of a linkage section is rather simple in case of the static linking, hence dynamic linking [ORG1] is assumed here.

When a procedure is started being executed, this procedure first intends to set the linkage pointer from the offset table. If this procedure is executed at the first time, the linkage section for this procedure has not been created yet, hence a static storage fault is caused [JAN1], and the linkage section is created. If this procedure intends to refer to an external segment via an unsnapped link, a linkage fault is caused, and linking process is executed. If the target segment requires the owner's capability, its capability is placed in the owner's capability list, the domain switching condition is placed in the offset table, and then the owner's capability is switched (see Figure 16.12). It is enough to prepare an owner's capability list for each

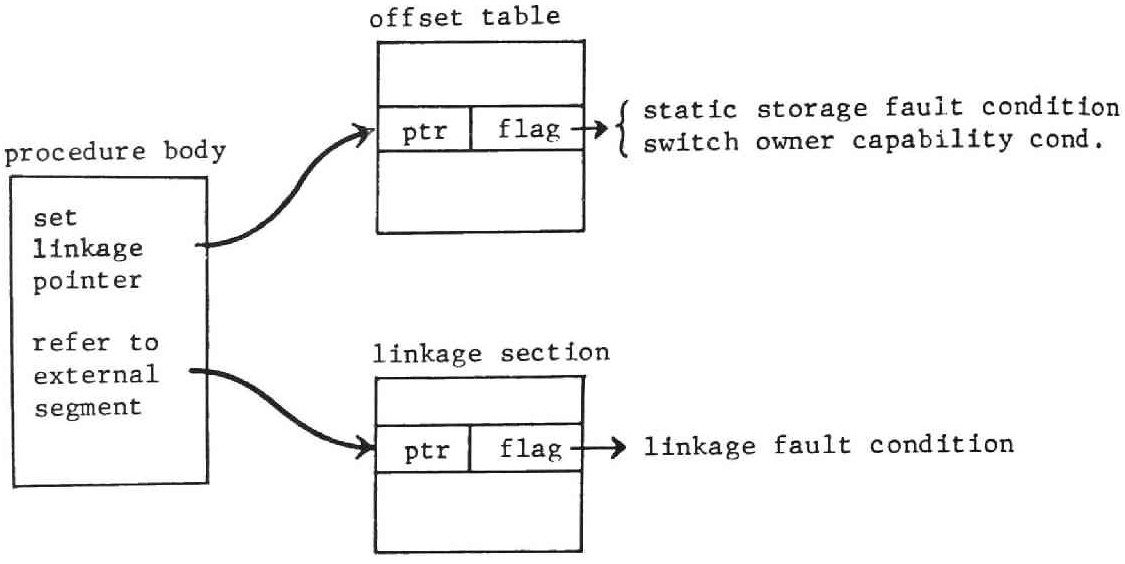


Figure 16.12 Switching of owner's capability lists.

owner. Some computations do not need such lists at all. It goes without saying that both the stack segment and the linkage segment should also be switched to the ones which are placed in the owner's capability list, too. Both the stack segment and the linkage segment are pointed at by the pointer registers in procedures, and therefore, it is not necessary to change the indications of the capability lists within the program, but to change the settings of these registers to point at the proper lists. Of course, it is necessary to copy the contents of the linkage section of the procedure, which has found that the target segment should be placed in the owner's capability list, in the old linkage segment to the new one and then to delete the old section.

It is able to switch back the owner's capability upon return from the called procedure if this condition is registered in the argument descriptor stack as the

return gate information.

CHAPTER 17
MEMORYLESS SYSTEM

Information systems which guarantee that no extra copies of procedures, data or results of processing are created against the intention of the user or the offerer are called memoryless systems [GRA2].

Hereafter, many cases would be caused that mutually suspicious subsystems of users share resources of procedures and databases each other in a computer utility. The purpose of a memoryless system is to protect such mutually suspicious subsystems.

In order to fulfill the complicated requirements of information protection, it has been shown in the previous chapter that it is useful to employ multiple capability lists, and this chapter discusses the conditions to construct the capability lists, especially write access under the owner's capability, for a memoryless system.

1. Memoryless System

There are various types of information sharing in a computer utility. Among them there are such mutually suspicious subsystems that user A permits user B to execute his program P with charge but A doesn't like that P is copied by B, and that B doesn't like that his data D which is going to be processed by P is copied by A without notice. In order to protect such mutually suspicious subsystems [LAM1], it is necessary to constitute a memoryless system which guarantees that no

copy of submitted data and results of processing is taken.

The requirements for a memoryless system are summarized as follows:

A. A user permits others to execute his procedures but doesn't like that they are copied by others.
B. A user permits others to refer to his databases indirectly via his procedures but doesn't like that they are directly copied by others.
C. A user who offers his procedures or databases wants to collect accounting data for charging or statistics.
D. A user wants to process his database utilizing other's procedures but doesn't like that his database and the results of the processing are copied by the borrowed procedures without notice.
E. A user doesn't want that the computer system makes any copy or leaves any traces of data or results of processing.

2. Gains and Losses in a Computer Utility

It would be useful for the management of access capabilities to discuss what are gains and what are losses with regards to the sharing of procedures and data in a computer utility. The followings would be gains:

1. To get useful information,
2. To execute operations which produce useful results.

And the followings would be losses:

3. To lose useful information,
4. To be disclosed secret information.

1. corresponds to the read capabilities which make possible to read directly the target or to obtain

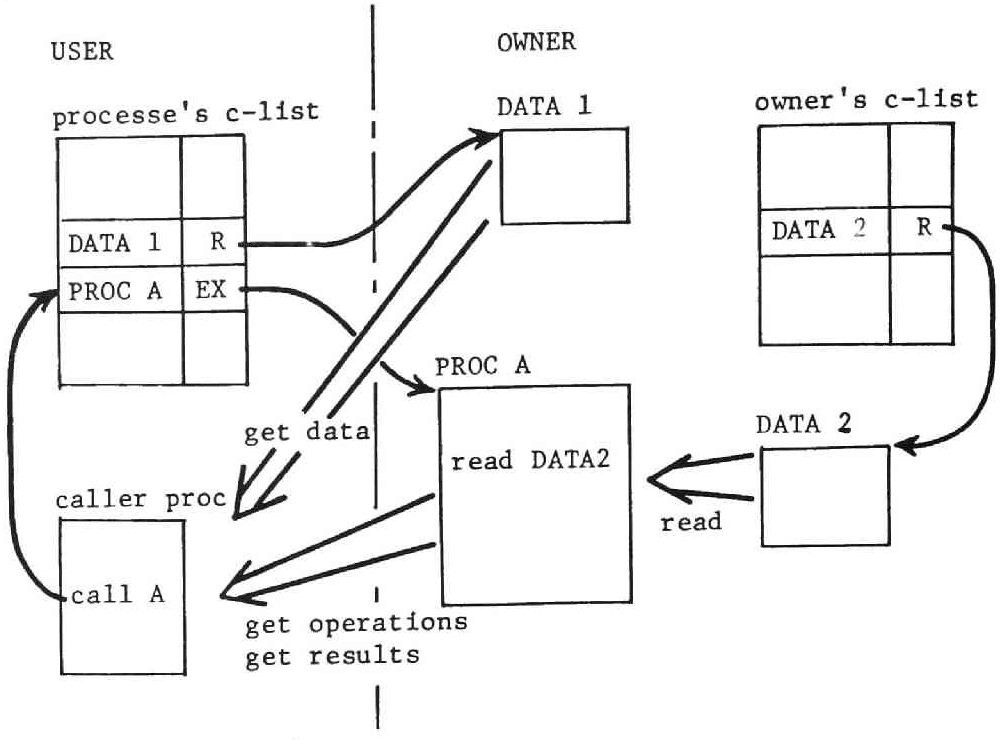


Figure 17.1 Reference to a data segment by owner's capability.

indirectly results of processing based on the target information (see Figure 17.1). The execute capabilities relate to 2. Problems of sneak paths arise when a procedure itself owns the write capabilities to some segments which are independent of the capabilities of the user who is currently borrowing and executing this procedure. When information which is passed to a procedure is written into such segments and the intended communication to the others is carried out correctly, it becomes gains to the user. On the contrary, when information is written into such segments without notice, it might become losses to him (see Figure 17.2).

Thus, it is recommended to execute a procedure with minimum capabilities, and if there is such a fear that secret copies would be made, it is better to abandon such write capabilities.

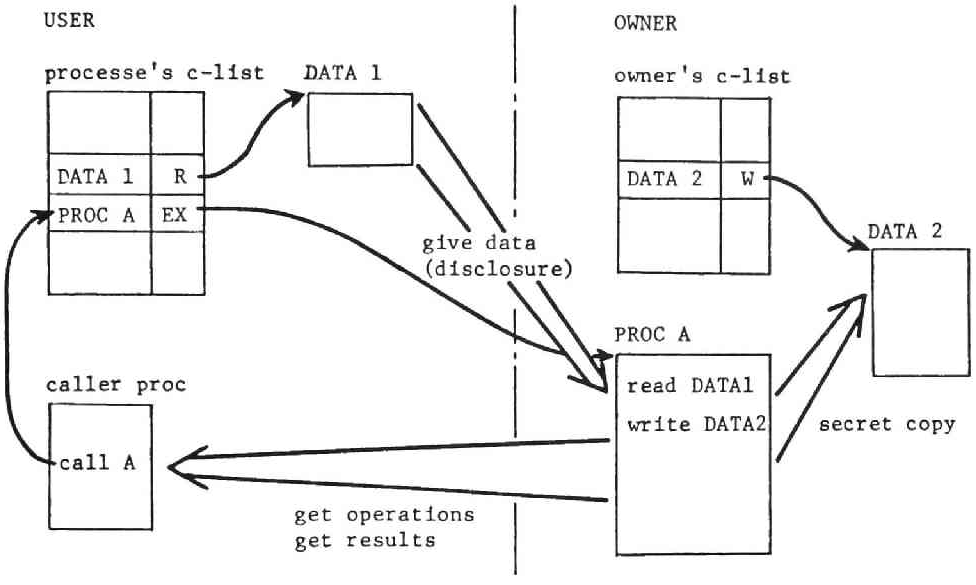


Figure 17.2 Secret copy by owner's capability.

3. Protection with Multiple Capability Lists

A protection mechanism which incorporates multiple capability lists has been discussed in the previous chapter. This mechanism uses three types of capability lists at least, they are the lists of the capability of the process, that is, the user himself, the capability with regard to the owner of the procedure that is running, and the capability that is used to refer to the arguments accompanied by the procedure invocation.

3.1 Processe's Capability

The processe's capability shows the capability of the user himself. This capability should be the least

according to the general strategy of protection - least privileges [SAL3] -. The less the capabilities the less the chances which misuse the capabilities that result in the disclosure of secret information.

3.2 Owner's Capability

The owner's capability is the double edged sword. Read or execute operations allowed by the owner's capability bring obvious gains to the user. And at the same time, the owner's capability guards the rights of the owner effectively. The owner's capability is, however, not always profitable to those who borrow procedures which are executed under the owner's capabilities. Sometimes, it happens that a user might result in to write data into some segments which are not readable by the user himself through the use of borrowed procedures, and this means that the user's information is given to the other. And to make matters worse, such write actions might be executed while the user is not notified where his control cannot reach.

Such sneak paths are of little problem in the case of calculations of functional values given fragments of data as arguments. Sometimes, such paths turn to be useful in the case of interprocess communications which pass data by the write operations under the owner's capability or in the case of malice, though it is anti-social, which intends to interfere the activity of others.

3.3 Capability for the Reference to the Arguments

By abandoning the write capability in the owner's capability, it is possible to eliminate the sneak paths mentioned above. Thus, the processe's capability and the

argument capability may be executed as the usual calculations.

4. Towards Memoryless System

Here, we trust the system itself, and assume that the memory area where the process uses is cleared and that the temporary databases are deleted completely after the calculation of the process.

It is rather easy to satisfy requirement A. Procedures are made execute-only and are executed in separate domains from the caller's domain lest copies of data segments such as stack and linkage segments should be made.

Requirement B is satisfied by referring to the required segments through some procedures of the owner of the segments under the owner's capability. The degree of release of the database is entirely controlled by the owner's will.

Requirement C seems to be satisfied by the use of the owner's capability that permits the write operations needed to log the accounting data; however, this method creates sneak paths, which are not suitable for requirement D. That is, borrowed procedures might copy the databases of the user that are to be processed into the ones of the owner. Thus, it is sometimes better to abandon the write rights of the owner's capability.

Stack and linkage segments are managed under the owner's capability in order to protect the owner's interest, but these segments cannot be reserved permanently by the will of the owner, thus copies of databases into these segments are of no use for later use.

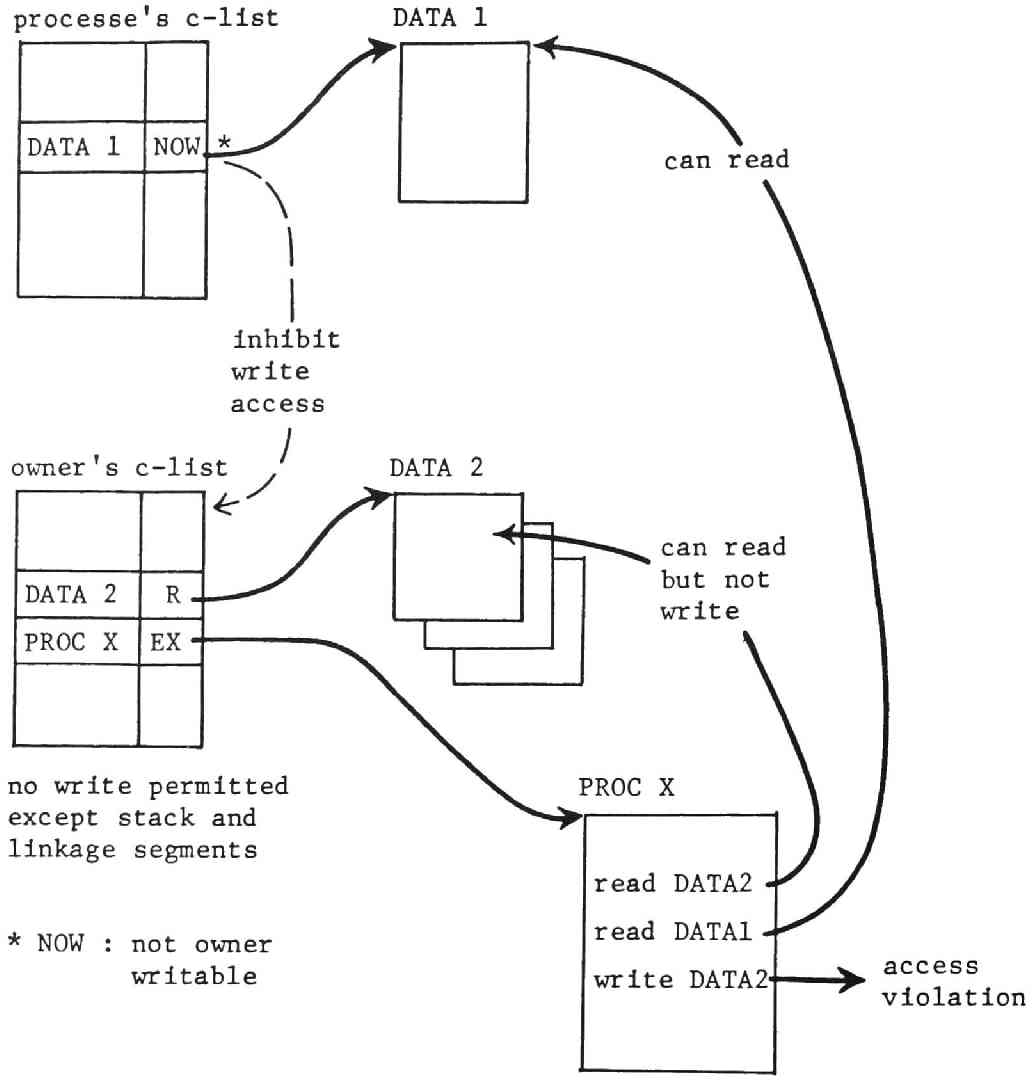


Figure 17.3 Not owner writable access right.

The decision whether or not the write operations of the owner's capability should be abandoned is left to the user. This condition might be notified to the security kernel prior to the call. This, however, violates the policy of protection [SAL3] - program generality -. An alternative way is to incorporate a "not owner writable"

flag in a segment descriptor. If there is a segment descriptor whose "not owner writable" flag is on in the processe's capability list, the write actions in the owner's capability list are forbidden except to stack and linkage segments. This "not owner writable" right is a new type of access rights, and is registered in the access control list of the directory entry of the segment that requires such rights. When such a segment is activated the access rights are copied into the segment descriptor in the processe's capability list (see Figure 17.3).

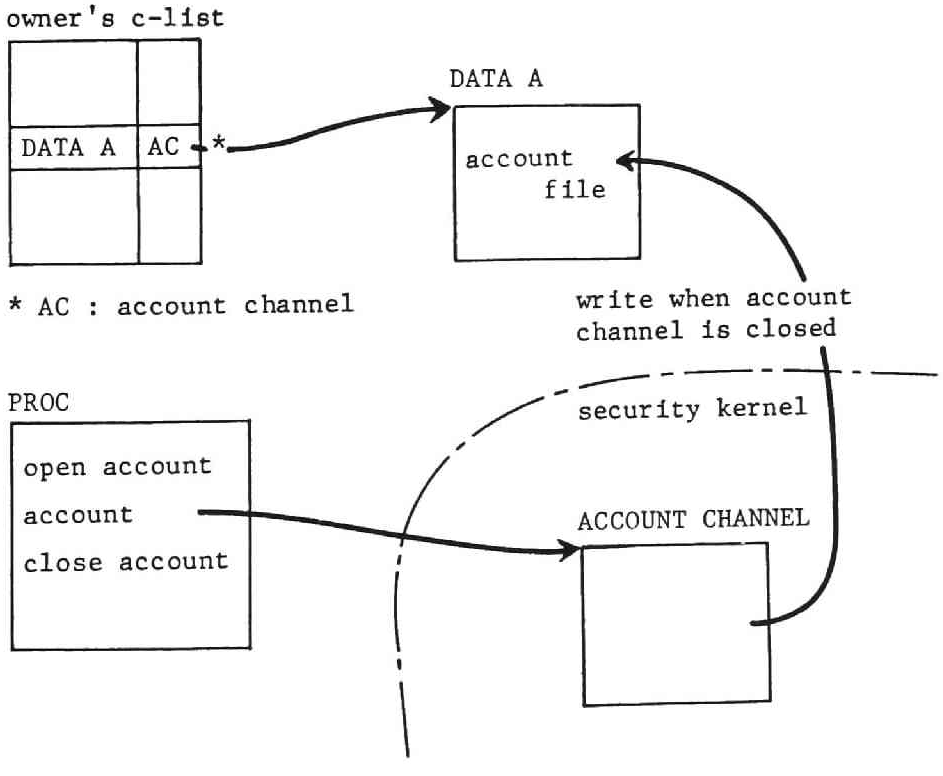


Figure 17.4 Account channel.

Thus, it is not adequate to write accounting information directly to some segments by the owner's

capability as is shown above. This might be resolved by negotiations of the user and the owner. Generally, accounting facility, say an account channel, supported by the system is needed for this purpose. A procedure asks the system to log accounting information through the account channel into some pre-assigned databases of the owner (see Figure 17.4). Care must be taken not to convey the raw parameters but to pass only the results of processing by the account channel lest the direct copy or the "coding" of user's information should be done by the frequency of calls or by the time interval of calls of the account channel. The account channel should produce the degenerated information such as the total quantity or the number of operations, etc. from the raw parameters given by the procedure. It would be impossible to develop the original information from the degenerated one. Here, a requirement of a new type of access rights, "append through the account channel", arises which permits the appending of accounting information to a segment.

CHAPTER 18
CONCLUSION

This thesis has discussed the significance, the points of issue, the constitution, and the related problems of an address space of a computer utility.

An address space of a computer utility is the space where a programmer expresses his algorithm and runs the calculation when he wants to solve a problem, utilizing a computer system. Thus, its constitution affects greatly on the method of solving a problem. So far, this problem is only partially solved within the scope of programming languages and their processors, but it would not be satisfactory unless this problem is resolved as the basic framework of a computer system. This thesis has discussed the problem of constituting an address space from the viewpoint of expressing and executing the algorithm to solve a problem putting relevant things together as a total system, and has shown the framework for the constitution of a desirable address space. This thesis has clarified and developed the mechanism of dynamic linking, clarified and extended the mechanism of information protection, clarified the mechanism of constitution and connection of information space and an address space of a process, clarified the mechanism of intra-procedural communication of a process, developed a unified mechanism of inter-procedural communication of a process, shown the structure of supervisor, related world to the address space, and the effective applications, developed a protection mechanism incorporating multiple

capability lists, and discussed the constitution of a memoryless system.

The productivity and the reliability of software have been very poor, though great efforts have been devoted to improve them. Not only users but also system programmers themselves have had much trouble in this fact. One of the reasons why the productivity and the reliability are poor is that algorithms expressed in software are based on "super"-concepts and their concepts and logical structures are sometimes left unclarified, or conveyed erroneously. In addition to this, algorithms are often affected by the configuration of a computer system, which has nothing to do with the algorithms of programs.

A programmer should devote all his energy to the development of an algorithm to solve his problem, without being puzzled by the system configuration, and algorithms should not be influenced by the configuration of a computer system, by the storage location of information, or by the mechanism of information protection.

This thesis has discussed the constitution of an address space based on segmentation and dynamic linking where direct addressing independent of the location of the target information is realized.

An address space is constituted in the information space of a computer system, including all the on-line files.

When we are developing an algorithm to solve a problem, the space for consideration might not be so simple as the one-dimensional memory space of a computer. Thus, it is desired that the space of the "surface" world should be mapped in a natural manner into the address space in a computer system.

Pieces of information, which include procedures and

data referred to in a computation, are separately managed according to their characters called attributes. Such a piece is called a segment, whose capacity or attributes are not fixed but happen to vary even in the course of a computation. It is also required that the number of segments which are referred to in a computation should not be confined to a small number.

Reference to the required segments should be carried out in a well formulated and easily understood manner lest it should make an algorithm needlessly complicated. And recursion should be supported by the basic framework of the system lest it should impose improper restrictions to the expression of an algorithm. In addition to above problems, supports of structures such as block structure, stack or queue are also needed. As a conclusion, segmentation supported by the two-dimensional address formation mechanism is suitable to form an address space in a computer utility, while a general three-dimensional addressing is not necessary for the moment.

Reference to information is an essential function of an information processing system. Reference to information is made effective by "linking" it to the target. The characters of an address space, and, as a result of it, the characters of a computer system, are basically changed by the method of linking. Linking of references in a computer utility should be delayed as late as possible. The method of linking when a reference is actually required - this is the last time of all - is called dynamic linking. Dynamic linking makes direct addressing of information independent of its storage location possible. And this implies that no input or output is required at all to refer to the information which is contained in on-line files, and a programmer can

refer to all the pieces of information he wants to use as if all were placed and referred to in the main memory. This will make program logics very clear and easy to understand and develop, and improve the productivity and the reliability of software much. The mechanism of segmentation is also suitable to support dynamic linking. This thesis has developed a method to remove the linker from the security kernel, whose method would be considerably useful to improve the integrity of the system.

In order to be able to refer to information in a computer utility, sharing of information is indispensable, and as the direct consequence of information sharing, the demand for protecting of information arises. Effective sharing of information results in the ensuring of the consistency of information and the saving of physical space to hold information. These problems are resolved by using the "original" information itself wherever it is placed. Dynamic linking is also suitable for this purpose. In order to share the "original" information, procedures must be pure, and processes use their own data segments to execute such pure procedures. Thus, the problem of recursion is resolved as well and a procedure can call itself so long as its static and automatic variables are definitely separated and properly allocated.

Protection of information is very important problem of today's computer system. Information protection in a computer utility demands that different access control be enforced in a natural way according to a process and the state of execution of this process even the same target is referred to, and such a control should be undertaken each and every time the reference is made. In addition,

information protection should not affect the logics of a program, and the mechanism of information protection should be simple enough to understand and to use; otherwise, erroneous usages would be caused and they would impair the security of the system instead. Further, the cost of utilizing such a protection mechanism should be proportional to the functional capability actually used.

The ring protection mechanism has solved many of these problems and realized useful protection domains. The structure of the ring protection mechanism is clarified and extended by constituting the clusters of ring domains to augment its applicability. And the structure of CPU is analyzed from the viewpoint of the ring protection. The mechanism of segmentation is also suitable to support the mechanism of sharing and protection of information.

Next problem is to realize an address space to run a process. For this purpose, the mechanism to describe an address space and to enable the CPU to form address under segmentation is needed as well as the mechanism to switch address spaces.

A process executes a procedure, referring to the associated data area and link area. When a procedure refers to another procedure, it executes such an instruction as a call, a return, or a non-local go to. In addition to these instructions, an interrupt, a fault, and a SVC instruction are often used. Their primary function is to invoke a handling procedure, and sometimes protection issue is raised. So far, these two points have hardly been separated, and each of them has been processed in an odd manner, which is entirely different from the usual sequences of procedure invocation executed

by a call and a return instruction. As a result of such processing, the structure of a program becomes needlessly complicated, which is difficult to understand and to develop its algorithm.

The unified method of procedure invocation is shown in this thesis, which will be useful to simplify the structure of programs.

The problem of constituting an address space is also summarized from the standpoint of constitution of supervisor. A computer system may be considered as a layered computer. Each layer is analyzed one by one in a top-down way.

There exist one main process and related i/o sub-processes for a calculation. The world outside an address space is disclosed, including the stream i/o and databases in a computer network.

A strategy toward a programming system which is independent of the configuration of a computer system is discussed, and it is shown that dynamic linking method is useful for this purpose. And constitution of a command system is shown as one useful application of dynamic linking.

Finally, constitution of protection mechanisms which incorporate multiple capability lists is discussed and developed. In the complicated situation of a computer utility where there is a variety of competing users, more powerful protection mechanisms are required, and the necessity of the process capability list, the owner capability list, and the argument capability list is shown. In addition to this, constitution of a memoryless system is discussed. It is shown that a certain kind of access capability sometimes produces harmful reactions on the information protection. This problem is resolved by

additional access rights.

ACKNOWLEDGEMENT

The author would like to express his sincere gratitude to Professor Takeshi Kiyono for supervising this thesis with constant encouragement. Professor Kiyono was thoughtful enough to provide the author with the valuable chances for the study of the living large computer systems.

This research was first stimulated by the constitution work of Information Processing Center of Kyoto University, and advanced and developed by studying the structure of Multics at Project MAC in Massachusetts Institute of Technology. The mechanism of dynamic linking and the ring protection, which gave the basis of this research work, were first contrived and implemented in Multics.

Acknowledgement and special thanks go to Professor Fernando J. Corbato and Professor Jerome H. Saltzer who gave the author the chance and enthusiastic encouragement for the study of Multics.

The author wishes to express his thanks to Professor Shuzo Yajima for his encouragement and valuable discussions.

The author also wishes to express his thanks to staffs of Professor Kiyono's research group for helpful discussions and various conveniences for the research. This thesis is edited and ROFFed on the small scale computer utility in that group. Thanks go to the people who contributed to build the utility.

Professor Tsuneko Ikemiya of Tezukayama College read through the early version of the draft of this thesis and pointed out errors in English sentences. The author thanks for the errors in English that are not here.

REFERENCES

[ACOS] "ACOS-77 Systems Manual," NEC/TOSHIBA.

[AND1] Andrew, J., Bash, J.L., Gondy, M.L., Hart, J.E., "GE-645 Processor Reference Manual," GE, 1970.

[ARD1] Arden, B.W., Galler, B.A., et al., "Program and Addressing Structure in a Time-Sharing Environment," JACM Vol. 13, No. 1, pp. 1-66, 1966.

[BAK1] Baker, H., Scheffer, L., Spencer, D., "Initialization and Shutdown on the Rise and Fall of Multics," Private communication with Project MAC, 1971.

[BEN1] Bensoussan, A., Clingen, C.T., Daley, R.C., "The Multics Virtual Memory: Concepts and Design," CACM Vol. 15, No. 5, pp. 308-318, 1972.

[BOB1] Bobrow, D.G., Burchfiel, J.D., et al., "TENEX, A Paged Time Sharing System for the PDP-10," Proc. Third Symp. on Operating System Principles, pp. 1-10, 1971.

[BOB2] Bobrow, D.G., Wegbreit, B., "A Model and Stack Implementation of Multiple Environments," CACM, Vol. 16, No. 10, pp. 591-602, 1973.

[BOC1] Bock, R.V., "An Interrupt Control for the B5500 Data Processor System," AFIPS Conf. Proc. Vol. 23, pp. 229-241, 1963 FJCC.

[BON1] Bonneau, R., Haas, R., Konig, D., "User Control: Command System: Error Handling," Private communication with Project MAC, 1971.

[BUR1] "Burroughs B6700 Information Processing System: Reference manual," Burroughs, 1972.

[COD1] Codd, E.F., "Multiprogramming Stretch," IFIP Proc. pp. 574- , 1962.

[CON1] Confort, W.T., "A Computing System Design for User Service," AFIPS Conf. Proc. Vol. 27, pp. 619-626, 1965 FJCC.

[COR1] Corbato, F.J., Vyssotsky, V.A., "Introduction and Overview of the Multics System," AFIPS Conf. Proc. Vol. 27, pp. 185-195, 1965 FJCC.

[COR2] Corbato, F.J., "A Paging Experiment with the Multics System," In Honor of P.M. Morse, pp. 217-228, MIT Press, Cambridge, 1969.

[COR3] Corbato, F.J., "PL/I as a Tool for System Programming," Datamation Vol. 15, pp. 68-76, May 1969.

[CRS1] Crisman, P.A., "The Compatible Time-Sharing System," MIT Press, Cambridge, 1965.

[CRT1] Critchlow, "Generalized Multiprocessing and Multiprogramming Systems," AFIPS Conf. Proc. Vol. pp. 107-126, 1963 FJCC.

[DAL1] Daley, R.C., Neuman, P.G., "A General-Purpose File System for Secondary Storage," AFIPS Conf. Proc. Vol. 27, pp. 213-229, 1965 FJCC.

[DAL2] Daley, R.C., Dennis, J.B., "Virtual Memory, Process, and Sharing in Multics," CACM Vol. 11, No. 5, pp. 306-312, 1968.

[DAT1] Date, C.J., "An Introduction to Database System," Addison-Wesley, New York, 1976.

[DAV1] David, E.E., Jr., Fano, R.M., "Some Thoughts About the Social Implications of Accessible Computing," AFIPS Conf. Proc. Vol. 27, pp. 243-247, 1965 FJCC.

[DET1] De Treville, J., Flower, R., Baron, R., "The Multics Dynamic Linking and Related Topics," Private communication with Project MAC, 1971.

[DON1] Donovan, J.J., "System Programming," McGraw-Hill, New York, 1972.

[DNG1] Denning, P.J., "Resource Allocation in Multiprocess Computer Systems," MAC-TR-50, MIT, 1968.

[DNG2] Denning, P.J., "Virtual Memory," Computing Surveys, Vol. 2, pp. 153-189, 1970.

[DNG3] Denning, P.J., "The Working Set Model for Program Behavior," CACM, Vol. 11, No. 5, pp. 323-333, May 1968.

[DNS1] Dennis, J.B., "Segmentation and the Design of Multiprogrammed Computer System," JACM Vol. 12, No. 4, pp. 589-602, 1965.

[EVA1] Evans, D.C., LeClerc, J.Y., "Address Mapping and the Control of Access in an Interactive Computer," AFIPS Conf. Proc. Vol. 30, pp. 23-30, 1967 SJCC.

[FAB1] Fabry, R.S., "Capability-Based Addressing," CACM, Vol. 17, No. 7, pp. 403-412, 1974.

[FEI1] Feiertag, R.J., Organick, E.I., "The Multics Input/Output System," Proc. Third Symp. on Operating System Principles, pp. 35-41, 1971.

[FOR1] Forgie, J.W., "A Time and Memory-Sharing Executive Program for Quick-Response On-line Applications," AFIPS Conf. Proc. Vol. 27, pp. 599-609, 1965 FJCC.

[GIB1] Gibson, C.T., "Time-Sharing with IBM System 360: Model 67," AFIPS Conf. Proc. Vol. 28, pp. 61-78, 1966 SJCC.

[GLA1] Glaser, E.L., Couleur, J.F., Oliver, G.A., "System Design of a Computer for Time Sharing Applications," AFIPS Conf. Proc. Vol. 27, pp. 197-202, 1965 FJCC.

[GRA1] Graham, R.M., "Protection in an Information Processing Utility," CACM Vol. 11, No. 5, pp. 365-369, 1968.

[GRA2] Graham, G.S., Denning, P.J., "Protection - Principle and Practice," Proc. AFIPS, Vol. 40, pp. 417-424, 1972 SJCC.

[HON1] "The Multics Virtual Memory," Honeywell Manual, AG 95, 1972.

[IBM1] "IBM Operating System 360 : Concepts and Facilities," IBM Form C28-6535, 1965.

[IBM2] "IBM System 360 Operating System: PL/I Reference Manual," IBM Form C28-8201, 1969.

[IBM3] "IBM System 370 Principles of Operation Manual," IBM Form GA 22-7000, 1970.

[JAN1] Janson, P.A., "Removing the Dynamic Linker from the Security Kernel of a Computer Utility," MAC-TR-132, MIT, 1974.

[KAR1] Karger, P.A., Longtin, B.G., "Multilevel Memory Management Extended to Dismountable Disk Packs," Private communication with Project MAC, MIT, 1971.

[KUR1] Kurtz, T.E., Lochner, K.M. Jr., "Supervisory System for the Dartmouth Time-Sharing System," COMPUTER and AUTOMATION, pp. 25-27, Oct. 1965.

[LAM1] Lampson, B.W., "Dynamic Protection Structure," Proc. AFIPS, Vol. 35, pp. 27-38, 1969 FJCC.

[LIN1] Linden, T.A., "Operating System Structure to Support Security and Reliable Software," Computing Surveys, Vol. 8, No. 4, pp. 409-445, 1976.

[MAD1] Madnick, S.E., Donovan, J.J., "Operating System," McGraw-Hill, New York, 1974.

[MCC1] McCullough, J.D., Speierman, K.H., "A Design for a Multiple user Multiprocessing System," AFIPS Conf. Proc. Vol. 27, pp. 611-617, 1965 FJCC.

[MOT1] Motobayashi, S., Masuda, T., Takahashi, N., "The HITAC 5020 Time Sharing System," Proc. ACM 24th Nat. Conf., pp. 419-429, 1969.

[MPM1] "Multics Programmer's Manual," MIT Information Processing Center, 1972.

[MSPM] "Multics System Programmer's Manual," Project MAC, MIT, 1969.

[NAK1] Nakazawa, K., et al., "Memory Control of HITAC 8700/8800," JOHOSHORI, Vol. 16, No. 4, pp. 325-330, 1975.

[NEE1] Needham, R.M., "Protection Systems and Protection Implementation," AFIPS Conf. Proc. Vol. 41, pp. 572-578, 1972 FJCC.

[ONI1] Onishi, I., Noguchi, K., "Constitution of Virtual Space of Super High-Performance Computer System," Conf. Proc. of Information Proc. Society of Japan, No. 18, 1972.

[ORG1] Organick, E.I., "The Multics System: An Examination of Its Structure," MIT Press, Cambridge, 1972.

[ORG2] Organick, E.I., "Computer System Organization," Academic Press, New York, 1973.

[OSS1] Ossanna, J.F., "Communications and Input/Output Switching in a Multiplex Computing System," AFIPS Conf. Proc. Vol. 27, pp. 231-241, 1965 FJCC.

[OSS2] Ossanna, J.F., Saltzer, J.H., "Technical and Human Engineering Problems in Connecting Terminals to a Time-Sharing System," AFIPS Conf. Proc. Vol. 37, pp. 355-362, 1970 FJCC.

[RAN1] Randell, B., Russell, L.J., "ALGOL 60 Implementation," Academic Press, New York, 1964.

[RAP1] Rappaport, R.L., "Implementing Multi-Process Primitives in a Multiplexed Computer System," MAC-TR-55, MIT, 1968.

[RIT1] Ritchie, D.M., Thompson, K., "The UNIX Time-Sharing System," CACM, Vol. 17, No. 7, PP.365-375,1974.

[SAL1] Saltzer, J.H., "Traffic Control in a Multiplexed Computer System," MAC-TR-30, MIT, 1966.

[SAL2] Saltzer, J.H., Ossanna, J.F., "Remote Terminal Character Stream Processing in Multics," AFIPS Conf. Proc. Vol. 36, pp. 621-627, 1970 SJCC.

[SAL3] Saltzer, J.H., Schroeder, M.D., "The Protection of Information in Computer System," Proc. IEEE, Vol. 63, No. 9, pp. 1278-1308, 1975.

[SCH1] Schroeder, M.D., "Performance of the GE-645 Associative Memory While Multics is in Operation," ACM Workshop on System Performance Evaluation, pp. 227-245, 1971.

[SCH2] Schroeder, M.D., "Cooperation of Mutually Suspicious Subsystems in a Computing Utility," MAC-TR-104, MIT, 1972.

[SCH3] Schroeder, M.D., Saltzer, J.H., "A Hardware Architecture for Implementing Protection Rings," CACM, Vol. 15, No. 3, pp. 157-170, 1972.

[SIM1] Simizu, H., Takeyama, H., Aoshima, K., "Dynamic Linking of OS7," Conf. Proc. of Information Proc. Society of Japan, No. 184, 1973.

[SPI1] Spier, M., "Multics Standard Object Segment," Multics Staff Bulletin, No. 27, MIT, 1972.

[STE1] Stern, J., "Traffic Control Primitives," Private communication with Project MAC, MIT, 1971.

[VYS1] Vyssotsky, V.A., Corbato, F.J., Graham, R.M., "Structure of the Multics Supervisor," AFIPS Conf. Proc., Vol. 27, pp. 197-202, 1965 FJCC.

[WAT1] Watson, R.W., "Timesharing System Design Concept," McGraw-Hill, New York, 1970.

[WEB1] Webber, S.H., "Stack Header and Stack Frame Format," Multics Internal Document, 1972.

[WIL1] Wilkes, M.V., "Time-Sharing Computer System," American Elsvier, 1968.

LIST OF PUBLICATIONS

[IKE1] Ikeda, K., "Management of Address Space in Multics," Meeting Memo. No. 3, Dept. of Information Sci., Kyoto Univ., 1972.

[IKE2] Ikeda, K., "Control of Interrupts and Faults in Multics," Seminar Report No. 4, Information Processing Center, Kyoto Univ., 1973.

[IKE3] Ikeda, K., "Virtual Memory of Multics," Information Proc. Society of Japan, Kansai chapter Seminar Report, 1973.

[IKE4] Ikeda, K., "Structure of a Computer Utility," Shokodo, Tokyo, 1974.

[IKE5] Ikeda, K., "Structure and Constitution of an Address Space in Multics," JOHOSHORI, Vol. 16, No. 4, pp. 369-372, 1975.

[IKE6] Ikeda, K., "A Few Problems on the Reference and the Protection of Information in a Computer Utility," Report No. 27, SIG Computer Architecture of IPSJ, 1977.

[IKE7] Ikeda, K., "A Scheme to Execute the Dynamic Linker as a Non-Privileged Procedure," * Information Processing, Vol. 1, No. 1, 1978 (to apperar).

[IKE8] Ikeda, K., "Unification of Sequences of Procedure Invocation," in subscription to * Information Processing.

[IKE9] Ikeda, K., "Protection with Multiple Capability Lists," in subscription to * Information Processing.

[IKEA] Ikeda, K., "Memoryless Protection Mechanism," in subscription to * Information Processing.

[HOR1] Hori, Y., Ikeda, K., Kiyono, T., "High Level Command System of Interactive Processing," 17 th Conf. Proc. of Information Proc. Society of Japan, No. 162, pp. 317-318, 1976.

[NIS1] Nishikado, I., Ikeda, K., Kiyono, T., Hoshino, S., "A Study on a Computer Complex," 12 th Conf. Proc. of Information Proc. Society of Japan, No. 3, pp. 5-6, 1971.

*(insert) Journal of

[OKA1] Okajima, I., Ikeda, K., Kiyono, T., "Construction of Virtual Memory in a Mini-Computer," 14 th Conf. Proc. of Information Proc. Society of Japan, No. 178, pp. 355-356, 1973.

[OMU1] Omura, S., Ikeda, K., Kiyono, T., "Construction of Computer Complex for Input/Output Processing," 16 th Conf. Proc. of Information Proc. Society of Japan, No. 277, pp. 553-554, 1975.

[SHM1] Shimono, M., Ikeda, K., Kiyono, T., "Implementation of Segmentation in a Mini-Computer," 18 th Conf. Proc. of Information Proc. Society of Japan, No. 56, pp. 111-112, 1977.

[YAM1] Yamashita, K., Ikeda, K., Kiyono, T., "Constitution of Hierarchical File System," 18 th Conf. Proc. of Information Proc. Society of Japan, No. 166, pp. 325-326, 1977.