RICE UNIVERSITY

An implementation model for a multiprocessor Operating System on a Descriptor Oriented Architecture

by

Anand Jagannathan

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

THESIS DIRECTOR'S SIGNATURE:

Edward Alvin Feustel

HOUSTON, TEXAS

(JULY 1976)
ABSTRACT

An Implementation model for a Multiprocessor Operating System on a Descriptor Oriented Architecture

by

Anand Jagannathan

This thesis evolves a computer system which serves as the basis for operating system design in a multiprocessor environment. The evolution presupposes the distinction between the 'mechanisms' and 'policies' which comprise an operating system and suggests an implementation for the set of mechanisms, which satisfy the criteria of flexibility and extensibility. The assumption is that the policy defining part of the operating system can be considered as a set of cooperating concurrent processes.

The set of mechanisms, available to the system designer implementing the policy defining operating system, is termed as the 'nucleus' and is treated as a set of protected entities available to all processes in the system. The control of access is effected through a protection structure, which provides for the flexibility, extensibility and protected access. The protection structure uses a 'capability' model for protection and is supported by a descriptor oriented architecture. It is shown that the limited protection features of the supporting architecture can be generalized to construct the protection structure, with respect to which, the nucleus can be appropriately specified. The nucleus is directed towards the delineation of as many features as possible, from a 'monitor' which is constrained to be used exclusively and which provides system wide functions.
Acknowledgements

I wish to express my sincere appreciation and gratitude to Dr. E.A. Feustel, my thesis advisor, for his guidance. I wish to thank Dr. G.S. Rao and Dr. S.K. Rusk for serving on my thesis committee and for their helpful suggestions. I would also like to extend my thanks to Mr. J.K. Iliffe, who provided some valuable hints during the initial phase of this research. Finally, I am grateful to all friends and well wishers for their friendship and good wishes.

This research was supported by the National Science Foundation under grants NSF-MCS-72-03751 A01 and NSF-MCS-72-03609 A01.
# Table of Contents

## Chapter I Introduction

1.1 The Objective .................................................. I-1
1.2 The Need .......................................................... I-2
1.3 Protection ......................................................... I-4
1.4 Operating Systems .............................................. I-9
1.5 The Base Architecture ........................................... I-12
1.6 Outline of the thesis ........................................... I-13

## Chapter II The conceptual basis for the Protection Structure

II.1 Introduction .................................................... II-1
II.2 The theory of Capability Structures ......................... II-1
II.3 Analysis of Capability Structures ............................ II-7
II.4 A generalized Pointer Number Machine ..................... II-12
II.5 Applicability of the GPN model to the Capability Structure ........................................ II-17
II.6 Conclusion ...................................................... II-18

## Chapter III The Protection Structure

III.1 Overview ....................................................... III-1
III.2 Elementary Resource Management Functions ................ III-2
III.3 The Process State Vector ...................................... III-6
III.4 The Protection Structure ...................................... III-10
III.5 Domain Interaction ............................................ III-31
III.6 Multiprocessing ................................................ III-36
III.7 Implementation of the Protection Structure ............... III-37

## Chapter IV The nucleus of the Operating System

IV.1 Introduction .................................................... IV-1
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV.2</td>
<td>The design philosophy of the Operating System</td>
<td>IV-1</td>
</tr>
<tr>
<td>IV.3</td>
<td>Process Cooperation</td>
<td>IV-6</td>
</tr>
<tr>
<td>IV.4</td>
<td>Input/Output handling features</td>
<td>IV-18</td>
</tr>
<tr>
<td>IV.5</td>
<td>Process creation and control</td>
<td>IV-22</td>
</tr>
<tr>
<td>IV.6</td>
<td>Short term Resource Management</td>
<td>IV-25</td>
</tr>
<tr>
<td>IV.7</td>
<td>Conclusion</td>
<td>IV-26</td>
</tr>
<tr>
<td>Chapter V</td>
<td>Analysis of the System</td>
<td>V-1</td>
</tr>
<tr>
<td>Chapter VI</td>
<td>Conclusion</td>
<td>VI-1</td>
</tr>
<tr>
<td>VI.1</td>
<td>Summary</td>
<td>VI-1</td>
</tr>
<tr>
<td>VI.2</td>
<td>Areas for future development</td>
<td>VI-2</td>
</tr>
<tr>
<td>Appendix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.1</td>
<td>The Micro PN machine</td>
<td>A-1</td>
</tr>
<tr>
<td>A.2</td>
<td>The modified Micro PN machine (MPN)</td>
<td>A-8</td>
</tr>
<tr>
<td>A.3</td>
<td>Analysis of the MPN machine</td>
<td>A-12</td>
</tr>
<tr>
<td>Abbreviations</td>
<td></td>
<td>Ab-1</td>
</tr>
<tr>
<td>Glossary</td>
<td></td>
<td>G-1</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>R-1</td>
</tr>
<tr>
<td>List of Tables</td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td></td>
<td>vi</td>
</tr>
<tr>
<td>Table Title</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>2.1 GPN tagged element formats</td>
<td>II-13a</td>
<td></td>
</tr>
<tr>
<td>3.1 MPN Instructions</td>
<td>III-2a</td>
<td></td>
</tr>
<tr>
<td>3.2 EPN tagged element format</td>
<td>III-10a</td>
<td></td>
</tr>
<tr>
<td>3.3 Interpretation of the fields in the CWD,RCOD formats</td>
<td>III-12a</td>
<td></td>
</tr>
<tr>
<td>3.4 Base level capabilities</td>
<td>III-20a</td>
<td></td>
</tr>
<tr>
<td>4.1a The Process description</td>
<td>IV-22a</td>
<td></td>
</tr>
<tr>
<td>4.1b The monitor primitives</td>
<td>IV-22a</td>
<td></td>
</tr>
<tr>
<td>4.2 Table of operations and Nucleus' features</td>
<td>IV-27a</td>
<td></td>
</tr>
<tr>
<td>A.1 The micro PN tagged elements</td>
<td>A-1a</td>
<td></td>
</tr>
<tr>
<td>A.2 Micro PN instructions</td>
<td>A-4a</td>
<td></td>
</tr>
<tr>
<td>A.3 MPN Instructions</td>
<td>A-8a</td>
<td></td>
</tr>
<tr>
<td>A.4 Classification of MPN instructions</td>
<td>A-12a</td>
<td></td>
</tr>
<tr>
<td>A.5 Allowed operations on elements</td>
<td>A-15a</td>
<td></td>
</tr>
</tbody>
</table>
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The fields within a capability</td>
<td>II-2a</td>
</tr>
<tr>
<td>2.2a</td>
<td>Capability formation</td>
<td>II-6a</td>
</tr>
<tr>
<td>2.2b</td>
<td>Alternate scheme for capability formation</td>
<td>II-6a</td>
</tr>
<tr>
<td>2.2c</td>
<td>CPC formation</td>
<td>II-6a</td>
</tr>
<tr>
<td>3.1</td>
<td>Elementary resource management functions</td>
<td>III-4a</td>
</tr>
<tr>
<td>3.2</td>
<td>The Process State Vector</td>
<td>III-6a</td>
</tr>
<tr>
<td>3.3</td>
<td>MARK and CALL operations on the PSV</td>
<td>III-8a</td>
</tr>
<tr>
<td>3.4</td>
<td>The RETURN operation</td>
<td>III-9a</td>
</tr>
<tr>
<td>3.5</td>
<td>The hardware registers</td>
<td>III-9b</td>
</tr>
<tr>
<td>3.6</td>
<td>Segment structure in storage</td>
<td>III-10b</td>
</tr>
<tr>
<td>3.7</td>
<td>Illustration of the SEAL, REVOKE and LOCK operations</td>
<td>III-14a</td>
</tr>
<tr>
<td>3.8</td>
<td>The SEAL operation</td>
<td>III-15a</td>
</tr>
<tr>
<td>3.9</td>
<td>The REVOKE operation</td>
<td>III-15b</td>
</tr>
<tr>
<td>3.10</td>
<td>Structure representation using normal codewords</td>
<td>III-16a</td>
</tr>
<tr>
<td>3.11</td>
<td>Structure representation with relative codewords</td>
<td>III-16b</td>
</tr>
<tr>
<td>3.12</td>
<td>The GROUP operation</td>
<td>III-17a</td>
</tr>
<tr>
<td>3.13</td>
<td>The EXPAN operation</td>
<td>III-18a</td>
</tr>
<tr>
<td>3.14</td>
<td>The Base description</td>
<td>III-22a</td>
</tr>
<tr>
<td>3.15</td>
<td>Definition segment format for extended capabilities</td>
<td>III-27a</td>
</tr>
<tr>
<td>3.16</td>
<td>Different cases of extended capability definition</td>
<td>III-28a</td>
</tr>
<tr>
<td>3.17</td>
<td>The process stack after a cross domain call</td>
<td>III-35a</td>
</tr>
<tr>
<td>3.18</td>
<td>The Master Segment Table entry</td>
<td>III-40a</td>
</tr>
<tr>
<td>3.19</td>
<td>The EVAL operation</td>
<td>III-41a</td>
</tr>
<tr>
<td>4.1</td>
<td>EJAP formation with a normal codeword</td>
<td>IV-10a</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.2</td>
<td>ICAP formation with a STRUCTURE codeword</td>
<td>IV-10b</td>
</tr>
<tr>
<td>4.3</td>
<td>Segment format for a Q structure ECAP</td>
<td>IV-12a</td>
</tr>
<tr>
<td>4.4</td>
<td>Definition of a Queue D structure</td>
<td>IV-13a</td>
</tr>
<tr>
<td>4.5</td>
<td>The Semaphore implementation</td>
<td>IV-13b</td>
</tr>
<tr>
<td>4.6</td>
<td>Message Buffer implementation</td>
<td>IV-14a</td>
</tr>
<tr>
<td>4.7</td>
<td>Event variable implementation</td>
<td>IV-15a</td>
</tr>
<tr>
<td>4.8</td>
<td>The Input/Output operations</td>
<td>IV-20a</td>
</tr>
<tr>
<td>A.1</td>
<td>The micro PN EVAL operations</td>
<td>A-7a</td>
</tr>
<tr>
<td>A.2</td>
<td>The MOVE, TAG and Interrogative operations</td>
<td>A-9a</td>
</tr>
<tr>
<td>A.3</td>
<td>Class D1 and D2 operations</td>
<td>A-14a</td>
</tr>
</tbody>
</table>
Chapter I

Introduction

I.1 The Objective

This thesis proposes a generalized methodology for the design of operating systems. The term 'implementation model' refers to the evolution of a framework for the design of any operating system. The framework assumes the context of a multiprocessing environment. The basis for the hardware support is a descriptor oriented architecture [Bu 61, Il 72, Fe 73], the associated protection mechanisms of which, form an integral part of the framework.

All along, the emphasis is on an organized design methodology. Therefore, to be consistent, the 'meta design' of this design methodology needs to be formulated in a systematic way. The meta design follows a top down scheme, although the resulting design methodology is based on a bottom up structuring technique. The meta design is based on the premise of, the need for a systematic process to design operating systems, which reflects the changing features of architectural environment and user requirements. Thus the motivation for this thesis is to suggest a basic tool for operating system design, which incorporates the features of multiprocessing, at the same time providing flexibility and ease of use to the system designer. The specification of this design methodology is organized on a scheme of protected mechanisms implemented on a descriptor oriented architecture.

The essential requirement for a generalized methodology in operating system design is the differentiation between the 'mechanisms' common to all operating systems and the 'policies' which by using
these mechanisms, define specific operating systems. A scheme of providing this identification of mechanisms and policies is by a hierarchical structuring of the operating system into various layers. This enables a grouping of the layers basic to all operating systems as the set of mechanisms and the identification of the layers overlaid on these as the set of policies. This approach assumes the notion of extendibility in the operating system.

The hierarchical structuring of an operating system suggests the treatment of the policy defining layers as the cooperation of concurrently executing processes [Di 68b]. This implies that the development of the set of mechanisms is appropriate in the context of a multiprocessing environment. The need for an orderly specification of this set of mechanisms is satisfied by enforcing them as protected entities available to the policy defining concurrent processes. The treatment of mechanisms as protected entities requires an architecture on which this protection environment can be structured. This is satisfied by the class of descriptor oriented architectures. This completes the outline of the top down meta design. As indicated earlier, the resulting design methodology will be shown as an evolution from the base architecture to the final specification of an environment which supports the design of specific operating systems, thus implying a bottom up structuring.

I.2 The Need

This thesis asserts that, rather than following the prevailing trend of tailoring existing operating systems to satisfy user requirements and architectural innovations, a more fundamental
modification of the design process itself is required; with the provision of an architectural support, which reflects the new design methodology. The need for this has been brought about because of the gradual change in the utility of computer systems.

Originally, a computer system was dedicated to one user at a time, the only concern being the provision of a computational facility which could complement the user's mental capability. The gross underutilization of resources with the resulting high cost of operation led to multiprogramming systems. Still, the basic concern remained the same, namely the emphasis on the power of computation rather than flexibility, resulting in the complete isolation of users. The sharing of some common system features prompted the sharing of user defined features, thus leading to the need for protection between users. Solutions to this problem of protection were provided as supplements to the operating system, but it was still not recognized that these solutions could be modified to form an integral part of the operating system. The essence of this thesis is a treatment of the components of an operating system as undifferentiated from users of the system and the adaptation of protection mechanisms to control the interaction of these components.

We now realize that the design of a flexible and dynamic framework for the development of operating systems requires further study in the areas of protection, multiprocessing operating systems and a suitable base architecture. The fusion of these apparently independent concepts forms the basis for our development.
I.3 Protection

Current protection systems are concerned with inter user interaction. In this context, protection is a general term for all the mechanisms which control the access of a user program to some entities or 'basic units' belonging to other users, or to the system. In such an environment, simultaneous utilization of the computer system is provided to a large community of users. The operating system for such systems, not only needs to ensure that users have the ability to maintain their privacy, but also that users are able to interact with each other without sacrificing their integrity. Moreover the operating system has to make sure that a user is not able to interfere with its own operation.

Violation of protection is then, securing access to a unit for which the user is not authorized. Lampson [La 74] classifies two scales of violation, namely those caused by accident and those caused by malice. On this basis, protection is defined as a defense against specific classes of dangers or threats and is subdivided into two categories- absolute and defensive protection. The former unconditionally guarantees the containment of a program to access only those entities, it is authorized to; while the latter stresses on the likelihood of a protection violation being low rather than impossible.

Protection mechanisms have been studied extensively in literature [Gr 68, La 69, La 71, La 74] and schemes have been proposed which utilize these mechanisms to form viable protection structures in a user environment. We identify the design parameters used to characterize typical protection structures. These parameters common
to all protection structures are:

i) The identification of a 'basic unit' which is the object of protection for the mechanisms.

ii) The specification of 'access' to this 'basic unit'.

iii) The identification of the 'entity' which is restricted in its 'access' to the set of 'basic units'.

iv) The specification of a 'representation' which indicates that an 'entity' is authorized to 'access' a 'basic unit'.

v) The formulation of a mechanism which can alter the representation which indicates the 'accessibility' of a 'basic unit' to an 'entity'.

In most practical situations, the basic unit of protection is a logical unit of information, more specifically a segment in storage.* This model is general enough to include all cases of protection mechanisms. For instance, the supervisor/user mode of operation in standard computers can be translated as the execution of instructions stored in restricted areas of memory, for supervisor mode and the execution of instructions in the rest of memory for user mode. More important however, the generalized notion of a resource in a system permits the interpretation of its protected access as the control of access to stored information, thus conforming to the above model. We will use this idea in our implementation scheme.

We had earlier identified protection with the control of access of a user program to some 'basic units'. This implies that permission to access a basic unit is to be given to a user program. However a user program is too static and inflexible for this purpose.

*Assuming segmentation in a virtual store.
So we identify the entity to which permission is given, as a process. A process represents the dynamic execution state of a program and can be termed as the logical processor for the execution of the program.

Since a process is restricted in its access to the set of basic units, we need a means of indicating this restriction or alternatively indicating the allowed accesses, a process is authorized to make. We therefore define the 'execution environment' of a process as the 'representation' which indicates the set of basic units accessible to the process. If the access is such that it is possible in varying degrees, then the maximum degree of access allowed is indicated as part of the execution environment. As an example, if access is defined as the ability to read/write with respect to a storage segment, then the execution environment needs to specify whether the degree of access is restricted to read only or write only or to allow both or to allow neither.

Finally, provision should be made to modify the execution environment with respect to which the process executes. This may be by the process itself, assuming that the execution environment contains the permission to allow the process to change it. Alternatively, the execution environment may be changed externally by some other process which considers the execution environment as a 'basic unit' and its own execution environment contains the permission for it to change the other execution environment.

The actual control of a process' access to the set of basic units is provided by protection mechanisms. These can be based on either of three models -- list oriented mechanisms, ticket oriented mechanisms and message transmission mechanisms. List oriented mechanisms
associate the basic unit of protection with an access control list which defines the allowed range of accesses to various processes in the system. It is inelegant for direct hardware support but compensates with a degree of completeness and flexibility with regards to alteration of access. Ticket oriented mechanisms, on the other hand, provide simple verification at the basic unit level. It facilitates hardware implementation but is inefficient with respect to the enumeration of a large number of access specifications. Transfer of access privileges is easily effected by copying the ticket but subsequent revocation is difficult to enforce. Message oriented mechanisms rely on the complete isolation of execution environments, the sole form of interaction being through the exchange of messages. The assumption is that reliable message communication and message authentication is guaranteed [La 74]. Rather than being a separate scheme, it serves to model the other two and can be treated as the conceptual basis for the other two schemes. In practice, list oriented schemes are applied to user level interfaces and ticket oriented schemes at a hardware level. Our primary concern is to apply ticket oriented schemes to form a protection structure on which the design methodology can be formulated.

We have indicated that although, the ticket oriented mechanism supports transfer of privileges, revocation of these transferred privileges is difficult. However solutions for revocation are possible and we will illustrate the idea of using the message transmission scheme to model a solution. The message mechanism models the passing of a ticket between two processes as the transmission of the ticket in a message between the two governing execution environments. Later revocation is modeled as another message to the borrower execution
environment asking it to destroy the ticket, but no controls exist to ensure that the borrowing environment obeys the lender's message. We can solve this problem of revocation by a scheme of indirection through another ticket of the lender. This is modeled as three execution environment: the borrower, the lender and a caretaker. Whenever access is necessary, the borrower transmits a message to the caretaker which then allows access. Revocation is implemented as a message from the lender to the caretaker environment (which is under the control of the lender's execution environment and consequently has to obey the message), asking it to ignore further requests of access from the borrower. Thus we have illustrated the adaptability of the message transmission scheme to model the ticket oriented scheme.

We will use the nomenclature 'domain' to denote the execution environment of a process. The domain process linkage can be described as a binary relation between the set of processes and the set of domains in the system. In practical implementations, this binary relation is often restricted to be a function, a reversible function, a one to one function and so on. The key idea in dynamic protection structures is that the domain with respect to which the process executes, changes dynamically depending on the current execution point of the process. From the initial model proposed by Graham [Gr 68], restrictions, by enforcing a linear ordering on the set of domains with respect to which a process can execute, have enabled the hardware implementation of such dynamic structures [SS 72]. Removal of the linear ordering restriction and providing for full generality in terms of 'mutually suspicious systems', have also been implemented as dynamic protection structures [Sc 72].
In a ticket oriented approach, the domain of a process can be described as a set of tickets indicating the 'basic units' accessible to the process. For a dynamic protection structure, this set of tickets needs to vary as the execution point passes through various procedures called by the process. Furthermore, this set may be changed by other processes as long as they possess the tickets to do so.

We have discussed protection as prevalent in existing user interactive systems. The essential idea was to recognize the fundamental concepts underlying protection structures. We now need to apply these concepts to our specific requirements, which will be done in the remaining chapters.

1.4 Operating Systems

There are two central ideas in layered and extendible operating systems [La 69b]. First, the layers are arranged so that each one creates a more convenient environment for the next layer. In lower layers, a bare hardware machine is converted into a large number of user machines which are given access to common resources in a controlled manner. Secondly, the design admits increasing complexity in higher layers, so that at the user interface, it is possible to use elaborate procedures, which treat lower level operations to be primitive, so as not to comprise system integrity.

Dijkstra [Di 68] first recognized the need for an organized and systematic design methodology. The motivation was to prove the correctness of an operating system by using a layered approach wherein each layer could assume the lower layer (the one closer to the hardware) to be functionally correct. The assumption was, that each layer in the
Hierarchical structure was described by making no assumptions about the higher layers. However, the term hierarchy is misleading for even a system with no hierarchical structure essentially exhibits a trivial one-level hierarchy [Pa 74]. Thus, it is more meaningful to identify the partial ordering relation, with respect to the components of the system, which is defined by the hierarchy. This relation can be represented by an acyclic downward directed graph. It is also possible, as in the case of the T.H.E. operating system [D1 68], for two hierarchical relations to exist simultaneously. In the T.H.E. system, the governing relation exhibited in the design is that of 'calling another procedure', while simultaneously the system is also viewed as a set of parallel sequential processes which exhibit the relation 'give work to' [Pa 74].

Recent developments in operating system design have called for a separation of the mechanisms and policies in an operating system [Ha 70, Ha 73, Wu 74]. This has been brought about because of the ever pressing need for a 'flexible mode of operation' in present day computer utilities. The underlying idea is that no longer should an operating system be defined by assuming a fixed mode of operation like batch processing, spooling, real time scheduling and so on, but should be flexible and dynamic to allow this to be changed even on a short term basis. Thus, it is more important to concentrate on the control of an environment where such changes in mode can be dynamically effected. This brings up the requirement for a basic set of mechanisms which control this environment. This basic set has been termed a 'kernel' [Wu 74] or a 'nucleus' [Ha 70] or a 'monitor' [Ha 73] and is essentially a collection of facilities of 'universal applicability'.
and 'absolute reliability'.

As stated earlier, we will specify the set of mechanisms as protected entities accessible in an environment which supports the cooperation of concurrent processes. The processes are hierarchically structured, the hierarchy defining the partial ordering relation of a parent process 'creating' a child process, the implicit assumption being that a parent process maintains full control of its child processes and is responsible for its medium term strategies of scheduling, resource management and so on. The hierarchical control maintained by a parent process enables the execution of different processes under essentially different operating systems.

The set of mechanisms is termed as the nucleus and is built around a protection structure. The protection structure is specified as a set of restricted operations and is itself hierarchically structured, the partial ordering relation being that of an operation being 'defined by a sequence of already existing operations'. The important feature in the protection structure is the support of a scheme of extendibility which allows the rest of the nucleus to also be structured on the basis of this same relationship. The nucleus, since it comprises of the set of mechanisms essential to all operating systems, is responsible for the short term resource handling, processor management and input/output required of any operating system. More important, it controls the environment in which the processes defining the 'policies' are able to cooperate with each other.

Since the variable operating system is now considered as a set of concurrent processes executing in an environment controlled by the nucleus, it is logical to design the nucleus assuming the context
of a multiprocessing environment. Multiprocessing has become of paramount importance, ever since the enhancement of system performance through hardware has been curtailed by limitations in processing speed due to the device technology. At this stage, a clarification with respect to the semantics of the nomenclature is useful in later discussions. Multiprogramming is the phenomenon of having more than one process active in time, in that it has been granted some resources [Lo 72]. The assumption is that physically only one processor is available, whereas multiprocessing is the case of multiprogramming on more than one processor. Thus, we have both multiple logical and physical processors. Parallel processing corresponds to the speed up of a single process by executing some operations on it concurrently or by a scheme of pipelining. The presence of a multiprocessing environment enforces some synchronization restraints on the design of the nucleus as will be shown later.

We have now identified the underlying philosophy for designing a basic tool for operating system design and shown the motivation for our approach. Using the rationale developed, we will study the specifics of designing and implementing a nucleus, in later chapters.

I.5 The Base Architecture

The requirements of a base architecture are dictated by the protection structure mentioned in the earlier sections. Since the protection structure is to be specified as a set of restricted operations, we need an architecture which assumes protection as an integral part of its construction and its specification. The protection structure defines the intervening layer between the hardware machine and the policy
defining operating system, with the nucleus being built around the
protection structure. One requirement of the protection structure is
therefore, a support for a natural scheme of extendibility. Understand¬
ably, the architecture is also to reflect this characteristic.
Fortunately, all these requirements are met by the class of descriptor
oriented architectures.

The basis of the protection structure is a treatment of the
system resources as 'objects' whose access is to be controlled. The
ticket oriented model for a protection mechanism introduces the notion
of a capability which identifies an object as its 'basic unit' and
controls the access allowed on the object. Descriptor oriented archi¬
tectures essentially support this notion of a capability by utilizing
a scheme of tags affixed to elements to identify them as capabilities
and by defining a restricted set of operations on capabilities.

The actual scheme used for the base architecture is a soph¬
isticated version of a descriptor oriented architecture and is a
refinement of the forerunner of this class of architecture, the B.L.M.
[Il 72]. This base architecture is called the 'Micro Pointer Number
machine' [ref. Appendix, I1 75] and provides a secure and efficient
code on which the protection structure can be reliably and systematic¬
ally constructed.

I.6 Outline of the thesis

In this chapter, we have stated the purpose and motivation
for the thesis and then studied the areas of protection, operating
systems and the requirements of the base architecture, in order to
provide a clear understanding of the concepts discussed in the rest
of the thesis.

Chapter II specifies the particular model which is used as the basis for the development of the protection structure. The essential features of the class of descriptor oriented architectures are also recognized and their applicability to the above model shown.

Chapter III contains a detailed description of the protection structure, overlaid on a base machine specified and analysed in the appendix. A strategy for the development of the nucleus is evolved on the basis of the features provided by the protection structure.

Chapter IV describes the requirements of the basic set of mechanisms for an operating system. A scheme, for the implementation of the nucleus as evolved from the protection structure, is proposed.

Chapter V conducts an analysis of the system and Chapter VI summarizes the research and also indicates new avenues for future development which have opened up as a result.
Chapter II

The conceptual basis for the Protection Structure

II.1 Introduction

The purpose of this chapter is to underline the theory and the concept behind the protection structure described in chapter III. The protection structure is interpreted as a flexible framework for resource management, on which any set of policies governing the resource allocation can be implemented. Given the above requirements, we need to develop a model which combines the ideas of a resource, protected access to this resource and a natural architectural support for this protected access. We will therefore introduce the concepts of a capability, an execution environment and a descriptor oriented architecture.

II.2 The theory of Capability Structures

a) The concept A capability is a ticket, the possession of which may be interpreted as prima facie evidence that the possessor is allowed to use the object defined by the capability. This idea was developed by Dennis and Van Horn [DV 66] as a generalization of protection and addressing schemes such as codewords* [Il 72], descriptors** [Bu 61] and segment tables***. The applicability of the above definition hinges on the interpretation of the key words 'use' and 'object'. Various capability architectures have provided different connotations and accordingly implemented different capability structures. For example, in the case of a capability serving as the replacement of an addressing scheme, an 'object' may well be a region of storage and 'use' may be-

* as in the Rice Computer
** as in the Burroughs machines
*** as in the IBM 360/67 models
in terms of gaining access to the storage in the form of read, write operations. Our interpretation relies on the notion of an object being an instance of a resource and the use of an object is termed as the set of operations defined on the resource and which are used to manipulate it. This generalized notion of a resource forms the basis for the implementation of a framework for resource management.

Since the capability defines an object, it has to provide the information to localize the object; which, in our case, means the identification of the resource associated with the capability. This can be done by a 'reference' field in the capability. Furthermore, the class, to which the resource belongs, also needs to be indicated. This calls for a 'type' field in the capability which identifies the set of objects in the same class. Finally, as we have interpreted the 'use' of an object as a set of operations on the resource, it is reasonable to assume that the capability also specifies the subset of valid operations which are allowed to be applied on the resource by the posessor of the capability. This specification may be provided for by an 'options' field. Thus a capability, as has been developed, consists of three information fields— the type, the reference and the options, as illustrated in Fig. 2.1. This development has essentially followed the scheme provided by Lindsay [Li 73]. Variations with respect to the information required in a capability have been suggested [Wu 74, Re 74, Fa 74], all these involving the requirement of an additional field* which specifies the unique identification code for the capability.

Although, the capability is a manipulable entity in the

* This is in addition to the 'reference' to the object.
Fig. 2.1 The fields within a capability
address space of a computation, it identifies an object in a much larger context. Thus the reference information is a system-wide entity. For purposes of implementation however, it is convenient to interpret the 'reference' as well as the 'options' field with respect to the 'type' information. This does not result in a loss of generality as capabilities of different types define disjoint classes of objects. The 'object' may, in turn, be defined in terms of capabilities although there is no requirement for this to be done. For instance, the HYDRA [Wu 74] view of an object is in terms of a unique name (which serves as the 'reference' in the capability), a type part and a representation part which consists of a capability part and a data part.

In a non-extendible system, it is possible, at the outset, to determine all the classes of objects possible and assign a unique 'type' code to each. On the other hand, in an extendible system, although a set of base level types can be identified, there needs to be a scheme for characterizing new classes of objects which may be formed as part of the extensions to the system. A commonly agreed upon scheme is to assign an object as the representative of the new class and use the 'reference' to the object as the type classification.

A similar problem arises in the interpretation of the 'options' field of a capability in an extended system. The main problem, both in the case of the 'type' interpretation and the 'options' interpretation, arises because of the fact that this interpretation is to be provided as a base level feature and is to be unaware of any extensions to the system. We can get around the problem again by specifying a convention which is to be followed by the extensions. Since the 'options' field specifies the subset of operations
allowed to the posessor of the capability, the extended object just needs to have its operations arranged serially and then follow the same ordering with respect to all capabilities defined on the object. The 'options' field is then simply interpreted as the specification of the indexes, in the serial ordering, of the allowed operations on the object. This is easily represented as a bit string where the operation is not allowed if the corresponding bit is not set.

b) The maintenance We had, in the previous sub-section, introduced the various fields which are required in a capability and the interpretation of these fields. Obviously, in a digital computer, these fields of a capability are going to be a string of bits. This brings about the imperative need to prevent any string of bits from being interpreted as the bits of a capability. More specifically we need to differentiate between ordinary data and capabilities, as also between the starting point of each of them. There are two basic methods of identifying capabilities. The first is by the partition scheme by which capabilities are restricted to exist only in certain regions (or partitions) of store. The second is the tagged storage scheme where tag bits are affixed to all elements and capabilities are flagged by a prefixed tag. The pros and cons of either scheme have been discussed extensively in literature [Li 73, Fa 74, Re 74, Wu 74], the arguments involving the fact that partition schemes result in forced structures with the associated loss of flexibility; while tagged storage schemes require specialized hardware support and the overhead of affixing tags to all elements in storage. We will use a tagged storage scheme which uses a specialized architecture which considerably minimizes this overhead.
Although the need for identification of capabilities in storage arises from the fact that data is not to be interpreted as a capability, a more important consideration arises in the context of protection namely the need to prevent unauthorized modification of capabilities. Since a capability defines the use of a system wide resource, it defeats the purpose if the capability can be modified so as to define another resource or so as to change the degree of use allowed on the same resource. Using the analogy of a ticket, we would not want the ticket to be altered to correspond to a different show altogether or even just to get a better seat. Thus we have to define a set of operations allowed on capabilities (as differentiated from the operations defined on resources referred to by the capabilities) and a set of operations allowed on data, realizing of course that these need not be disjoint. For instance, in the case of the HYDRA system [Wu 74], a computation may manipulate the data part of an object freely but can manipulate the capability part only by invoking special functions called 'kernel' functions. It naturally follows that since a capability is itself a resource with a set of operations defined on it, a computation may be restricted in the set of allowed operations on any capability. This suggests the idea of another capability which defines the set of operations allowed on capabilities.

Since, by modifying a capability, we get another capability, creation of a capability can then be provided for by modification. With a little generalization, modification of any element (data or capability) may be made so as to result in a capability, permission to make this modification being granted by a capability forming capability (CFC). Thus to construct a capability, a computation needs to
first form the fields of the capability as a data element and then by using the operation authorized by a CFC element, modify the data element into a new capability. This scheme is illustrated in Fig. 2.2a and incorporates the idea first suggested by Lindsay [Li 73]. His scheme was to control the type of the capability which can be formed, since interpretation of the two other fields is with respect to it. The new type information was part of the reference field in the CFC element and the new capability formed had all its options enabled (Fig. 2.2b). Conceptually, one could think of a CFC capability as being formed by the use of a CFC capability which allows the formation of another CFC capability (Fig. 2.2c).

We have been till now referring to the entity which uses the capability as a 'computation'. It is pertinent to clarify at this stage, that we are implicitly assuming a model where each user of the computer utility is represented by a process. A process is considered as the cumulative state of an independently schedulable task. It directs the computation of the user with respect to an environment defined for it. Thus it is the environment which possesses the capability and any process executing with respect to the environment may use the capability.

A point worth mentioning, regarding the implementation of capabilities, is that in case the interpretation of the capabilities is done by software, a 'display' operation is needed. The crux of the interpretation is in using the 'reference' field to localize the object defined. Commonly used schemes [Li 73, Fa 74, Re 74, Wu 74] rely on the association of a unique identification code (ID) with the object defined or with the capability. This ID can be conveniently used in a hash
**Fig. 2.2a Capability Formation**

**Fig. 2.2b Alternate scheme for capability formation**

**Fig. 2.2c CFO Formation**
coding scheme which indexes into a table of objects. The need for the
table is brought about because of the fact that many copies of a cap-
ability may exist and hence the actual object localizing information
has to be centralized so that address relocation or object removal may
be facilitated. Of course, natural size limitations prevent all the
objects from being defined in primary memory thus bringing about the
need for a backup table in secondary memory.

Summarizing, we have studied the structure of a capability,
its identification in storage and its creation. We have indicated three
classes of operations -- those valid on all capabilities, those valid
on all data elements and those valid on the objects defined by capab-
ilities. From the user point of view, a capability is simply the addr-
ness of a virtual object, from the system point of view, it localizes
the object referred to and specifies what can be done with the object.
The capability model we have specified serves as the basis for an anal-
ysis and implementation of capability structures.

II.3 Analysis of Capability Structures
a) Applications of capabilities Applications of the capability concept
break up into the two areas of context independent addressing [Fa 74]
and implementation of a protection base [Wu 74]. Since a capability
defines a system wide object, it provides a convenient mechanism for
referring to objects in shared procedures. Fabry [Fa 74] has shown the
inadequacy of segment table addressing for this purpose and traced it
to the lack of an efficient absolute address. A capability can be used,
both to provide the absolute address as well as an execution environ-
ment defined address, thus unifying the two requirements of shared
procedure addressing. Our requirement of multiprocessing primarily avails of this feature. However, the adaptability to protection is more vital to our implementation.

Let us first define a model for protection. As earlier defined, we can consider the process as an independently schedulable unit which represents a task of the user in a computer utility. The execution environment with respect to which the process executes is called a domain. The execution environment is an entity which specifies all the resources which are accessible to the process and the nature of access allowed to the process. On the assumption that all possible operations, defined on data, capabilities and resources, are executable by a process, the execution environment defines the \textit{valid operand space} of the process. The domain may be implied as in the case of a protection system using the access control list model, or it may be a physical and manipulable entity as in the case of a system using the capability scheme. To clarify, in the latter case, the domain can be specified as a list of capabilities which the process can avail of. As will be pointed out later (in chapter III), the term 'base' is used to define the static list mentioned above and the term 'domain', to refer to the dynamic operand space accessible.

Studies of protection have been made in the context of user interaction in a computer utility [Sc 72, La 74, Gr 68] but only marginal consideration given to the idea of using protection mechanisms as a basis for resource management [Re 74, Wu 74]. A protection system first needs to specify the basic unit of protection. If user interaction is the primary consideration, then this basic unit can be simply a segment (defined as a logical collection of information) in storage. In our
case, since resource management is the motivation, the basic unit of protection is a resource, which may be a logical entity like a process, a data structure, a base (the static version of a domain) or a physical entity like a device, or a segment of storage.

The flexibility of our model lies in the fact that it is a generalization of all protection systems. If user interaction is the basis, then protection in a computer utility is to guarantee total user separation when desired, allow unrestricted user cooperation if needed, and provide as many intermediate degrees of control [Sc 72].

The fact that a user's program and data can be considered as resources, allow us to satisfy the above requirements in the context of our capability structure. Protection studies emphasize the need for the execution environment to change according to the location and mode of control. The protection problem then resolves into a definition of the execution environment to allow this dynamic behaviour. Schroeder [Sc 72] has analysed the case of user interaction where each user is represented by a protected subsystem which is a collection of procedures and their associated data bases. Here, the effort has been directed towards the detection of the need for a dynamic change of execution environment and then providing this change automatically even for the worst case situation of two mutually suspicious protected subsystems.

The rationale for the development of our protected resource management system is that protection is to exist in a uniform manner throughout the system and not to be confined to specific entities. Following the HYDRA [Wu 74] concept, we delineate between protection which is a mechanism and security which is a policy defined on the usage of mechanisms. For example, we will provide a mechanism to imp-
lement the case of cooperation between mutually suspicious subsystems, but will not ensure this automatically.

b) **Limitations of capabilities and solutions to these** Capability structures have their limitations and although these can be overcome, it is at the expense of flexibility. A reasonable compromise can be effected between the two to arrive at a viable capability structure.

The first limitation is the ever increasing size of the capabilities. As elaborated earlier, the capability needs to contain three fields. We had implicitly taken care of the 'reference' field by centralizing the localizing information in a table. This is at the cost of flexibility for the ideal scheme would contain the definition of the resource within the capability itself. This definition can be in terms of other capabilities thus leading to the need of a variable capability size. The type field, at least in the case of extended capabilities, needs to contain the capability of a representative object. We forego this facility in the actual implementation and make the definition of the extended object identify its structure. The options field can of course be implemented as a bit string.

The inherent flexibility of capabilities causes the problem of revocation. The flexibility lies in the fact that permission to use a resource can be simply passed in the form of a copy of the original capability. The problem arises in later revocation, as then the borrowing domain has to be made to destroy all copies of the capability it has, as also any copies it has passed to other domains. Redell[Re 74] has provided a detailed analysis of this problem and we will use a modification of his solution. The point to be recognized is that revoc-
ation can be ensured if an indirect revokable version of the original capability can be passed, with a copy of the original form maintained in a indirection table. Revoking privileges from the copy in the indirection table also results in the effect of revoking the same privileges in the revokable version. This is because every time the revokable version is used, automatic traversal of the linkage path in the indirection table occurs until a non revokable version is obtained; with the privileges being a minimum subset of the options fields of all the capabilities in the traversal path. Any level of revocation is possible in this scheme. The demands of flexibility in the revocation scheme and the actual implementation are discussed in Chapter III.

Our primary motivation has been to provide a framework for resource management so that any operating system can be designed around this. This necessitates a need for a capability structure which can support extensions to it. This is obvious since the essence of a capability is it defines a resource and we are never going to know all the resources definable in an operating system. However, we can identify a set of base resources like storage, processes, devices etc. and define base level capabilities for these. The common scheme suggested for an extended capability is to encapsulate the capabilities defining the resource, within another capability [Re 74, Li 73]. Redell has analysed various schemes for capability extension and arrived at a unified scheme for both revocation and extension. Our scheme for capability extension does not rely on encapsulation but makes use of the architectural support to implement the extended capability. This is possible by a scheme of definition in a segment of storage in terms of already existing capabilities and the association of a set of
procedures operating on this definition. Thus basically the scheme relies on the mapping of a resource definition into a storage definition.

Finally, a problem for which an effective solution has not been proposed is the 'lost object' problem. This arises if all the capabilities defining an object are destroyed, thus making the object unusable and hence implying that it needs to be destroyed. It can be resolved partially by making use of the 'owner concept' wherein the creator of the object is made to be responsible for the funding of the object as well as ensuring that it is destroyed. An ineffective solution could be the maintenance of a reference count of the capabilities, which means capability destruction should be an explicit operation.

II.4 A generalized Pointer Number Machine

In this section, we will describe a generalized pointer number machine (GPN). The micro pointer number machine (Appendix A1,A2) is a particular instance of the generalized machine and will serve as the hardware base for the capability structure. The term pointer number machine arises from the observation that the system divides the set of all elements into two main classes— one which comprises of all data (integers in our implementation) elements and the other which comprises of descriptions of sets of data and description elements, much in the sense of data and capabilities in our capability model. These elements are either numbers or pointers to a set of numbers and pointers, hence the term pointer number machine. This term is equivalent to the term 'descriptor oriented architecture', where the descriptor is equated to a pointer. The system uses a scheme of tags to identify the classes of elements.
The set of Tagged elements Although all elements are divided into two classes, we will provide a further subdivision of the set of descriptor elements. We have introduced the notion of a capability as a definition of a resource. In the context of the GPN, we will use the term 'codeword' to denote the special case of a capability defining storage. We also introduce the term 'pointer' which is not to be confused with the earlier usage of the term, where it was equated to a descriptor. We will henceforth use the terms with the following interpretation:

- **CODEWORD** - controls access to storage segments
- **CAPABILITY** - controls access to all other resources
- **POINTER** - these elements are derived from codewords and provide access to storage at an elementary level
- **DESCRIPTOR** - the union of codeword, capability and pointer elements
- **NUMERIC** - data elements, integers in our implementation

The difference between a codeword and pointer is that the former provides the information using which the actual location of the segment can be derived. A pointer, on the other hand, defines a segment by providing the actual location and size of the segment. The five classes of elements are indicated with their tagged formats in Table 2.1.

1) **Numeric** This element is just an integer data element with its tag.
2) **Capability** This confirms to the earlier capability model, where the 'object identity' field provides the reference to the localizing information. The capability class indicates the class of the identified resource as shown. The interpretation of the various classes of resources is confined to later discussions. The GPN provides the definition of a set of base level resources with no support for capability extension.
Tag

0 | ------- | Value | NUMERIC (Integer)

1 | C | option | Object ident. | CAPABILITY

Where C = Capability class, e.g.

- Null
- Name
- Capability Forming
- Device
- Process
- Base

2 | T | Limit | FBL | ADDRESS POINTER

Where T = Segment class, e.g.

- Resource
- Mixed
- Control

Numeric, Sizes of 1, 2 or 4 byte elements

T also indicates if the segment is Store Protected

FBL = First Byte Location of segment

3 | (m) | (link) | FBL | CONTROL POINTER

(First instruction byte at FBL)

4 | T | option | MST Index | CODEWORD

Table 2.1 GPN tagged element formats
iii) **Codeword** This element identifies a segment in storage. The various classes of segments are to be interpreted as:

- **NUMERIC** - contains only Numeric elements
- **RESOURCE** - can contain Numeric, Capability and Codeword elements
- **CONTROL** - can contain Numeric elements, coded instructions and Name, Null and Capability forming capabilities
- **MIXED** - can contain Numeric and Descriptor elements

iv) **Address Pointer** We differentiate between address pointers pointing to segments in storage and the particular case of an address pointer pointing to a control segment. The 'class' interpretation is the same as in a codeword. In the case of NUMERIC segments, the 'T' field also contains a code indicating the size of the numeric elements. In other cases, the element size is predetermined. Pointers describe a segment using a 'base limit' format by indicating the size of the segment. Finally, both in the case of codewords and address pointers, 'options' information is provided in the 'T' field itself by indicating whether the segment is write protected or not.

v) **Control Pointer** These elements are used as operands in the case of control transfers or to point to the next instruction to be executed. Since the segment class, element size and access options (write protection) are implied in the case of control pointers, the 'T' field is not required. Moreover, since the pointer need not define a segment but just needs to point to one element, the 'limit' field is also not required. These fields are used to contain a 'mark' value and a dynamic link, the need for which will be explained when the operation of the associated process stack is considered.

A codeword element specifies the index in a system wide
Master Segment Table (MST). The entry in this table needs to contain the starting location of the segment, the size and the class specification. Thus in effect an entry needs to contain an address pointer defining the whole segment. In practical implementations*, the information required is very much reduced as some of it is already present in the codeword. The 'object identity' field in the case of a capability may actually be used to contain the resource as in the case of Null, Name and Capability Forming capabilities, and in other cases may be used to index into a table of codewords which define the definition segment of the capability.

b) The set of GPN operations (or instructions) By definition, the set of GPN machine instructions constitute a set of operations available without limit to all computations. However, the operations are defined to be valid only on particular classes of tagged elements. The set of operations fall into five categories:

i) Arithmetic and logical instructions provide the normal computational ability of the machine and are defined on Numeric elements or Address pointers which point to a numeric element. In the latter case, the numeric element pointed to is automatically fetched.

ii) Control transfer instructions essentially involve the transfer of control with the destination indicated by a control pointer or a label for local intra segment transfers. They also implement procedure call and return mechanisms.

iii) Tag and Type independent instructions ensure the copying of elements between registers or between store and register. In the latter

*as in the case of the micro PN machine (Appendix A1,A2)
case, restrictions implied by the pointer with respect to storage are ensured based on the class of element being transferred.

iv) Descriptor manipulating instructions enable the modification of descriptors. In the case of codewords and capabilities, modification is disallowed while in the case of pointers the modification is allowed as long as the new pointer formed describes a subset of the elements defined by the old pointer. A new tagged element is allowed to be formed by derivation from a capability forming capability. A set of display functions to interrogate the various descriptor fields, is also provided.

v) Resource management instructions deal with operations like the derivation of a pointer from a codeword, appending a resource to a base, formation of either a new segment or a new base or a new process and other such functions.

The GPN description is not meant as the specification of an architecture, but just to serve as a model for the architecture. We conceptualize this model as follows. The GPN architecture divides the set of all elements into five classes. However, it does not necessarily identify the sets of elements by affixing tags to them. It does identify elements in its register store by this mechanism, but the scheme for the identification mechanism in main storage is by controlling the derivation of a pointer. Thus it identifies a set of elements in storage by means of the pointer which can be derived to point to the set. So, it is conceivable that the same set may be construed as a different class of elements, but again with restrictions. For example, we may want to consider a set of halfword elements as a set of byte elements, in a Numeric segment, but surely not as a set of descriptor
elements. The main advantage of the above interpretation is that it reduces the overhead for the storage of numeric segments (which will form the bulk of storage); although, numeric elements present with other classes in a segment, still need to be identified by tags. Actual segment formats are illustrated in chapter III. The above scheme relies on the assumption that elements in storage can only be accessed by pointers.

We have shown the scheme by which the model identifies its operands. It should be recollected that an element pointed to by an operand is the operand in the first category of instructions, since automatic fetch occurs. The GPN then defines restrictions on the set of operations as applicable on the set of identified elements. As long as we can arrive at a criteria for protection, these restrictions can be formulated and arranged to show that the machine cannot violate protection. This is the rationale for the analysis of the micro PN machine in Appendix A3.

II.5 Applicability of the GPN model to the Capability Structure

We need to unify our capability and protection models with the GPN model defining the architectural support. From the capability model, we observe that the capabilities and codewords of the GPN model satisfy the requirements of a capability but this is not so of pointers. The heart of the GPN model is the 'concept' of pointers. From the GPN model there seems to be no obvious distinction between a pointer and a codeword, other than the fact that any address relocation scheme has to account for the pointers. The crucial difference is that access rights can be checked more elaborately at the indirection stage through
the Master Segment Table (MST) while, for the sake of efficiency, we
do not want this to be done for a pointer. For instance, a check can
be made for a synchronization constraint or a revocation constraint
at the MST. The pointer is then a 'super capability' in that the checks
on it are minimal. This implies that its existence has to be more
strictly controlled.

The protection requirement of a dynamic execution environ-
ment is satisfied by a 'base' capability. A process, which is a resource
itself, can only access the resources specified in the base with
respect to which it executes. A base changing facility can then support
the idea of a dynamic environment. The base itself is treated as a
resource and should contain a capability identifying it and which
permits the change operation. The GPN identifies a segment as the
basic unit of protection. The capability model generalization of this
requiring a resource to be the basic unit of protection is simply
effected by considering that every resource is defined by a description
segment and restricting access to this segment. In the case of base
level resources, the operations are available as machine instructions.
For extended resources defined by extended capabilities, operations
are available as procedures; the set of procedures being associated
with a storage structure by the definition segment. Thus capability
extension is supported by a scheme of mapping the resource into a
description segment.

II.6 Conclusion

We have set up a model for a capability structure based on
the guidelines of protection and shown its adaptability to a model
defining the architectural support. The resulting structure is to be referred to as the protection structure for the rest of the thesis. The essence of the development has been the notion of a capability which defines a resource by identifying the nature of the resource and a set of operations on it. The limitations of capabilities have been resolved with respect to a practical implementation, as well as providing a flexible scheme for revocation and capability extension. The architecture translates the abstract notion of a capability into the simpler task of storage management, thus making the resultant structure simple and economical. On the basis of the model developed, we can now provide a detailed specification of the protection structure which is to be implemented on the micro PN machine. All along the criteria for definition is the identification of the object and the specification of a set of operations on the object. This is the rationale for the protection structure design of chapter III.
III.1 Overview

This chapter describes a practical scheme for the extension of the hardware support into a layered protection structure. This includes a rigorous specification of the protection structure by a scheme of defining operations on tagged elements. The motivation behind the various mechanisms introduced is also indicated. The protection structure follows as a natural extension of the architecture. The layered implementation of the protection structure supports the idea that the demarcation between the hardware and the software may be made at any of the levels defined, suggesting the possibility of this being a design parameter in a trade-off between cost and efficiency. The processor itself needs to ensure protection by a scheme of hardware level checks on operands, which may be implemented as an additional software layer on a conventional machine.

The motivation between the description which follows can best be described as a resolution of the conflict between the problems of complexity in assembly language system programming (which increases the frequency of errors) and that of inefficiency with respect to code generated in using high level system programming languages. In programming in a language close to the machine architecture (like assembly language), the higher level constructs, essential to operating system design, are difficult to establish and recognize. In addition, to prove the correctness of a resultant program is a problem of equal magnitude, if not more, as designing the program itself. On the other hand, in programming with system programming languages, the features of the
architectural support is lost. The proposed protection structure with its intrinsic hardware support, provides a base for system programming which reflects the architecture as well as recognizes the constructs required by operating systems. The restrictions enforced by the layered nature of the protection structure and provisions for strict error confinement enable a validation of the correctness of operation. Finally a natural base for implementing protected user interaction is implicitly provided by the protection structure.

The detailed specification of the hardware MPN machine is found in Appendix A2. This is a modified version of a proposed machine called the micro PN machine (Appendix A1). The hardware machine formalizes the concept of a 'pointer number' machine introduced in chapter II. It is assumed in further discussions of the protection structure that the reader has a full understanding of the hardware support, specifically the instructions crucial to descriptor maintenance.

III.2 Elementary Resource Management Functions

The MPN instruction set (Refer Table 3.1) describes a set of operations which act on POINTERS or INTEGERS, but other than the restricted version of an existing pointer, no other pointer can be formed. Thus if we are to utilize the MPN machine as the hardware base, we need a set of resource management functions for the orderly formation of pointers from resources, at the same time maintaining protection constraints.

Resources in the MPN machine are described by CAPABILITYs and CODEWORDS. Codewords, as indicated in the previous chapter, are a specialized form of capability, used in store management, while
Table 3.1 MPN Instructions

<table>
<thead>
<tr>
<th>f,gh</th>
<th>Mnem.</th>
<th>Validity</th>
<th>Operation</th>
<th>Cond.Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GOTO</td>
<td>CURR.Lim</td>
<td>IP = IP + nS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DATA</td>
<td>CURR.Lim</td>
<td>X0.Tag = Cap, X0.Typ = Nam, X0L = f(IP+nS)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MODE'</td>
<td>X&amp;Aptr</td>
<td>X0 = X'n</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IFGO</td>
<td>CURR,Fbl</td>
<td>If c, IP = IP - 2n + 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3 Shifts</td>
<td>X&amp;Int</td>
<td>Left, Right, Right circular, Arith. left</td>
<td>OV, ZE, MI</td>
</tr>
<tr>
<td>0</td>
<td>4 Shifts</td>
<td>X&amp;Int</td>
<td>Arith. right by n bits</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Move</td>
<td>X&amp;Int</td>
<td>MOVEF: 4 bit field, MOVEB: byte</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Field</td>
<td>MOVEH: half word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TAG</td>
<td>Y&amp;CFC</td>
<td>X.Tag = Y(29:32), X.Typ = Y(25:28)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>COPY</td>
<td>X&amp;Int</td>
<td>X.Tag = Int, X(25:32) = n</td>
<td>OV, ZE, MI</td>
</tr>
<tr>
<td>6</td>
<td>Arith</td>
<td>X&amp;Int</td>
<td>ADD, AND, MASK, NEQ, SUBTRACT, OR, MVN</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4 LOAD</td>
<td>Y&amp;Aptr</td>
<td>MOVEF: 4 bit field, MOVEB: byte</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>STORE</td>
<td>Y&amp;Ptrw</td>
<td>g(Y) = X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>COPY</td>
<td>X = Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ITAG</td>
<td>X(29:32) = Y.Tag, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CHL</td>
<td>XH=0, XH=YH, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LCH</td>
<td>XH=YH, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MARK</td>
<td>IP.m = n, MP = SP, SP = null pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MOD</td>
<td>X&amp;Aptr</td>
<td>X = X'n</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>UNDU</td>
<td>SP</td>
<td>Unstack element into X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CASE</td>
<td>X&amp;Int</td>
<td>If 0&lt;X*n, IP = IP + 2X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>RETURN</td>
<td>X&amp;Int</td>
<td>Return control to mark n</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ADDI</td>
<td>X&amp;Int</td>
<td>X = X + nS</td>
<td>OV, ZE, MI</td>
</tr>
<tr>
<td>7</td>
<td>Dump</td>
<td>X&amp;Int</td>
<td>Stack element from X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>SKIP</td>
<td>CURR.Lim</td>
<td>If c, IP = IP + 2n + 2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CALL</td>
<td>X&amp;Ptrx</td>
<td>If n, Stack link, IP = X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>GOTO</td>
<td>X&amp;Ptrx</td>
<td>If n, IP = X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ITYP</td>
<td>X(29:32) = Y.Typ, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>INDEX</td>
<td>X(21:32) = Y.Lim, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Formats

<table>
<thead>
<tr>
<th>f,gh</th>
<th>n</th>
<th>f</th>
<th>X</th>
<th>g</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>X</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3n</td>
<td>f</td>
<td>c</td>
<td>n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operand Classes: |  |  |  |  |
| Mptr  = Byte| Byte| Hw| Hw| Res| Res| Cptr |
| Mptw  = Byte| Byte| Hw| Hw| Res| Res|     |
| Nptr  = Integer| Byte| Byte| Hw| Hw| Res| Res|     |
| Int   = Integer |   |   |   |   |   |   |
| Aptr  = Mptr| Mix| Mix| Mix| Mix| Mix| Mix|
| Cptr  = Control pointer |   |   |   |   |   |   |

Keys:
- X.Tag = Tag of the X register
- X.Typ = Type field of the X register
- X.Lim = Limit field if the X register contains a pointer
- X(n1:n2) = Bits n1 to n2 of the X register
- g(X) = element pointed to by pointer in X register
- X'n = Fbl field of the pointer in X is modified by n
capabilities define access to all other resources. The codeword identifies a segment in storage and the codeword class indicates the class to which the codeword belongs to. There are five classes of segments—CONTROL, NUMERIC, RESOURCE, MIXED and STRUCTURE. The last class is necessitated because of the protection structure and its relevance will be discussed later. A POINTER to the segment can be obtained by performing the EVAL operation on the associated codeword. The nature of the EVAL operation is indicated in the description of the micro PN machine (Appendix A1).

The Base Pointer (BP) points to a RESOURCE segment which specifies the accessible resources of the process. The term 'domain' represents the dynamic access available to a process executing with respect to a 'base' which provides the static definition of the execution environment. In order to reference within a base, we need a set of functions which allow the selection of a resource from the current base. The functions CONNECT and NAMF provide this facility. To allow for generality with respect to shared procedures, references to a resource in the current base need to be made possible through a NAME capability, which serves in the role of an identifier. This is necessitated by the fact that a shared procedure may be called by different processes executing with respect to different bases. Thus the shared procedure should be able to refer to a resource in the base by a Name capability, rather than by an index in the base. The calling process ensures, by the CONNECT operation, that the Name capability has been 'associated' with a resource in the base. Thus, any called procedure can form a POINTER to the resource in the base segment by applying the NAMF operation on the Name capability.
The requirements for the NAMF and CONNECT operations are as illustrated in Fig. 3.1. We will now consider various schemes to implement these two operations. The simple scheme would be for the Name capability to contain the index of an entry in the global name table. The CONNECT operation updates this entry to point to the associated resource and the NAMF operation obtains this entry as a result. However, with such a scheme, concurrent procedure sharing is not possible, as each process would require a different pointer in the global name table entry. To allow for procedure sharing, the table entry can point to a linked list of (base ID, pointer to resource) pairs, but this is an inefficient and inelegant scheme. The final scheme arrived at was to require the CONNECT operation to make an entry in the current base which associates the Name capability to the resource by some scheme consistent with the base specification. The NAMF operation scans the base and by means of this association, forms a pointer to the resource. The nature of the association will be explained when the base's structure is specified as part of the protection structure.

The EVAL operation as applied to a codeword is also subject to a constraint. The EVAL operation, as explained in Appendix A1, obtains the pointer to the defined segment by an indirection through the Master Segment Table (MST). The blocking bit in the MST entry, if set will cause the process using the EVAL function to be blocked. When the bit is reset, the process is allowed to execute and it sets the bit, blocking any other process. Hence, after a process has acquired access to a segment, it has to indicate when it no longer needs the access, so as to allow other processes to continue. This is done by a RELEASE operation, which clears the blocking bit and enables processes
Elementary Resource Management Functions

EVAL applies on a CODEWORD and produces a POINTER (APTR or CPTR).

CONNECT applies on a NAME CAPABILITY, INTEGER and associates the NAME capability with a resource indexed by the integer.

NAMF applies on a NAME CAPABILITY and produces a POINTER to the associated resource or a NULL capability if no resource is associated.

REG applies on a CRA CAPABILITY and loads (stores) the hardware control register specified by the CRA cap. into (from) a general purpose register.

RELEASE applies on a CODEWORD and releases the blocking condition in the segment.

Conceptual view of NAMF and CONNECT operations

![Diagram showing the conceptual view of NAMF and CONNECT operations.](image)

**Fig. 3.1** Elementary Resource Management Functions
blocked on the segment to continue executing. It is seen that this form of busy waiting for a process is inefficient. Of course, when a process needs a 'read only' access, it need not set the blocking bit but this gives rise to the additional complication that when a process requires 'exclusive' access, it has to ensure that no other processes have 'read' access at the same time.

Finally, in order to access the hardware control registers, the REG function is provided. It is defined to operate on a CRA (control register access) capability which indicates the nature of the access allowed in its 'reference' field. The CRA capability format will be discussed in the context of the protection structure.

The simplified set of resource management functions introduced, together with the modified MPN machine (Appendix A1,A2) forms the foundation for the protection structure to be discussed in the rest of the chapter. The design of the protection structure will require modification of the MPN element format, a new interpretation of the MPN instructions and a generalized set of resource management functions.

The essential characteristics of the system described thus far are:

1) Dynamic revocation is not possible. Once a base obtains a resource, it can maintain possession throughout its existence.

2) Any extension to the set of operations is implemented as procedures and activated by procedure call.

3) When an EVAL operation is applied on a codeword, a blocking condition in the segment defined by this codeword causes the process to wait on the processor until the using process releases the segment.

4) Maintenance of pointers is assumed to be done by the process
itself, for instance, a process may release a segment but still have pointers to the segment.

The main implication of the above features is that they rely on trust between processes giving rise to the possibility of processes sabotaging one another, since a malfunction in one could affect others. Since the MPN instruction set securely handles the maintenance of pointers, the weak link is in their formation. Moreover, greater flexibility is required in handling resources. It is towards these objectives that the protection structure is directed.

III.3 The Process State Vector

Before discussing the protection structure extension to the MPN machine, we first need to consider the process state vector (PSV) and the operation of the associated stack. The current state of a process is described by the PSV and the process stack is part of it (Fig. 3.2). The format of the elements in Fig 3.2 corresponds to the revised element format of Table 3.2. The need for this revised format will be discussed in the next section as part of the protection structure specification. The protection structure extension to the MPN machine together with the MPN machine will be called the 'extended pointer number system' (EPN), with all the MPN instructions being valid for it.

The process state vector (Fig. 3.2) contains the process capability, the four processor control registers (Instruction pointer (IP), Mark pointer (MP), Stack pointer (SP) and Base pointer (BP)), the sixteen general purpose registers (GPRs) -- X0 to X15, the extensions to the general purpose and control registers (the need for these will be explained shortly), the current stack and the extended stack.
Fig. 3.2 The Process State Vector
The two stacks together make up the process stack.

As established in the MPN machine description, an element can be accessed from memory only if a pointer pointing to it exists in one of the GPRs. Pointers were restricted to exist only in the processor registers or the stack segment but not in memory. However, in actual implementation, only the GPRs and control registers exist as hardware registers (Fig. 3.5), with the rest of the PSV existing in memory as a MIXED segment.

Restrictions for pointer maintenance arise because they are the only tagged elements which provide direct access to the storage segment. All other elements (other than INTEGERS which are a form of resource) refer to a resource by a scheme of indirection and hence access check is possible at the level of indirection. This direct access to storage has associated with it, a more serious problem. As indicated, a pointer describing a set of elements comprising a segment is first formed as a result of the EVAL operation on the codeword defining the segment. From this initial pointer, restricted pointers describing a subset of elements can be formed, but the pointer does not retain any information about the segment to which it points. This information is needed in certain cases of updating pointers and is therefore provided by associating an extension to every pointer in existence. This extension contains the codeword to the segment, from which the pointer was derived.

Since pointers can only exist in the segment defined by the process state vector, we establish a set of STACK CONVENTIONS:

1) A pointer in any of the control or processor registers has associated with it an extension in the PSV which contains the codeword
defining the segment to which the pointer points.

2) Whenever a pointer is stacked into the current stack, its register extension from the PSV is also stacked.

3) Whenever a pointer is unstacked into a register, the associated codeword extension is entered into the register extension.

4) A pointer when formed in a register by the EVAL operation, also causes the codeword to be entered into the register extension.

5) The extension of an element which is not a pointer is a NULL capability.

6) The extensions to the control pointers are:
   Instruction pointer: codeword to the current control segment
   Stack pointer: codeword to the current PSV segment
   Mark pointer: -do*-.
   Base pointer: codeword to the base segment

The process stack maintains a dynamic link between the various procedure activations by storing marked links (Fig. 3.2). The current stack defines the active portion of the process stack. The marked link chain can be used to return to a predetermined restart point. The nature of the MARK, CALL operations are illustrated in Fig. 3.3.

Attention may be focussed on the fact that the MARK and CALL need not be consecutive instructions. In this case, the MARK operation seals the current stack, stores the mark value in the instruction pointer and leaves space for the dynamic link. The stack pointer now describes a new current stack of length zero. Thus, at any stage, the Stack pointer defines the current stack after the last MARK instruction was executed, while the Mark pointer defines the current stack which existed at the time the last MARK instruction was executed. A subsequent *do. is used as an abbreviation for 'ditto' throughout the thesis
MARK m:

\[
\begin{align*}
\text{MP} &= \text{SP} \\
\text{SP.LIMIT} &= 0 \\
\text{SP.FBL} &= \text{SP.FBL} - 8 \\
\text{CP.MARK} &= m
\end{align*}
\]

where,

\[\text{!MP} \text{ means the element pointed to by the Mark Pointer} \]
\[\text{CWD (CP) means the codeword associated with the CP as an extension} \]
\[\text{SP.x means the 'x' field of the SP register} \]

Fig. 3.3 MARK and CALL operations on the PSV.
CALL operation stores the current Instruction pointer with its associated codeword in the space for the link provided (pointed to by the Mark pointer) and then transfers control.

The RETURN mechanism is more complicated as it involves searching down the dynamic link until a link, with mark value greater than or equal to the return value is found. This enables a return to any restart point desired. Since the highest return value allowed is 7, a link with mark 7 at the bottom of the extended stack ensures that control always returns to some point. The RETURN operation is specified in Fig. 3.4.

In the implementation of the RETURN function, the pointer in $X_b$ and the Mark pointer with the new link can both be created by operating with respect to the pointer describing the whole PSV, which includes the process stack. For efficient code however, the TAG operation may be used to construct the new pointer. For this purpose, the RETURN instruction, like all other EPN operations, is implemented in the privileged mode. This enables the use of the TAG operation on a suitable CFC capability.

The various hardware processor registers are shown in Fig. 3.5. Apart from the four control registers and the sixteen general purpose registers, we also have a PSVR register which contains the pointer describing the PSV, a CURR register describing the current control segment, a PI register which contains the process capability of the executing process and an INST register which contains the current instruction. Those registers which can contain pointers have extensions associated with them.
**III-9a**

**RETURN n :**

1. **START**
2. \( X_a - MP \)
3. \( X_b - MP'(MP,Limit) \)
4. Form Ptr. in \( X_b \) to \( ^\prime(X_b+1) \)
5. \( MP - X_b \)
6. \( MP,Limit - X_b,Link \)
7. **Is**
   - **Yes**
     - \( SP - X_{a'}^2 \)
   - **No**
8. \( CP - ^\prime X_a \)
9. **STOP**

**Registers at Stage 1**

- **SP**
  - \( r \)
  - \( q \)
- **X_a**
  - \( n \)
- **X_b**
  - \( m \)
  - \( n \)
  - \( i \)
- **MP**
  - \( q+r \)
  - \( i' \)
  - \( i+n \)
- **FSV**

**Notation**

- \( X^n \) : MOD operation on pointer in \( X \)
- \( ^X \) : element pointed to by \( X \)
- \( X.x \) : \( x \) field of the \( X \) register
- \( ^X.x \) : \( x \) field of element pointed to by \( X \)

\( X_a, X_b \) are registers reserved for system use

**Fig. 3.4 The RETURN Operation**
4 4 8 16 : No. of bits.

SP
Mix - Stack Pointer

MP
Mix - Mark Pointer

PSVR
Mix - PSVR register (defines the PSV)

CURR
Hwdrl - CURR register (defines the current control segment)

IP
Cptr m CC - Instruction Pointer (points to the next instruction to be executed)

X0 - X0

-------------------------- 16 General Purpose Registers

X15 - X15

BP
Res - Base Pointer (specifies the execution environment)

PI
Cap Pro - Process Identity (contains the process capability of the resident process)

INST
- Instruction register (contains the current instruction being executed)

Fig. 3.5 The Hardware Registers
III.4  The Protection Structure

The protection structure will be specified by dividing the elements into various classes, identified by tags and then defining a set of operations for each element class. The various classes of elements are indicated in Table 3.2 and the rest of the section is devoted to the consideration of the element format and the explanation of the rationale behind the operations defined for the various classes of elements.

a) Integer (INT) elements

The integer element remains unchanged from the MPN format, where integers are treated either as 16 bit or 8 bit numbers and contained in the 'N' field of the element. The rest of the fields are:

\[ \text{INT.U} = \text{user tag available for any purpose to the user} \]

\[ \text{INT.O} = \text{overflow byte used in multiplication and division operations} \]

Although, most usable information exists as integers, the conclusion, that the tagged element scheme effectively doubles the storage requirement (as only half the element size is used), is erroneous. This is because an element in storage is identified by the pointer which points to it and not necessarily by a TAG affixed to the element. This is illustrated in Fig. 3.6 where the actual structure of four classes of segments is shown*. From this it is obvious that overhead in storage for NUMERIC (data) segments is negligible.

Numeric and Control segments are restricted to start at halfword boundaries while resource and mixed segments must start at word boundaries. Control segments may in addition to instructions, contain Name

* The format for STRUCTURE segments is similar to that of a RESOURCE segment and will be introduced later.
### Table 3.2  EPN Tagged Element Format

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Field Name</th>
<th>Tag Typ Options</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abbreviation</td>
<td>No. of bits:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sub-fields:</td>
<td></td>
</tr>
</tbody>
</table>

The convention denoted above will be used in the specifications.

#### Integer
- **Tag**: U
- **Tag Mark**: 0
- **Limit**: N
- No. of bits: 4

#### Control Pointer
- **Tag Mark**: Link
- **Limit**: FBL
- No. of bits: 4

#### Codeword
- **Tag Mark**: Identity
- No. of bits: 4

#### Revokable Codeword
- **Tag Mark**: Identity
- No. of bits: 4

#### Base Level Capability
- **Tag Mark**: Use
- No. of bits: 4

#### Revokable Base Level Capability
- **Tag Mark**: Use
- No. of bits: 4

#### Extended Capability
- **Tag Mark**: EPriv
- No. of bits: 4

#### Revokable Extended Capability
- **Tag Mark**: EPriv
- No. of bits: 4

**ADDRESS POINTERS: APTR**

- **Byte Pointer**

- **Mixed Pointer (read only)**

APTRs are: BYTE, BYTER, HWD, HWDR, RES, RESR, MIX, MIXR.
Fig. 3.6 Segment Structure in Storage
capabilities. However, only the ID part of the Name capability (lower halfword) is contained in the control segment with the higher halfword being added on automatically when the DATA operation is used to form the Name capability.

The first element of each segment contains the limit value of the segment and the tag of the pointer to be formed, which is used by the EVAL operation to form a pointer defining the rest of the segment. As seen, the pointer 'tag' and 'limit' information is provided in the 'value' portion of the first element, which is an integer, in the case of Resource and Mixed segments.

b) Control Pointer (CPTR) elements

Control pointer elements have the same format except for bit R which, if not set, prevents READ access to the process. Since, in terms of MPN instructions, a READ access is effected through the LOAD instruction, the instruction fails if the operand is a CPTR with the 'R' bit not set. The 'P' bit if set indicates a privileged mode of operation, the mechanism for setting this bit being by hardware intervention. This is elaborated when considering the implementation of the protection structure operations. The privileged mode is only used for the implementation code for the various protection structure operations and later on to implement features of the nucleus. Any of the higher levels are unaware of the existence of the privileged mode. Understandably, we cannot have an instruction to set the P bit as then any process would be able to enter the privileged mode.

c) Codeword (CWD) and Revokable codeword (RCOD) elements

A codeword is a specification of the allowed access on a storage segment. In the
MPN, a codeword specified the class of a segment and contained the index of an entry in the Master Segment Table (MST), which indicated the starting location of the segment. The EVAL operation when applied on a codeword, produced the Pointer to the segment by evaluating the linkage. In the EPN, we extend the codeword format to include a set of access options, the need for which becomes evident when schemes for revocation, capability expansion, mutual exclusion and cross domain transfers are considered.

Table 3.3 indicates the interpretation of the various fields in the format for a codeword. The 'typ' field specifies the class of the defined segment. The necessity for a revokable version (RCOD) and the relative form of a codeword will become apparent during the progress of this discussion. Since a codeword defines the resource of storage, it is imperative that there be flexibility in handling it by providing for revocation of privileges in a codeword.

The requirements of a revocation scheme are:

1) The revocation scheme needs to differentiate between the two operations which can be used to duplicate a codeword. One is the COPY instruction of the MPN which forms an identical copy of the original, with the same privileges. The other operation is one which forms the revokable version of a codeword.

2) Any copy of the revokable version formed may be used to revoke privileges from the revokable version and all its copies, by applying a specific operation.

3) On the other hand, an operation is needed to prevent a copy of the revokable version from affecting the original revokable version and any other copies. Thus revocation on this element affects the
Table 3.3 Interpretation of the fields in the CWD, RCOD formats

<table>
<thead>
<tr>
<th>Codeword (CWD)</th>
<th>Tag Typ Options</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 Cl</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Revokable Codeword (RCOD)</th>
<th>Tag Typ Options</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 Cl</td>
<td></td>
</tr>
</tbody>
</table>

The fields are -

Cl : indicates the codeword (or RCOD) class which can be NUMERIC, CONTROL, RESOURCE, MIXED or STRUCTURE.

Options : if the corresponding bit is set, then the indicated operation is allowed -

R- READ operation is allowed (LOAD instruction is valid)
W- WRITE operation is allowed (STORE instruction is valid)
E- EXECUTE operation is allowed (for CONTROL segments)
V- EVAL operation is allowed (a POINTER can be formed)
D- Cross domain allowed using the TRANSD operation (for CONTROL segments)

P- CWD or RCOD can be accessed in either 'privileged' or 'normal' mode. If this bit is not set, none of the other operations are valid for the normal mode.

For codewords -

Index : is the index of the Master Segment Table entry.

Y : if this bit is set, it indicates the relative form of the codeword. In this case, 'Index' is the offset of the defined segment from the current location of the CWD.

For revokable codewords -

ID : is the index into a Table of Revokable Codewords.

L : is a lock bit which is used in the revocation mechanism.
privileges of that element only and none of the others.

4) Finally, it should be possible to form the revokable version of a revokable version. Thus revocation is to be supported to any level.

Summarizing the requirements, we need three operations in the revocation scheme— one to form a revokable version, one to revoke privileges from a revokable version or its copy and one to prevent a copy of the revokable version from affecting other copies. These are met by providing a new tagged element, ROOD, which is a revokable codeword and by using a lock bit 'L' in the ROOD format. We can then define three operations— SEAL, REVOKE and LOCK.

The introduction of a new tagged element does not mask the identity of the original tagged element. This is because all operations valid on codewords are also valid on RCODs, since a RCOD can only correspond to the revokable version of a codeword*. The only difference is that the EVAL operation has to go through an extra level of indirection. The REVOKE operation is defined to reduce the access options of a ROOD element (it is not valid on a CWD element). The SEAL operation produces a revokable version of a CWD or ROOD element. The LOCK operation prevents a RCOD element from affecting any other copies of it which may exist.

Revocation is effected through a scheme of indirection through a Table of revokable codewords. A RCOD element contains in its ID field, an index into this table. The entry in this table contains the original CWD or ROOD from which the revokable version was formed. The SEAL operation on a CWD or a RCOD creates a new entry in the Table of

* For instance a capability will have its own revokable version identified by an unique tag.
RCODs, updates this entry with the CWD or RCOD and finally forms a new RCOD element which indexes this entry. The 'options' field and the 'class' field are the same as the original version, but the lock bit is not set. This satisfies the first requirement.

The logic for the REVOKE operation is as follows:

IF RCOD.L = 1 THEN REVOKE ▪ Downgrading of the OPTIONS in the RCOD only
ELSE REVOKE ▪ Downgrading of the OPTIONS in both the RCOD as well as in the element which is in the table entry indexed by RCOD.ID

Thus, once a RCOD element is formed by the SEAL operation; then, as long as the 'L' bit is not set, revocation on the RCOD or any copy of it causes the revocation to affect the element in the Table, implying that all the copies have the corresponding privilege revoked. An INT element which is the second operand of the REVOKE operation serves as the mask for a logical AND operation with the 'options' field of the RCOD element. Thus, the second requirement is satisfied.

The LOCK operation sets the lock bit 'L' and hence from the logic of the REVOKE operation, it is seen that only that RCOD element is affected and not any copies. This satisfies the third requirement. Finally, since SEAL can be applied on either a CWD or a RCOD element to form a revokable version, nesting to any level is possible. This satisfies the last requirement.

The effect of the SEAL, REVOKE and LOCK operations are illustrated in Fig. 3.7. It should be noted that the REVOKE operation reduces the access options of a RCOD or CWD in the indirection path of the RCOD element on which it is applied. The assumption here is that
Initially \( X_1 \) contains a codeword with only the 'read, write' (RW) options set. \( X_5 \) and \( X_6 \) contain integers which provide masks for 'read only' and 'no options' respectively. \( X_2 \) contains a copy of the codeword in \( X_1 \).

**Stage I**

<table>
<thead>
<tr>
<th>Stage I</th>
<th>Stage II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Recods</td>
<td>Table</td>
</tr>
<tr>
<td></td>
<td>(lock bit set)</td>
</tr>
<tr>
<td></td>
<td>( X_4 )</td>
</tr>
<tr>
<td>( X_2 )</td>
<td>SEAL ( X_2 )</td>
</tr>
<tr>
<td>( X_3 )</td>
<td>&amp; COPY ( X_3, X_2 )</td>
</tr>
<tr>
<td>( X_4 )</td>
<td>&amp; COPY ( X_4, X_2 )</td>
</tr>
<tr>
<td>( X_5 )</td>
<td>LOCK ( X_4 )</td>
</tr>
<tr>
<td>( X_6 )</td>
<td></td>
</tr>
</tbody>
</table>

\( X_4 \) indicates Revokable codeword (short form for REVOKE).

**Stage III**

<table>
<thead>
<tr>
<th>Stage III</th>
<th>Stage IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Table</td>
</tr>
<tr>
<td>( X_4 )</td>
<td>LOCK ( X_2 ) &amp; REVK ( X_2, X_5 )</td>
</tr>
<tr>
<td>( X_3 )</td>
<td>&amp; REVK ( X_3, X_5 )</td>
</tr>
<tr>
<td>( X_2 )</td>
<td></td>
</tr>
<tr>
<td>( X_1 )</td>
<td>REVK ( X_3, X_6 )</td>
</tr>
</tbody>
</table>

**Fig. 3.7 Illustration of the SEAL, REVOKE and LOCK operations**
the EVAL operation when applied on a ROOD element, follows the indirection path, at each stage masking the access options fields of the CWD or ROOD with the field which it already has, till finally a pointer to the segment is formed satisfying the access options till the last stage. For example, referring to Fig. 3.7, if the EVAL function is applied on X₂ at each of the four stages, the results are as follows:

Stage I - no indirection and READ, WRITE access granted
Stage II - indirection with READ, WRITE access granted
Stage III - -do-
Stage IV - indirection with no access granted

The implementation of the SEAL and REVOKE operations are shown in Figs. 3.8 and 3.9.

In the description of the codeword format, it is stated that the 'Y' bit if set indicated a relative form of the codeword. The motivation behind this is the need to represent data structure in one segment and since pointers are not allowed for memory representation an alternate means of representation is required. The ability to represent structures in one segment facilitates efficient store management as well as supports capability extension (this will be explained later). Essentially, the idea is that once a pointer to a segment has been formed by an EVAL operation on the codeword, pointers to sections within the segment can be formed at an reduced overhead. There is a loss of flexibility in using relative codewords as it is no longer possible to use the revocation facility on components of the structure. This does not imply that selective control of access options is not possible with respect to structures, but only that these access options have to be decided at the time the structure is formed. The correspondence
Fig. 3.8 The SEAL operation
REVOKE \( x_b, x_a \)

Is\( x_a \) an INT

Yes

Is\( x_b \) a RECOD

Yes

Logical AND low order 7 bits of INT with the 7 bits of the options field in the RECOD in \( x_b \)

Is lock bit set in RECOD

Yes

Form pointer to the entry in the Table of RECODs using RECOD.ID

Logical AND low order 7 bits of INT with the 7 bits of the options field of the table entry

Stop

Fail

Fail

Fig. 3.9 The REVOKE operation
between structure representation using normal codewords and that using relative codewords is illustrated in Figs. 3.10 and 3.11.

The intrinsic nature of codewords is such that all structures are formed as tree structures. We therefore introduce the GROUP operation, which is responsible for collapsing a level of the tree structure, combining two levels and representing it as a segment containing relative codewords. Such a segment is of the distinct class STRUCTURE, the need for which will be indicated.

Considering the tree structure of Fig. 3.10, it is seen that three levels of the structure exist. Codeword A comprises the zeroth level, segment A the first level and segments B,C,D the second level. By applying the GROUP operation on codeword A, the first and second levels are compacted into one level and represented within a STRUCTURE segment (STR) as illustrated in Fig. 3.11. This segment also contains codeword A in relative form and is defined by a new codeword, Y, which is of type STRUCTURE and which now forms the zeroth level. Any elements other than codewords in level 1 of the structure are unaffected as also any codewords in the second level of the structure.

A complication arises in the GROUP operation because of the fact that there may be other codewords existing which define segments in the levels collapsed. Either of the following can serve as a solution:

i) All the codewords defining the concerned segments can be located and invalidated.

ii) The GROUP operation can just form a copy of the segments in the levels collapsed.

Considering the impracticality of the first, the second solution is resorted to. Furthermore, if revokable codewords are pre-
Assuming CWD A is in X1 (reg.),
the following code sequence
produces pointers to the segs.
A,B,C,D in X3,X4,X5,X6 respectively:

\[
\begin{align*}
\text{EVAL} & \ X3, X1 \\
\text{COPY} & \ X4, X3 \\
\text{EVAL} & \ X4, X4 \\
\text{COPY} & \ X5, X3'1 \quad \text{actually} \quad \begin{cases}
\text{COPY} & X_t, X3 \\
\text{MOD} & X_t, 1 \\
\text{COPY} & X5, X3'2 \quad \text{COPY} & X5, X3 \\
\text{EVAL} & X6, X6 \\
\end{cases} \\
\end{align*}
\]

(destination (source register) register)

Fig 3.10 Structure representation using Normal Codewords
Initially X1 contains the STR codeword \[ \text{CwdStr} | b' \]

Y bit (this bit is shaded if it is set)

```
\begin{figure}
\centering
\includegraphics[width=\textwidth]{structure_representation_with_relative_codewords.png}
\caption{Structure representation with Relative Codewords}
\end{figure}
```

Pointers to segs. A, B, C, D are formed in regs. X3, X4, X5, X6
sent in the first level segment (seg. A of Fig. 3.10), the GROUP operation needs to fail. This is because the revokable codeword may later be revoked, but the STRUCTURE segment formed by the GROUP operation still retains a copy of it. For the same reason, the GROUP operation fails if a codeword in the first level does not have the bit corresponding to the EVAL operation set. The implementation of the GROUP operation is specified in Fig. 3.12.

Access to components of the tree structure within a structure segment is provided by the EXPAN operation. First, a pointer to the structure segment needs to be formed by the EVAL operation on the defining STR codeword. The resultant pointer formed describes only one element, namely the codeword in relative form which originally comprised the zeroth level (Cwd. A in Fig. 3.11). We see here the necessity for a distinct segment class. If the segment was simply considered as a RESOURCE segment, the EVAL operation would form a resource pointer describing the whole segment. This resource pointer may then be modified (using the MOD instruction) to point into NUMERIC segments (like segments B,C in Fig. 3.11) and hence the components of these will be interpreted as tagged elements in a resource segment, thus causing a violation of protection. By defining a STRUCTURE codeword, the resultant resource pointer formed by the EVAL operation can be constricted to describe only one element.

The EXPAN operation applies on a resource pointer which points to a relative codeword. The resulting pointer describes the whole segment defined by the relative codeword. Thus this operation is equivalent to the EVAL operation defined on normal codewords. The resource pointer formed by the EVAL operation on a STR codeword has a 'read only'
Assume register X contains the zeroth level codeword, say CWD A which describes segment A at the first level. Reg. Y will contain the new codeword formed.

NOTE: The original first level segment is used (Seg.A) and not a copy.

Fig. 3.12 The GROUP operation
restriction which prevents the relative codeword from being replaced. Fig. 3.11 illustrates the use of the EXPAN operation in order to access components of the structure. The implementation of the EXPAN operation is indicated in Fig. 3.13.

We have been discussing operations on codewords under the implicit assumption that the codeword is already present in the domain. Since formation of a codeword is not allowed, the problem is how to legitimately generate a codeword. We propose that, request for all resources be made by calling a RESOURCE MANAGER. This routine is not part of the protection structure and is implemented in a higher level software layer. Thus the necessary book-keeping operations are performed by this routine and it returns a codeword.

The resource manager returns along with the codeword, an INC capability. For the convenience of the EVAL operation, the entry in the Master Segment Table maintains a set of four access options—read (R), write (W), execute (E) and mutual exclusion (M). The necessity for these is clarified when the implementation of the EVAL operation is considered. The INC capability is returned to the creator of a segment and contains the MST index corresponding to the segment. Initially when the segment is created all the access options in the MST entry are enabled. The owner can mask some of them by a DOWNGRADE operation. It can later upgrade these options by a UPGRADE operation. In either case, the second operand is an integer which indicates the options affected.

Summarizing, the operations introduced as applicable on codewords (normal, revokable and relative) are:

EVAL - applies on a normal codeword (Y bit=0) or a RCODE and produces a pointer to the segment referred to.
The EXPAN operation

**EXPAN X, Y**

Is Y a Resource Ptr.
- No → Fail
- Yes → Load element pointed by Y into register X.

Is X a a codeword
- No → Fail
- Yes → Is X a relative CWD
- No → Fail
- Yes → Add to the FBL field of the pointer in reg. Y, the value in the Index field of X and form the resulting ptr. in X.

Move low order 16 bits of the INT element pointed by X to the high order 16 bits of X. (final ptr. will be in X)

X,Fbl = X,Fbl + 4
(X now points to the next word)

Stop
SEAL - applies on a CWD (normal) or a ROOD and forms a revokable version of the CWD or ROOD.

REVOKE - applies on a ROOD and revokes the access options as indicated by a mask specified as an integer element.

LOCK - applies on a ROOD and sets the lock bit in the ROOD, localizing the effect of revocation to the ROOD only.

EXPAN - indirectly applies on a relative codeword (directly on the resource pointer describing the relative codeword) and produces a pointer to the segment defined by the relative codeword.

GROUP - applies on a RESOURCE codeword (normal) and produces a relative codeword form of the structure defined by the codeword. The new segment formed is of class STRUCTURE and is defined by a STR codeword.

At this stage we have introduced one more complication in the EVAL operation, namely that it has to traverse through levels of indirection if it applies on a ROOD element. As evident from earlier discussions, the EVAL is one of the basic operations of the protection structure and requires modification often when new features are introduced. Thus the final specification of this operation is postponed to later discussions. Henceforth, unless specifically stated, we will use the term 'codeword' to refer to the codeword element as well as the revokable codeword element, since all further operations involving codewords do not differentiate between the two forms.

d) Base level Capabilities (CAP,RCAP) Capabilities define access to all resources other than storage. These resources may be physical like devices or logical like processes. The term 'base level' refers to the fact that these elements are used to define 'known' resources. Later on
the concept of extended capabilities is used to define arbitrary resources which can be created. The TYP field in the capability format indicates the type of resource for which access is defined. The USE field is reserved for type dependent use in the capability. The ID field of the capability is usually an index into a table of capabilities of that particular type. The interpretation of the various fields for different types of capabilities is indicated in Table 3.4.

Following the established strategy, we will discuss the capability types by specifying the operations valid for that type:

1) **Null** Any operation on a NULL capability, except a tag independent operation, causes a failure which may be implemented as a trap to an error routine or a RETURN to the process stack for user defined error handling. The ID field of the capability can be used to contain a code indicating how it was set to null.

2) **Capability Forming Capability (CFC)** The use of the CFC has already been introduced in connection with the TAG operation, which is the only operation valid on it. It contains the TAG and TYPE of the new tagged element to be formed. It is therefore the authority for the possessor to create elements of that particular TAG and TYPE. The USE field, if zero, implies that the capability can only be used in the privileged mode. Normally, for reasons of economy in storage, each base need only contain a CFC, set for privileged use and which allows the formation of another CFC, thus enabling the implementation routines to build whatever tagged elements are required. These implementation routines execute in privileged mode, so that the CFC can be set for privileged use only.

3) **Device capabilities (DEV)** The Device capability is used for I/O
### Table 3.4 Base level Capabilities

<table>
<thead>
<tr>
<th>Type</th>
<th>USE (Bit 0-6)</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) NUL</td>
<td>-</td>
<td>Error code</td>
</tr>
<tr>
<td>2) CFC</td>
<td>ZERO → privileged mode only</td>
<td>Tag and TYPE of new element to be formed</td>
</tr>
<tr>
<td>3) DEV</td>
<td>Options on using the device</td>
<td>Index into a table of ISVs</td>
</tr>
<tr>
<td>4) PRO</td>
<td>if set Bit 6 → START allowed do. 5 → STOP do. do. 4 → CONTINUE do. do. 3 → TERMINATE do. do. 2 → INITIALIZE do. do. 1 → VALIDATE do. do. 0 → COMMUNICATE do.</td>
<td>Index into a table of PSVs</td>
</tr>
<tr>
<td>5) BAS</td>
<td>if set Bit 6 → CHANGEB allowed do. 5 → TRANS do. do. 4 → APPEND do. do. 3 → DELETE do. do. 2 → START do. do. 1 → CONNECT do.</td>
<td>Index into a Table of codewords describing Base segments</td>
</tr>
<tr>
<td>6) NAM</td>
<td>Index of associated codeword when used in name definition</td>
<td>Identifying Name ID</td>
</tr>
<tr>
<td>7) CRA</td>
<td>ZERO → privileged mode only</td>
<td>Code specifying the nature of access allowed.</td>
</tr>
<tr>
<td>8) PFC</td>
<td>-</td>
<td>ID of the process capability to be formed</td>
</tr>
<tr>
<td>9) BFC</td>
<td>-</td>
<td>ID of the Base capability to be formed</td>
</tr>
<tr>
<td>10) EFC</td>
<td>Identification of standard extended capabilities</td>
<td>ID of the new extended capability to be formed</td>
</tr>
<tr>
<td>11) INC</td>
<td>if set Bit 6 → DOWNGRADE allowed do. 5 → UPGRADE do.</td>
<td>MST index of the segment it corresponds to</td>
</tr>
</tbody>
</table>
operations and contains an index into a table of device state vectors. The device state vector describes the device, the USE field being used to indicate the access options allowed on the device. The DEV capability has two operations defined on it:

\[
\text{IN (DEV Cap., CWD)} - \text{transfers data into the buffer defined by the codeword, from the device.}
\]

\[
\text{OUT (DEV Cap., CWD)} - \text{transfers data from the buffer defined by the codeword, into the device.}
\]

These operations may be implemented as system calls, causing the buffer segment to be locked out during device operation. This will be considered in greater detail in the next chapter in the context of the nucleus.

4) Process Capability (PRO) The process capability indexes into a table of descriptor elements, each of which describe the Process State Vector (PSV) corresponding to a particular process. The PSV, as shown earlier, describes the current state of the process. Since the policy defining the operating system is to be designed as a set of concurrent processes executing in an environment provided by the 'nucleus', we need operations to be defined on process capabilities which provide for efficient cooperation, communication and control between processes. We will therefore postpone the discussion of these operations to the next chapter where the features to be provided by the nucleus are considered.

At this stage, we list the operations and associate a bit in the 7 bit USE field with each operation, the assumption being that an operation is allowed only if the corresponding bit in the USE bit of the PRO capability is set. The operations defined on a process capability are:

- COMMUNICATE
- INITIALIZE
- START
- STOP
- CONTINUE
- TERMINATE
- VALIDATE

A process capability defining an uninitialized PSV is initially formed
as result of an operation defined on a process forming capability.

5) **Base Capability (BAS)** The base capability controls access to a base in which the process can execute. The ID field indexes into a Table of Bases, while the USE field indicates the OPTIONS allowed on the base capability in terms of operations defined on the base capability. Each entry in the Table of bases contains a descriptor for a base segment, which is just a RESOURCE segment containing all the resources. There are four operations defined on base capabilities.

**CHANGEB (BAS Cap.)** - enables the process to change its base by replacing the current base capability with the new one simultaneously changing the Base Pointer appropriately. No checks are made on resources which are present in the stack or processor registers.

**TRANSD (BAS Cap.,CWD)** - is used to change the execution domain for cross domain calls between mutually suspicious domains, control being transferred to the control segment defined by the CWD (codeword). This operation is elaborated in a later sub-section.

**APPEND (BAS Cap.,Resource Definition)** - adds the resource definition (codeword, capability, extended capability) to the base and leaves the index in a register.

**DELETE (BAS Cap.,INT)** - deletes the resource indexed by the integer and replaces the resource definition by a Null capability.

In addition to the above operations specifically defined on base capabilities, there are two other operations which have a base capability as an operand. The first is the CONNECT operation (Sec. 3.2) which associates a Name capability with a resource in a base. The base organization is illustrated in Fig. 3.14. From it, we observe that the base segment consists of a 'name definition' part and a 'resource
Fig. 3.14 The Base Description
part'. The former contains a set of Name capabilities, while the latter contains the resource definitions. The CONNECT operation introduces the Name capability into the 'name definition' part with the index of the associated resource being entered in the USE field of the NAME capability.

The second operation which has a Base capability as an operand is the START operation defined on a process capability. This operation starts the execution of a process with respect to a base defined by the base capability. Thus, for all the six operations which have a base capability as an operand, the operation is allowed if the bit corresponding to it is set in the USE field of the base capability.

The base capability, like the process capability, is formed as the result of an operation defined on a Base Forming Capability. The new base formed is initialized to contain resource definitions used by the implementation code as well as some standard system resources used by all processes.

6) **Name Capability (NAM)** Name capabilities serve the role of identifiers. These, unlike all other capabilities, are not assigned to processes but exist as part of the control segment. As earlier explained, the control segment contains only the ID field of the Name capability. The DATA instruction is used to form a name capability corresponding to this ID. The operations defined on a name capability are CONNECT and NAMF. The former associates the name capability with a resource in the base (Fig. 3.14). The NAMF operation scans the 'name definition' part for the name capability and then forms a pointer to the resource indicated. As illustrated in Fig. 3.14, the USE field of the name capability indexes the resource associated with it, when the capability
is in the 'name definition' part.

7) **Control Register Access capability (CRA)** The CRA capability is used by processes to access the various hardware control registers. A zero USE field indicates that the capability can only be used in the privileged mode. The ID field contains a code which describes the nature of access. The REG operation applied on a CRA capability produces the following transfers according to the code:

<table>
<thead>
<tr>
<th>CODE</th>
<th>TRANSFER</th>
<th>CODE</th>
<th>TRANSFER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SP</td>
<td>2</td>
<td>register X to SP</td>
</tr>
<tr>
<td>3</td>
<td>MP</td>
<td>4</td>
<td>do. MP</td>
</tr>
<tr>
<td>5</td>
<td>PSVR</td>
<td>6</td>
<td>do. PSVR</td>
</tr>
<tr>
<td>7</td>
<td>CURR</td>
<td>8</td>
<td>do. CURR</td>
</tr>
<tr>
<td>9</td>
<td>CP</td>
<td>10</td>
<td>do. CP</td>
</tr>
<tr>
<td>11</td>
<td>BP</td>
<td>12</td>
<td>do. BP</td>
</tr>
<tr>
<td>13</td>
<td>PI</td>
<td>14</td>
<td>do. PI</td>
</tr>
</tbody>
</table>

8) **Process Forming Capability (PFC)** A PFC authorizes the posessor to spawn another process. It has the operation PROCMAK defined on it. This operation forms an uninitialized Process State Vector and returns a process capability. The PFC contains in its ID field, the ID of the new process capability to be formed. Therefore the PFC is a non reusable resource. It is obtained by calling the RESOURCE MANAGER routine.

9) **Base Forming Capability (BFC)** a BFC authorizes the creation of a new base. The operation CREATE defined on it forms a new base segment containing the resource definitions required by implementation routines as well as some standard system resources. The BFC is also a non reusable resource and contains the ID of the base capability which is returned as a result of the CREATE operation. It is obtained by
calling the RESOURCE MANAGER routine.

10) **Extension Forming Capability (EFC)** An EFC authorizes formation of capability extensions as will be described when the capability extension scheme is considered. It is obtained by a call to the RESOURCE MANAGER routine. The ID field contains the ID of the extended capability to be formed. The operation defined on an EFC is the EXTEND operation which forms an extended capability. In order to accommodate some standard system defined extended capabilities, the USE field of the EFC, if non zero, identifies one of these standard extended capabilities. The EFC is a non reusable resource.

11) **Increment Capability (INC)** When a segment is created, the owner process obtains possession of the INC capability corresponding to the segment. The INC capability contains the segment's MST index in its ID field. It has two operations defined on it, which change the 'options' field attached to the segment's MST entry:

    **DOWNGRADE (INC, INT)** - The options field in the MST index is masked with the integer value.

    **UPGRADE (INC, INT)** - The options field in the MST index is ORed (inclusive) with the integer value.

The revocation features provided for codewords is also available to capabilities. The SEAL operation is now made to include the case of sealing capabilities (CAP) and revokable versions of capabilities (RCAP). The REVOKE operation operates on a revokable version to produce a reduced version of the original capability. Finally, the LOCK operation is valid on a revokable version and sets the lock bit.

Because of the individual nature of the USE field in a capability, the interpretation of the REVOKE operation is dependent on the
type of the capability. In the case of device, process, base and INC capabilities, the USE field indicates the allowed operations on the capability and the REVOKE operation is used in the normal sense. For the CFC and CRA capabilities, the REVOKE operation can be used to set the USE field to zero, allowing the use of the capability in privileged mode only. For all other capabilities, the REVOKE operation is not defined. A significant point is that there is no equivalent operation to EVAL, applicable on capabilities. Thus every time a revokable version of a capability is used, the indirection path to the actual representation is evaluated.

We have considered so far, a set of capabilities which define access to the basic set of system resources. We have also indicated a set of primitive operations which are applicable on capabilities. These operations can be used to define various other actions on resources. Using the generalized notion of a resource, we can treat sets of resources with actions defined on them as an object defined by a capability. This then suggests the need for a scheme for extending capabilities.

e) Extended Capabilities (ECAP and RCAX) The protection structure has evolved on the basis of specifying a set of operations for the descriptors defining base level resources. Protection is ensured by constraining the set of operations valid for each tagged element. A natural extrapolation of this concept is to constrain the set of allowed actions on higher level objects. Since it has been shown that access to system resources can be transformed to the simpler problem of access to storage, all that is needed to support the extrapolation is a scheme
whereby the set of allowed procedures on a structure defined in memory, can be restricted. The capability extension scheme supports this idea.

Each extended capability (ECAp) has a unique identification code specified in its ID field, which can be used to index into a Table of ECAPs. There are two kinds of extended capabilities recognized, differentiated by the 'S' bit. If it is set, it implies the concept of restriction of a set of procedures on a structure. If the 'S' bit is not set, it indicates a structure without the implied restriction of the procedures allowed on it. The former form is called a 'data procedure' and the latter, a 'data structure'. The 'data structure' is provided as an encapsulation of a codeword by a capability. This enables the treatment of a storage structure as a structure rather than as a segment in storage. It may be used as an intermediate step in the design of a 'data procedure'. This is possible by first defining a 'data structure' and once the set of procedures applicable on the 'data structure' have been tested to be correct, converting the data structure to a data procedure. In a data structure, the options field contains the options of the encapsulated codeword. For a data procedure, this field is taken as a 7 bit privileges field (PRIV(7)). A maximum of seven procedures can be defined and the PRIV bit corresponding to a procedure indicates if the procedure can be applied on a structure.

The entry in the Table of ECAPs contains the codeword to the segment defining the extended capability. The segment format for defining an extended capability is shown in Fig. 3.15. The basic idea is that a structure can be defined by a codeword irrespective of whether the structure is represented in a normal or relative codeword format (Figs. 3.10,3.11). The resource segment described indirectly by the
<table>
<thead>
<tr>
<th>Int</th>
<th>Res</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cwd</td>
<td>Codeword (maybe relative) to STRUCTRE</td>
<td></td>
</tr>
<tr>
<td>Cwd</td>
<td>Codeword for Procedure #1</td>
<td></td>
</tr>
<tr>
<td>Cwd</td>
<td>Codeword for Procedure #2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This part does not exist for 'Data Structures'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Cwd</td>
<td>Codeword for Procedure #n</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.15 Definition Segment format for extended capabilities
ECAP has by convention, the codeword describing the structure as its first element. Following it are the codewords to the different procedure segments. If the structure defining codeword is of relative form, the structure is contained in the rest of the segment.

An extended capability may be formed by applying the EXTEND operation on the EFC capability. The EXTEND operation also requires a second operand, namely a codeword describing the definition segment of the extended capability. There are four possibilities for the definition segment as shown in Fig. 3.16:

i) This case corresponds to a data structure, where the structure is defined within a STRUCTURE segment. In this case, assuming the second operand is called X, then codeword X is a STR codeword.

ii) In this case, codeword X describes the structure in normal form. Therefore, no explicit definition segment exists but is implied by CWD X.

iii) Here codeword X defines the STRUCTURE segment shown in Fig. 3.16-iii. The structure segment is formed by constructing a RESOURCE segment containing a STR codeword Y defining the structure, and also the codewords defining the various procedure segments. By applying the GROUP operation on the codeword defining this resource segment, the definition segment of Fig. 3.16 is formed.

iv) Finally, if the codeword defining the structure is in normal form, codeword X simply describes a RESOURCE segment containing this codeword and the codeword to the various procedures.

In all cases, the EXTEND operation indexes into a Table of ECAPs using the ID field of the EFC and updates the entry with codeword X. The operation then forms a ECAP indexing this entry.

The operations defined on extended capabilities or their
ii) 'Data Structure' with normal codeword

iii) 'Data Procedure' with relative codeword

iv) 'Data Procedure' with normal codeword

Fig. 3.16 Different cases of Extended Capability definition
revokable versions are:

EVOKE (ECAP, INT) - This is valid on data procedures (S=1) only. The integer specifies the index of the procedure to be applied on the structure with the ECAP definition. The operation fails if:

- either i) INT is not within the bounds (1 to 6)
- or ii) no procedure is defined for the given index
- or iii) the corresponding PRIV bit is not set.

STRU (ECAP) - This is valid on data structures only (S=0) and forms a pointer to the data area. This is equivalent to an EVAL operation. The need for an extended capability here, is to enable the treatment of the structure as a whole, as a manipulable entity.

CONVERT (CWD or EFC, ECAP) - This converts a data structure type ECAP into a data procedure type. The codeword defines a segment containing the codewords to the various procedures. The case when an EFC is used as an operand will be discussed shortly.

In order to facilitate efficient utilization of the system, a set of standard system defined capabilities are also to be provided. When a process requests an EFC for such an extension, the USE field of the EFC contains the code identifying the standard ECAP. Thus when a ECAP is to be formed by the EXTEND operation, the codewords to the procedures need not be specified. Thus the definition segment just needs to identify the structure and the codewords to the procedures are automatically inserted by the EXTEND operation. Alternatively, a data structure type ECAP can be converted to a data procedure ECAP, with the procedures corresponding to a standard system defined ECAP. This is done by the CONVERT operation on an ECAP and EFC, where the ECAP identifies the data structure and the EFC corresponds to a standard
system defined ECAP.

The EVOKE operation on ECAPs results in the application of a procedure on the data structure defined within the ECAP. Basically, it involves a subroutine linkage to a procedure and the subsequent return. We decided to pass parameters through registers rather than the stack so as to confine the responsibility of executing the MARK and CALL operations to the code implementing the EVOKE operation. Otherwise a process needs to execute the MARK operation and then stack parameters before executing EVOKE. By convention, certain registers are used to pass parameters. The procedure may also use these registers for intermediate computation but is responsible for maintaining the original values. Henceforth, when procedures are specified as part of the ECAP definition, it is assumed that this is done normally and will therefore not be explicitly stated.

The EVOKE operation executes the following steps:

1) Call a system procedure which given the ECAP loads the codeword which comprises the entry in the Table of ECAPs. This codeword describes the definition segment for the ECAP and is loaded into register X1.

2) EVAL the codeword in X1 to form a pointer in X0. The pointer describes the portion containing the structure defining codeword and codewords to the procedures.

3) LOAD the codeword to the corresponding procedure in X1.

4) EVAL the codeword in X1 to form a control pointer in X2.

5) Execute MARK 1 (mark value =1 is used for ECAP linkage).

6) Execute CALL with respect to the control pointer in X2.

7) The procedure now executes and it can access the structure within the ECAP through the pointer in X0, either by using EXPAN
directly (if relative format) or by loading into a register and then using the EVAL operation.

8) The procedure finally returns by executing RETURN 1.

The revocation scheme is also applicable to extended capabilities. The SEAL operation produces the revokable version RGAX, while REVOKE can be used to revoke the PRIV bits of the revokable version of a data procedure type. In the case of the data structure type, the REVOKE is used to revoke access options of the defining codeword which describes the structure and which is encapsulated by the extended capability. The LOCK operation is used to set the lock bit 'L'.

f) Address Pointers Pointers have been introduced in Chapter II and the Appendices. The size of the elements in the segment referred to by the pointer as well as the 'write' access option are both incorporated into the TAG field of the pointer. A pointer defines a set of elements using the 'base limit' format where the FBL field specifies the location of the first element and the LIMIT field contains the index of the last element with respect to the first.

III.5 Domain Interaction

The domain of a process is the execution environment of a process. In terms of the MPN instruction set, we had defined it to be the set of all elements which can be accessed (either as operands or by transfer of control) by the process, by performing a sequence of MPN instructions. Access was then defined as possessing a POINTER to the element or the ability to form a POINTER which is a restricted version of an existing one. In the EPN, the idea was broadened to
include the case of possession of a CODEWORD or a CAPABILITY and finally to also include the case of extended capabilities. The Base capability provides the static definition of a domain by defining a segment containing the resources which can be accessed by a process. Even though a process can access all the resources specified in its base, it cannot increase its domain in the context of the protection structure as no operations have been provided for unrestricted creation of capabilities and codewords. However, we have indicated that a process can obtain a capability or a codeword by calling a system RESOURCE MANAGER routine, which is implemented at a level over the protection structure.

Before discussing the operation of inter domain transfers, it is necessary to establish the inter relationship between the base, process and domain. Essentially, a process is defined for purposes of scheduling as an independent scheduling unit, while the domain represents the dynamic execution environment. The base is of course the static entity. From a different viewpoint though, a process can be considered as a temporary entity, since it represents a physical user active in the computer utility for a certain period of time. All such processes are activated for a finite time and then terminated. A base, on the other hand, is a more permanent record of all the procedure activations with respect to the base. A process during the time it is active, executes with respect to a base and is constrained by the domain implied by the base. We can describe the process base linkage at any instant, as a one to one function from the set of processes to the set of bases. Thus every process has to execute with respect to a unique base.

All the features provided till now have been based on the
implicit assumption that the domain in which the process is executing not only trusts the system features but also any procedures borrowed from other domains. This applies to the case when a procedure is shared by explicitly passing the codeword to it, to the borrowing process as well as the case when implicit sharing occurs by passing an extended capability. The lender of the procedure segment can protect the procedure by disabling the 'read' access ('write' is never allowed on procedure segments), but the borrower process risks having its base fully accessible to the borrowed procedure. Thus we need, as part of the protection structure, features to transfer between bases which are mutually suspicious [Sc 72] of each other. Simplicity considerations with respect to ECAPs restrains us from incorporating these features as part of the capability extension scheme.

There are two operations by which a process can change its base -- CHANGEB and TRANSD, which, as earlier introduced, are defined as operations on a Base capability. The former just results in a change of base without enforcing any other restrictions. The second operation provides for 'mutual suspicion'. It has a second operand, namely the codeword of the procedure segment to which control is to be transferred. The codeword has to satisfy the following restrictions:

1) The 'D' bit in the 'OPTIONS' field is to be set to allow cross domain access.

2) The codeword is to be present in the new base with its 'E' bit set. This is to prevent the calling process from transferring to a procedure defined only in its own base, while changing its base.

* The mechanism for passing the codeword between the two processes is not part of the protection structure, but implemented at a higher level.
3) The Base capability is to exist in the calling base with its PRIV bit, corresponding to the TRANSD operation set.

The base possessing the control segment restricts the codeword in the calling base by keeping the E (execute) and V (allows EVAL) bits disabled in the options field. The calling process is responsible for placing the parameters for the call in order, in the registers and setting the rest of the registers to NULL capabilities. The parameters are to be in the form of codewords or capabilities (Pointers can lead to the compromise of the calling process' base). Integrity of the calling base as well as the called procedure is ensured in the TRANSD operation by using the process stack and its marked link feature.

The TRANSD operation is implemented as a branch to an implementation routine. The codewords for implementation procedures, which are present in all bases, are set for privileged use only (the 'P' bit is not set), thus ensuring their execution in privileged mode. The TRANSD implementation procedure performs the following steps in privileged mode:

1) Stack all the non null registers after first converting all resource descriptors to their revokable versions.

2) Stack the current Base pointer.

3) Execute the MARK 7 instruction (seals the stack and leaves space for a marked link with value 7).

4) Repeat step(1)(the first set of revokable versions will later be used for revocation).

5) Execute CHANGEB (Base Cap.), where the Base capability corresponds to the base in which the borrowed procedure executes.

6) Apply EVAL on the codeword to the procedure segment to form a
control pointer.

7) Execute CALL with respect to the control pointer. This stores the present control pointer (with privilege bit set) in the marked link with value 7. Control is then transferred to the borrowed procedure segment.

The process stack after control has been transferred through the TRANSD operation, is shown in Fig 3.17. The called procedure segment executing in the new domain, has access to only those parameters (passed in their revokable form) which are above the marked link in the stack. The called procedure cannot access the rest of the stack as the Mark pointer (MP) is not allowed to be accessed (allowed only if the corresponding CRA capability is accessible). Furthermore, it cannot access earlier portions of the stack by a RETURN n operation, for 'n' can have a maximum value of only 7 and therefore any RETURN operation will result in transfer of control back to the system procedure implementing the TRANSD. The called procedure can of course use the portion of the stack above the marked link for normal subroutine linkages. When it completes execution, a RETURN operation transfers control back to the implementation procedure which then performs the following steps:

1) Unstack the Base pointer and Base capability.

2) Execute CHANGEB with respect to the above Base capability.

3) Revoke all the revokable versions of resource descriptors which were used as parameters.

4) Transfer back to the calling process.

Since the called procedure is not allowed to maintain pointers in its base, it can only retain revokable versions passed as parameters
Fig. 3.17 The Process stack after a cross domain call
which are revoked as soon as control returns to the calling process. Thus the integrity of the calling process' base is guaranteed. The scheme allows nesting to any level of cross domain calls.

The fact that a RETURN 7 operation is also used to transfer control to the marked link at the bottom of the process stack (normally used to stop process execution) may suggest this to be a disadvantage. However this is not so as the two purposes for a RETURN 7, namely cross domain linkage and process termination, are indistinguishable only when the original process has called another procedure in a domain which it does not trust. Obviously in this situation, the executing procedure should not be allowed to terminate the process which called it, thus implying that the RETURN 7 is to be interpreted as a return to the calling domain.

III.6 Multiprocessing

Multiprocessing, although not specifically stated, has been one of the major motivations behind the protection structure. The need for a scheme of dynamic revocation is implicit in a multiprocessing environment. More important is the fact that the protection structure sets up the basis for a scheme of process cooperation in higher layers. This process cooperation assumes a multiprocessing environment. The situation of having more than one process executing at the same time, requires some hardware support.

As in standard architectures, a TEST AND SET (TST) instruction is incorporated into the hardware. It operates on an INTEGER element and sets the overflow indicator if the overflow byte of the INT element is nonzero, otherwise it increments the overflow byte
without setting the indicator. This is done as an indivisible operation. Another hardware instruction REDINC is provided as an indivisible operation which reads and increments the corresponding INT element.

The only case of processes accessing segments without using the EVAL operation is when system tables are accessed. These can be Tables of capabilities, codewords etc. The access can be by forming a pointer to the table and accessing an indexed location. Alternatively the next free entry in the table may be required. In this case, provision should be made to prevent two processes from simultaneously accessing the table and obtaining the same free entry. Assuming an INTEGER element indexes the next free entry, the REDINC (read and increment) operation ensures that each process can obtain a unique 'free' entry.

Schemes for interprocessor communication are rejected on the basis that violation of protection can occur. Hence processes can communicate only at a software level, which is implemented over the protection structure and therefore does not compromise protection.

III.7 Implementation of the Protection Structure

The implementation of the protection structure is indicated (not rigourously specified) with an emphasis on the three features of dynamic revocation, mutual exclusion and a layered structure.

a) Dynamic Revocation The suggested revocation scheme provides the framework for dynamic revocation but does not ensure immediate revocation. In the case of capabilities and extended capabilities, access to the actual representation of the object is determined each time the capability is used. Thus the revocation scheme provides for
immediate revocation with respect to capabilities. However, in the case of codewords, a pointer is evaluated from the codeword. Since the revocation scheme does not affect pointers, immediate revocation is not ensured. Various solutions to the problem exists:

1) Every time a pointer is used in an operation, the corresponding codeword in the register extension is EVALed.

2) Whenever revocation occurs, the PSVs of all processes are scanned and the pointers corresponding to codewords which have been revoked updated to the new access specifications.

3) Provide backpointers in the revocation scheme and maintain a check over the distribution of codewords.

4) Whenever a pointer is unstacked, perform an EVAL operation on the associated codeword.

The first solution is the ideal case but not feasible in practical situations as it will increase the instruction execution time by several orders of magnitude. It also defeats the very purpose for which pointers are introduced, namely to provide access to storage. The second solution increases the overhead of revocation but is feasible on a limited scale. The third solution involves cumbersome back-pointer structures which unnecessarily increase the complexity. The fourth solution is only a partial one as no check is made to ensure the validity of pointers in the registers.

First we point out that immediate revocation is not crucial to the system. This is because a process is only a temporary entity. Therefore, the record of activations of a user are maintained in the base. Since a base cannot contain pointers, everytime a process is activated with respect to a base, the domain is specified according
to the valid access at that time. Thus although we recognize the need for dynamic revocation, we reject the idea of immediate revocation.

The solution is a combination of the second and fourth solutions. The fourth one provides a partial solution. Processes maintain a knowledge of the distribution of resources at a process communication level, if necessary. Thus when revocation of a codeword occurs, the process can initiate an operation which revalidates all the pointers present in the registers of the process' PSV. This operation is the VALIDATE operation, which has as its operands a process capability and a codeword. The operation scans the register extensions in the PSV of the process and revalidates the access for pointers corresponding to the codeword. Again it is emphasized that immediate revocation is effective for all other resources, which are described by capabilities.

b) Mutual Exclusion is the provision of unique access to an object. We confine it to the problem of providing unique access to a segment defined by a codeword. The EVAL operation as it has evolved from its MPN definition performs the following steps:

1) Check whether the operand is a normal codeword or the revokable version of a codeword.

2) Follow the indirection path through the Table of ROCOD elements, in the latter case, to obtain the codeword defining the segment referred to.

3) Use the codeword to index into the Master Segment Table and access the corresponding entry.

4) Check whether the blocking bit is set in the MST entry. If so, the check is repeated, otherwise the bit is set.
5) Form a pointer to the segment indicated using the starting location given in the MST entry.

The above scheme provides for mutual exclusion on a segment basis. We reject this contention and state that the protection structure should only provide features so that mutual exclusion on a segment basis can, if needed by the higher layers, be ensured rather than enforcing it in all cases. The hardware memory constraints anyway provide for mutual exclusion on a byte basis between concurrent processors. The solution which will be suggested is dictated by the following observations:

1) The EVAL operation, as indicated at this stage, does not differentiate between write access where mutual exclusion is necessary and read or execute access where it is not. Thus the scheme is inefficient, since in all cases, only one process is allowed to access the segment.

2) In normal usage, a sizable fraction of the segments are 'read only' segments or non writable control segments.

3) Even in the case of segments where 'write' access is allowed, it is not necessary that mutual exclusion has to be guaranteed on a segment basis.

4) The EVAL operation is a basic operation and is to be implemented with minimum overhead.

Mutual exclusion is provided for by denoting an OPTIONS field in the MST entry (Fig. 3.18). The use of the 'options' field relies on a scheme where the owner (creator) of the segment decides the options which the segment is going to have. If the segment is a 'read only' segment, then it has only its 'R' bit set and the EVAL operation does not check for the blocking condition at any time. If
The INTEGER element fields are -

Op : Options - 4 bits
   R - Read access allowed
   W - Write access allowed
   E - Execute access allowed
   M - Mutual Exclusion to be implemented on the segment

Ovf : Overflow byte which if set indicates a blocking condition on the segment.

Fbl : First byte location of the segment

The blocking condition can be tested by the TST (test and set) instr.

Fig. 3.18 The Master Segment Table entry
the 'E' bit is set then execute access is to be granted (a control pointer can be formed) but again no check for a blocking condition occurs. When the 'W' bit is set, write access is granted but without the check. The implication is that the byte basis mutual exclusion is sufficient. Only if the 'M' bit is set, does the EVAL operation have to check for a blocking condition (indicated by the overflow byte of the INT element in the MST entry). If the condition exists, then the process capability in the next element of the entry (Fig. 3.18), is compared with that of the resident process. If they match, then access is granted. The EVAL operation is implemented as shown in Fig. 3.19. The essential idea is that the decision for mutual exclusion is now made by the higher levels and not by the protection structure itself.

c) Layered Structure The protection structure has been specified as a set of allowed operations on various classes of operands. It forms the intervening layer between the hardware architecture and the software layers of the operating system. However, the protection structure can itself be implemented as a layered structure. This assumes some hardware provisions to facilitate the implementation.

We define four layers of implementation of the EPN system which includes the hardware machine as well. The bottom layer (level 0) is the hardware machine and comprises of the set of MPN instructions which are directly implemented by hardware. The rest of the three layers constitute the protection structure. The next layer (level 1) comprises of a set of operations which are frequently used but require a certain degree of flexibility depending on the implementation. These operations are then implemented as microsystem functions, which use a
Fig. 3.19a The EVAL operation
Fig. 3.19b The EVAL operation (continued)

A (continued from Fig. 3.19a)

- Is 'M'=1 in Options of MST entry
  - No
  - Is $X_2$ of Typ-STR
    - No
      - Using Fbl of MST entry, form pointer to the segment
      - Use 'Limit' value from first element of segment
      - Reduce Ptr. TAG if options in MST entry does not allow access
    - Yes
      - Using Fbl of MST entry, form Res. Ptr of 'limit'=0
  - Yes
    - Is P1reg, equal to Proc. Cap. in entry
      - No
        - Transfer to a Process Block routine
      - Yes
        - Stop
small control memory for a microprogram and a shift register stack to contain the return link. These can be considered as variable hardware functions. The next layer (level 2) is implemented by making use of a macro assembler feature wherein the code to implement these operations is inserted into the user program, but using a privileged mode of operation. Finally, the top layer (level 3) is implemented as normal calls to the implementation routines.

The protection structure operations make use of the same instruction format as the MPN instructions. Whenever a protection structure operation is specified, the macro assembler inserts the appropriate code for level 2 and level 3 operations. In the former case it is the actual implementation code, in the latter, a call to the concerned implementation routine. The hardware function is to recognize the occurrence of a protection structure operation. If a level 1 operation is recognized, the transfer to a microprogram subroutine occurs. One of the functions available as a microprogram subroutine is a SETPRIV operation, which sets the privilege bit of the instruction pointer. Therefore, if a level 2 operation is recognized, the hardware first executes the SETPRIV operation automatically and the rest of the macro assembler code is then executed in privileged mode. SETPRIV is also executed automatically if a level 3 operation is detected, control then being transferred to the implementation routine. Once in privileged mode, the privilege bit can always be reset using the appropriate CRA capability. For simplicity, we define a REMPRIV hardware instruction which transfers back to the normal mode.

In a layered structure, the basic problem is how to divide the operations into layers. Level 1 requires a set of operations used
often by the higher layers but also simple enough to be implemented as microprogram subroutines. Level 3 operations require elaborate code which is inefficient to be inserted by a macro assembler. The remaining operations are implemented in level 2.

Apart from the different schemes of implementation for the various layers, there is a more important relation between the layers. This relation is that the implementation of a level uses the operations of the lower levels. Thus the implementation assumes that an operation can be defined in terms of lower level operations. Also, as will be shown in the next chapter, some protection structure operations need to be used exclusively by a process. This implies that while a process is executing the implementation code for this operation, no other process may simultaneously be executing the same code. Such operations are called monitor operations. The reason for classifying particular operations as monitor operations will be given later, but, at this stage, we just indicate the division of the protection structure operations into the various classifications:

Level 0: All MPN instructions (except RETURN), REDINC,TST, SETPRIV and REMPRIV.
Level 1: RETURN, EVAL and RELEASE.
Level 2: EXPAN, SEAL, REVOKE, LOCK, COMMUNICATE, CHANGEB, CONNECT, NAMF, DOWNGRADE, UPGRADE, STRUC.
Level 3: GROUP, IN, OUT, TRANSD, EVOKE, EXTEND, CONVERT.
Monitor: INITIALIZE, START, STOP, CONTINUE, TERMINATE, VALIDATE, APPEND, DELETE, PROCMAK, CREATE.
Chapter IV

The Nucleus of the Operating System

IV.1 Introduction

This chapter is not meant to serve as the detailed specification of a complete multiprocessor operating system. Apart from the fact that such a task involves several man years of work, it is not adaptable as a research topic. The emphasis in this chapter is, on the other hand, directed towards a consideration of the basic design principles involving such systems. The motivation is to show that the criteria required to be satisfied on the basis of these design principles are adequately met by the protection structure which has been evolved. The design accentuates the features of a multiprocessing environment by directing the effort to achieve maximum parallelism and efficiency, at the same time meeting the constraints of synchronization required by such an environment.

IV.2 The Design philosophy of the Operating System

Although the operating system does not correspond to any particular existing system, the design philosophy intrinsically reflects the 'kernel' concept of the HYDRA [Wu 74] and the 'nucleus' concept of the RC4000 [Ha 70, Ha 73]. The basic idea in both systems is the specification of a set of facilities which are easily adaptable to any operating system, at the same time, the reliability of which is assured. Given this collection of mechanisms, an arbitrary set of operating system facilities and policies can be conveniently, flexibly, efficiently and reliably constructed. The basic set of facilities or mechanisms is called the 'kernel' or 'nucleus' of the operating system.
This chapter formulates the design of the 'nucleus' of a multiprocessing system. We first need to establish the relation between the protection structure (Extended Pointer System) specified in the last chapter and the nucleus to be designed in this chapter. It is to be emphasized that the nucleus includes the protection structure. But while the protection structure transforms the architecture into a set of mechanisms, the nucleus utilizes these mechanisms to simulate an environment in which the specific policies governing an operating system can be implemented. The nucleus also includes the resource handling, input/output, processor allocation features required of any operating system.

The distinction, between the mechanisms required by an operating system and the policies which govern a specific operating system and which use these mechanisms, must be understood. For instance, the ability to use a base to specify the execution environment of a process, is a mechanism; while the actual definition of the environment in terms of the resources the base contains, is a matter of policy in a specific operating system. Taking a different viewpoint, a facility provided for allocating processors to processes on a short term basis, a facility to temporarily stop processes and another to restart these stopped processes, all comprise a set of mechanisms. The use of these, to determine a medium term scheduling strategy for processes is termed a policy.

The T.H.E. multiprogramming system [Di 65, Di 68] first recognized the idea of designing an operating system as a set of concurrently executing processes. These processes are characterized as sequential processes in that no assumptions were made about their speed
ratio. Consequently Dijkstra concluded that:

1) Synchronization between processes is necessary as no assumption is made about speed ratios.

2) On the other hand, synchronization is possible as temporary delay of processes is allowed.

The nucleus of the RG4000 system [Ha 73] was directed towards the control of an environment consisting of cooperating concurrent processes. In this context, the environment was structured so as to allow the uniform treatment of program execution and input/output as cooperating processes executing concurrently. The policy defining operating system is treated as any other process, a hierarchical ordering of the processes ensuring that various operating systems can be defined to control specific user processes. The design of the 'nucleus' for our system reflects this idea.

Let us first establish a set of design guidelines which identify the essential features required of the nucleus:

1) The nucleus needs to provide a scheme for process cooperation which incorporates a synchronized mechanism of message communication between processes.

2) The processes controlling input/output need to be executed concurrently with other processes.

3) The I/O processes also need to be able to cooperate with all the other processes, with the additional facility that a process can reserve the I/O process to cooperate with it exclusively for a temporary period. The I/O processes need to be synchronized with the physical devices also.

4) The nucleus needs to provide for the orderly creation of
processes and their subsequent control by the 'creating' process through well defined mechanisms. This provides the hierarchical ordering of processes required.

5) The nucleus should also provide the features for short term process scheduling. It also needs to implement short term resource management for storage and resources defined by other capabilities. This includes the maintenance of bases and processes.

Although the RC4000 supports many of the above features, it has some intrinsic disadvantages with respect to the basic design and violates some of the guidelines enumerated above. The rationale in the RC4000 was to implement the 'nucleus' within a 'monitor'. The essential requirement of the monitor was that it be exclusively accessed by any process. Thus concurrent sharing of the monitor features was not possible. The monitor was defined as a collection of procedures which maintained complete control over input/output, store protection, interrupt response and short term scheduling, as also providing a set of mechanisms for process cooperation, process creation and process control. The grouping of all these facilities inside one non-shareable unit resulted in an inherent loss of flexibility. More specifically, the disadvantages in such a scheme are:

1) Although, the monitor simulates an environment where normal processes cooperate with I/O processes; with respect to short term scheduling, the uniform treatment is not maintained. This is because, all I/O processes are implemented within the monitor and since the monitor is not interruptible, I/O processes are executed to their completion with no pre-emption.

2) Mutual exclusion is provided by the above crude method of
interrupt inhibition within the monitor with respect to monitor controlled data structures. Mutual exclusion for arbitrary data structures is by creating a process to control the structure and then using message communication to this process to access the data structure.

3) Process communication has many restrictions. Since communication mechanisms are implemented as monitor procedures, all process interactions need to exclude one another in time. Also flexibility constraints exist in message communication brought about by resource restriction (limited number of the buffers used for communication), message content restriction (fixed message length) and inefficient implementation (as the message needs to be copied to or from a buffer).

4) There is also a problem in the context of medium term scheduling with respect to sharing a data area (especially in the case of an I/O process transferring to a block) because of storage relocation.

It must be emphasized that all these disadvantages were realized by the designers of the RC4000 system and have been indicated here to stress the salient aspects of our 'nucleus' towards resolving these difficulties. The essential point is that although a 'monitor' is required to implement certain features of the 'nucleus', these should be as few as possible, with the rest of the features being supported outside the monitor. Thus, the monitor is no longer a bottleneck in the features provided by the nucleus. It is towards this end that our design efforts are directed.

The delineation of some of the features of the nucleus from the centralized monitor suggests the partitioning of the features on the basis of implementation:

1) Those implemented as monitor procedures and used by executing
monitor primitives' corresponding to these procedures.

2) Those implemented on the basis of defining an EPN operation on a tagged element following the rationale of the protection structure. The protection structure operations as shown earlier, could be implemented as macro expansions or a call to an implementation procedure (as this is not within the monitor, it is simultaneously shareable).

3) Finally we can also implement certain features using the capability extension scheme on the basis of defining procedures on data structure.

Although, we have shown the above classification, we will introduce the features of the nucleus based on a functional classification:

a) Those supporting process cooperation.

b) Those supporting input/output handling.

c) Those supporting process creation and control.

d) Those supporting short term resource management.

IV.3 Process Cooperation

As pointed out earlier, the crucial aspect of process cooperation between sequential processes is the synchronization of processes. By synchronization, we imply a constraint on the ordering of execution points of the processes, with respect to time. This constraint may be in the form of a priority of execution points or a mutual exclusion of execution points in time. The latter situation can of course occur only in the case of a multiprocessing environment.

The problem of mutual exclusion has been reviewed extensively in literature with Dijkstra [Di 65] introducing the concept of a
'critical section' which is implemented as a software construct. Our concern with mutual exclusion has been already introduced and we have provided, through the EVAL operation, a mechanism for implementing it on a segment basis. Although the solution to the mutual exclusion problem serves as the basis for all synchronization problems, we need to recognize and provide for other synchronizing mechanisms as well.

The three synchronizing mechanisms we will suggest implementations for are 'semaphores', 'message buffers' and 'event queues' [Ha 73]. Semaphores were originally introduced [Di 65] to provide a solution to the mutual exclusion problem, but serve as ideal synchronization mechanisms when an exchange of timing signals is required. The semaphore is defined as a semaphore variable 'v' with two operations applicable on it - SIGNAL(v) and WAIT(v) which satisfy the following rules:

Let s(v) be the number of signals sent
r(v) be the number of signals received
c(v) be the number of initial signals called the semaphore initialization value

At any instant, the semaphore is characterized by the above three components. The synchronizing rules defined are:

WAIT(v)- If the operation WAIT is executed when r(v)<s(v)+c(v), then r(v) is increased by one and the process continues. But if r(v)=s(v)+c(v), then the executing process is delayed in a process queue associated with the semaphore.

SIGNAL(v)- The operation SIGNAL increases s(v) by one. Now, if one or more processes are waiting in the associated process queue, then one is selected and enabled to continue and r(v) is increased by one.
The given synchronizing rules ensure that at all times $0 \leq r(v) + s(v) + c(v)$.

Message buffers are a generalization of semaphores in that instead of timing signals, actual messages are interchanged between processes. A message buffer is defined as a buffer variable 'B' constrained to a maximum capacity 'Max' of the messages it can accommodate. In semaphores, the capacity limitation is ignored on the basis that the normal bounds of integer values are satisfactory for all situations. Two operations are possible on buffers - SEND (M,B) and RECEIVE (M,B) where B is a message buffer and M is the message. The synchronizing rules are:

**SEND (M,B)**- If the buffer B has reached its Max capacity, then the executing process is delayed in a 'send' process queue associated with the buffer. At the same time, there is also a 'receive' process queue associated with the buffer. If the buffer has not reached its Max capacity, the message is inserted into the buffer and if there is a process waiting in the receive queue, it is enabled to continue.

**RECEIVE (M,B)**- If the buffer B is empty, then the executing process is delayed in the receive queue. If it is not empty, the message is removed from the buffer and if a process is delayed in the send queue, it is then enabled to continue.

Finally event variables are defined to be associated with a shared segment say 'v'. The event variable is a variable 'e' with two operations defined on it, AWAIT (e) and CAUSE (e). The operations are restricted to be executed when the executing process has exclusive access to the segment 'v' (by the EVAL operation) and defined by the following rules:

**AWAIT (e)**- The executing process releases the segment 'v' and is
delayed in a process queue associated with the event variable.

**CAUSE (e)**- All the processes delayed in the process queue are enabled to continue.

The mechanisms described are used to satisfy different synchronization requirements. For instance, a user process may initiate an I/O process and then later wait for its completion. A semaphore can be used for this, so that the I/O process can 'signal' completion and the user process can 'wait' for completion. Again a user process may require another user process to execute some operations. These operations can be specified to the other process by using a message buffer between the two processes. Finally, suppose an operating system wants to reset a resource after all processes in a set have exclusively used it once. This can be effected by each process using the resource and then incrementing a counter which denotes the number of processes which have used it. If this counter is less than the total number in the set, it executes an 'await' on an event variable, but if it is equal, then the process informs the operating system which can then proceed to reset the resource and execute a 'cause' on the event variable to enable all the blocked processes to continue.

All the three synchronization mechanisms introduced have been defined as a variable and a set of operations on this variable. Synchronization is effected by defining synchronizing rules which govern the operations. This then suggests an implementation in the form of extended capabilities as this supports the notion of defining an entity as a data structure and a set of procedures applicable on the data structure.

We first stress on the fact that, by using the protection
structure mechanisms earlier defined, we can provide for flexibility in the definition of extended capabilities (ECAPs). Recapitulating, in order to form the ECAP, we first need to obtain an EFC capability corresponding to the ECAP. If the ECAP to be formed is a standard system defined ECAP, then the procedures applicable on the data structure are already determined. The features of the nucleus to be implemented as ECAPs are all going to be defined as standard system ones. Thus a process needing to avail of a feature, needs to obtain an EFC corresponding to it; define its data structure as a segment and then apply the EXTEND operation. Referring to Fig. 4.1, if A is defined to be a 'mutually exclusive' segment (setting the M bit in the MST entry), a process can then by the EVAL operation secure exclusive access to the segment. Thus effectively, if an ECAP is formed, then each process can exclusively use the ECAP*. Of course, A need not be defined as 'mutually exclusive', which implies that processes may simultaneously share the ECAP. In fact, a particular operation on the structure could be restricted to be applied exclusively, by defining the control segment corresponding to the procedure as mutually exclusive. Thus flexibility can be incorporated into the ECAP at the segment definition stage.

The data structure in segment A (Fig. 4.1) can alternatively be collapsed by the GROUP operation (Fig. 4.2), taking care that A is a Resource segment and does not contain any revokable codewords. As earlier defined, the GROUP operation expands segment A to include copies of the segments at the next lower level in the structure. The flexibility indicated earlier is also possible in this case, but with the restriction that operations now apply only on copies of the second

*The EVOKE operation does the EVAL operation on the codeword in the Table of ECAPs entry, thus obtaining exclusive access.
EXTEND \((EFC, X_a)\) replaces the codeword \(X\) in register \(X_a\) with an ECAP as follows:

\[ X_a \]
\[ \text{ECAP} \]

Table of ECAPs

\[ \begin{array}{|c|c|}
\hline
\text{Int} & \text{Res} \ n \\
\hline
\text{Cwd} & X \\
\hline
\text{Cwd} & \text{to Proc.} \ #1 \\
\hline
\text{Cwd} & \text{to Proc.} \ #n \\
\hline
\end{array} \]

--- --- Implied link thru the ID

--- --- --- Implied link thru the MST

(Note: The codewords to the procedures have been inserted in the new segment formed.)

Fig. 4.1 ECAP formation with a normal codeword
EXTEND \((EFC, X_a)\) replaces the STR codeword in \(X_a\) with an ECAP as follows:

\[
\begin{align*}
X_a & \quad \text{Ecap} \\
\text{CwdStr} & \quad X'
\end{align*}
\]

\begin{itemize}
  \item \text{Table of ECAPs}
  \item \text{CwdStr}
  \item \text{Int} \quad \text{Str}
  \item \text{CwdRes} \quad X \text{ (rel.)}
  \item \text{Int} \quad \text{Res} \quad n
  \item \text{CwdRes} \quad X \text{ (rel.)}
  \item \text{Cwd} \quad \text{to Proc.} \#1
  \item \text{Cwd} \quad \text{to Proc.} \#n
  \item \text{First two levels of the data structure}
\end{itemize}

\begin{itemize}
  \item \text{Implied link thru the ID}
  \item \text{Implied link thru the MST}
\end{itemize}

\text{(NOTE: The codewords to the procedures have been inserted at the same level as codeword \(X\).)}
level structure. In either case, if a standard system defined ECAP is being formed, then the codewords to the procedures are inserted as part of the EXTEND operation as shown. Alternatively, if a new type of ECAP is being defined, the executing process is responsible for forming the procedure segments and inserting codewords to these when defining segment A. Conceivably, one could have commonly available procedures, which given the parameters characterizing a 'feature' described by an ECAP takes the responsibility of forming the segment A and after applying the EXTEND operation returns a newly formed ECAP to the calling process.

At this stage, we need to clarify some points in the EVAL operation in the context of the process scheduling environment. The responsibility for the short term processor allocation is given to the monitor. As stated, the EVAL operation, in the case of 'mutually exclusive' segments ensures unique access to a process. Hence, if a process has secured access to such a segment (by forming a pointer through the EVAL operation and setting the blocking bit), then another process executing the EVAL on the codeword to the segment must be blocked. In Chapter III, we had described this effect by specifying that a process block routine is called. Actually, the implementation code of the EVAL function executes a monitor primitive VACATE (WELO, codeword). The effect is that the executing process is blocked and the process capability stored in a WELO (waiting on block lockout) queue of processes along with the codeword of the segment on which the block occurred. Now, when a process releases a segment by executing RELEASE (refer Sec. III.2), the monitor primitive SCAN_WELO (CWD) is automatically executed resulting in a scan of the WELO queue for all the processes blocked on the
segment and reenabling these by inserting them in a queue of processes waiting for a processor, namely the WGPU (waiting for CPU) queue. We stress that the monitor is used only to implement such features, where it is essential that only one process use it at a time. Hence the EVAL and RELEASE functions correctly implement mutual exclusion by using monitor features which handle the scheduling queue, although they themselves can be simultaneously used by many processes.

It is crucial to the implementation, that the mutual exclusion required by the monitor primitives be enforced by hardware arbiters connected to the processors. Obviously, EVAL cannot be used to implement this mutual exclusion as the EVAL operation itself involves executing a monitor primitive. We have introduced two new monitor primitives, SCAN_WBLO (CWD) and VACATE (status, resource identifier), the latter being used to pre-empt the executing process with the specified status due to a lockout caused by the resource indicated. We will use this primitive again in later implementations.

In order to implement the synchronization mechanisms, we also have to implement some 'structures' which these mechanisms will use. All of them rely on the concept of capability extension.

1) Queue Structure a queue structure essentially involves a queue data structure, implemented as a circular array and with three operations defined on it-- INSERTS, REMOVES and QSIZE. The data structure is characterized by a maximum capacity 'LENGTH' and the 'TYPE' of the data structure segment. Recalling that arguments are passed through registers, we can assume, by convention, that the operations have the following effect:
Codeword in the Table of ECAPs entry

Subsegment A: contains the codewords to the 'data structure' as well as those to the procedures defined.

Subsegment B: contains the first level of the 'data structure', namely the codeword to an ARRAY, an INT element specifying the index of the HEAD of the queue and another INT element specifying the TAIL of the queue.

Subsegment C: contains the ARRAY where the queue elements are stored. The INSERTS, REMOVES, QSIZE procedures are standard procedures using a circular array structure to implement the queue and are used to insert an element, remove an element and interrogate the number of elements with respect to the queue.

**Fig. 4.3 Segment format for a Queue Structure ECAP**
INSERTS: Takes the element in register $X_a$ and inserts it in the tail of the queue.

REMOVES: Removes the element from the head of the queue and puts it in register $X_a$.

QSIZE: Returns an INTEGER element in $X_a$, whose value is 0 if the queue is empty, 2 if it is full and 1 if it is neither.

The element in $X_a$ for INSERTS and REMOVES is constrained by the TYPE classification of the queue structure. Also no checks are made for insertion and removal, so that insertion in a full queue may overwrite an element and removing from an empty queue may produce a garbage value. We will use the notation $Q(\text{LENGTH,TYPE})$ to denote a queue structure (Fig. 4.3).

2) Queue D Structure This is shown (Fig. 4.4) to illustrate the idea of using existing ECAPs to define further extensions. A queue D structure is basically a Q structure which inserts and removes two elements at a time from the registers $X_a$, $X_b$. It is also characterized by a TYPE and LENGTH where LENGTH represents the capacity in units of pairs of elements. The three operations defined are INSERTD, REMOVED and QSIZE and we will use the notation $QD(\text{LENGTH,TYPE})$.

3) Semaphores We have defined a semaphore as a variable with two operations defined on it. The implementation is illustrated in Fig. 4.5. Essentially, the synchronizing rules are established by using an INT (integer) element as a counter. The semaphore is characterized by an initial value to which this counter is set when the ECAP is created. For purposes of clarity, we will specify the procedures using a 'pidgin
codeword in the Table
of ECAPs entry

<table>
<thead>
<tr>
<th>Int</th>
<th>Res</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Q (2L, RES)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cwd Cnt</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'INSERTD'</td>
</tr>
<tr>
<td></td>
<td>'REMOVED'</td>
</tr>
<tr>
<td></td>
<td>'QSIZE'</td>
</tr>
</tbody>
</table>

QD(L, RES) ⇒

Operations:

\[
\begin{align*}
1 & \ - \ \text{INSERTD} \\
2 & \ - \ \text{REMOVED} \\
3 & \ - \ \text{QSIZE}
\end{align*}
\]

It is assumed that the EVOKE operation first results in a pointer to the 'data structure' representative element, being loaded in XO. In this case, a Res. Pointer to the ECAP is present in XO, when a call to the corresponding procedure is executed. The procedures are:

**INSERTD:**

```
LOAD XI, XO  
EVOKE 1, XI  
COPY Xa, Xb  
EVOKE 1, XI
```

(demarcate a comment field)

/* XI now contains the ECAP to Q(2L, RES) */

/* the element in Xa is 'INSERTS' into the Q struct. defined by the ECAP in XI */

/* Xb is copied into Xa */

/* 'INSERTS' is repeated */

**REMOVED:**

```
LOAD XI, XO  
EVOKE 2, XI  
COPY Xb, Xa  
EVOKE 2, XI
```

/* first element is 'REMOVES' from the Q */

/* Xa is copied into Xb */

/* 'REMOVES' is repeated */

**QSIZE:**

```
LOAD XI, XO  
EVOKE 3, XI
```

/* 'QSIZE' of the single Q structure */

Fig. 4.4 Definition of a Queue D Structure
codeword in the Table of ECAPs entry

SEM (n) ⇒

Operations:

- 1 - SIGNAL
- 2 - WAIT

Assume,
- the Q struct. ECAP is referred to as QEX
- the Counter (INT element) is referred to as COUNTER
- the Semaphore ECAP itself is referred to as EXC

**SIGNAL:**

```plaintext
eVOKE (QEX, 3) /* X_a now indicates if the Q is empty */
if X_a ≠ 0 then /* if the Q is not empty then */
    begin
        EVOKE (QEX, 2) /* removes a Proc. Cap from the Q into X_a */
        RESUME (X_a) /* monitor primitive which causes the resumption of the process corresponding to the process capability in X_a */
    end
else COUNTER = COUNTER + 1
```

**WAIT:**

```plaintext
if COUNTER > 0 then COUNTER = COUNTER - 1
else begin
    X_a = ϕ(PI) /* the process cap. of the executing process is loaded into X_a using the REG instr. */
    EVOKE (QEX, 1) /* it is then inserted into the Q */
    RELEASE (EXC) /* release the defn. seg. of the ECAP */
    VACATE (WSEM, EXC) /* the process is blocked in the WSEM status due to the ECAP » EXC */
end
```

Fig. 4.5 The Semaphore Implementation
ALGOL' language which is self explanatory. It is to be emphasized that although codewords, variables and ECAPs are referred to by identifiers, the actual procedure code refers to these using pointers. It is seen that the data structure uses a Q structure ECAP to maintain a queue of processes (represented by their process capabilities), blocked on the semaphore. The Q structure has a maximum capacity or length of N where N is the maximum number of processes allowed to exist in the system, thus ensuring that it never gets full.

The SIGNAL procedure makes use of another monitor procedure, RESUME, which re-activates the process represented by the process capability in the operand register. The re-activation is done by inserting the process capability in the WCPU queue and changing its status to WCPU. The WAIT procedure uses the monitor primitive VACATE earlier introduced, which pre-empts the executing process and indicates its status as WSEM (wait on semaphore lockout), caused by the semaphore whose ECAP is indicated. The process capability of the executing process indicated by the PI hardware register, is first inserted into the queue associated with the semaphore. Mutual exclusion of operations on the semaphore is guaranteed by making the definition segment 'mutually exclusive'. We use the notation SEM(n) to denote semaphore with initial value 'n'.

4) Message Buffers are characterized by a MAX capacity and have the operations SEND, RECEIVE defined. The implementation (Fig. 4.6) essentially involves the use of two semaphore ECAPs, one to prevent a process from sending a message into a full buffer and the other to prevent receiving from an empty buffer. The messages are contained in a queue
Codeword in the Table of ECAPs entry

<table>
<thead>
<tr>
<th>Int</th>
<th>Str</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cwd</td>
<td>Res</td>
<td>(relative)</td>
</tr>
<tr>
<td>Int</td>
<td>Res</td>
<td>2</td>
</tr>
<tr>
<td>Cwd</td>
<td>Res</td>
<td>(relative)</td>
</tr>
<tr>
<td>Cwd</td>
<td>Cnt</td>
<td>Procedure 'SEND'</td>
</tr>
<tr>
<td>Cwd</td>
<td>Cnt</td>
<td>Procedure 'RECEIVE'</td>
</tr>
<tr>
<td>Int</td>
<td>Res</td>
<td>2</td>
</tr>
<tr>
<td>Cap</td>
<td>QD(MAX,RES)</td>
<td></td>
</tr>
<tr>
<td>Cap</td>
<td>SEM(0)</td>
<td></td>
</tr>
<tr>
<td>Cap</td>
<td>SEM(MAX)</td>
<td></td>
</tr>
</tbody>
</table>

BUFF (MAX) ⇒

Q structure to contain messages

Operations:

1 - SEND

2 - RECEIVE

Semaphore: SEMF

Semaphore: SEME

Assume,

- the queue D structure used to contain messages is called MSGQ.
- the semaphore to prevent a message being entered when the buffer is full is referred to as SEME.
- the semaphore to prevent removing a message, when the buffer is empty, is referred to as SEMF.

SEND:

\[ X_b = \#(PI) \]

/* \[ X_b \] now contains the Proc. Cap. of the executing process */

EVOKE (SEME, 2) /* 'WAIT' until the buffer is not full */

EVOKE (MSGQ, 1) /* 'INSERTD' the CWD (in \[ X_a \]) describing the message along with the Proc. Cap. in \[ X_a \] */

EVOKE (SEMF, 1) /* 'SIGNAL' the semaphore SEMF */

RECEIVE:

EVOKE (SEMF, 2) /* 'WAIT' until the buffer is not empty */

EVOKE (MSGQ, 2) /* \[ X_a \] now contains the CWD describing the message and \[ X_b \], the sending process' cap */

EVOKE (SEME, 1) /* 'SIGNAL' the semaphore SEME */

Fig. 4.6 Message Buffer Implementation
D structure and the identity of the sending process is automatically attached to each message. By convention, the message is described by a codeword in register \( X_a \), with the described segment (it cannot be of class MIXED as pointers are not allowed in messages) containing the contents of the message. The process blocking is implemented by the semaphores and the message buffer implementation does not need to concern itself with this aspect. The semaphore queues associated with the semaphores SEME and SEMF correspond to the 'send' and 'receive' queues respectively. The notation to denote a message buffer of capacity \( \text{MAX} \) is \( \text{BUFF}(\text{MAX}) \).

5) Event Variables
The event variable is a queue associated with a shared segment and has two operations, \text{AWAIT} and \text{CAUSE} implemented as in Fig. 4.7. The \text{AWAIT} procedure effects the insertion of the executing process' capability in an event queue associated with the event variable and the pre-emption of the executing process in the status \( \text{WEVN} \) (waiting on event lockout) caused by the event variable \( \text{ECAP} \).

The \text{CAUSE} procedure re-enables all the processes in the event queue by repeatedly executing the monitor primitive \text{RESUME}.

So far we have introduced three monitor primitives—\text{VACATE}, \text{RESUME} and \text{SCAN_WEO}, all used in short term scheduling and available to the implementation code of various operations. Since all implementation code is executed in privileged mode, these primitives are executable only in this state. This is to prevent user processes from using primitives which are directly concerned with short term scheduling although processes can synchronize themselves by blocking using the standard \text{ECAP} implemented synchronization mechanisms. We have also
Codeword in the Table of ECAPs entry

<table>
<thead>
<tr>
<th></th>
<th>Int</th>
<th>Str</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cwd Res (relative)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cwd Res (relative)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cwd Cnt Procedure 'AWAIT'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cwd Cnt Procedure 'CAUSE'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ecap Q(N,RES)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cwd codeword to 'v'</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**EVENT (v) ➞**

**Operations:**

1. **AWAIT**
   - Operations:
     - EVOKE (EVQ,1) /* the process capability is inserted into the event Q */
     - RELEASE (CWDV) /* Release the shared segment */
     - RELEASE (EXV) /* Release the ECAP definition segment */
     - VACATE (WEVN,EXV) /* the process is blocked in WEVN status */

2. **CAUSE**
   - Operations:
     - EVOKE (EVQ,3) /* Xa indicates if the event Q is empty */
     - do while Xa = 0 /* do while the event Q is not empty */
     - begin
       - EVOKE (EVQ,2) /* remove a process cap. from the event Q into Xa */
       - RESUME (Xa) /* cause the process to be resumed */
       - EVOKE (EVQ,3) /* Xa now indicates if the Q is empty */
     - end

Assume,

- the event queue is referred to as EVQ
- the codeword to the shared segment is referred to as CWDV
- the ECAP defining the event variable is EXV

**AWAIT:**

Xa = & (PI) /* Xa now contains the Proc. Cap. of the executing process */

**CAUSE:**

EVOKE (EVQ,3) /* Xa indicates if the event Q is empty */

do while Xa = 0 /* do while the event Q is not empty */

begin
  EVOKE (EVQ,2) /* remove a process cap. from the event Q into Xa */
  RESUME (Xa) /* cause the process to be resumed */
  EVOKE (EVQ,3) /* Xa now indicates if the Q is empty */
end
introduced four states of a process—WCPU, WBLO, WSEM and WEVN.

Now that we have implemented various synchronizing mechanisms, we may specify how they will form the basis for process cooperation. The underlying principle is that of message communication implemented outside the monitor. The assumption is that associated with each process there is a message buffer, and the identification ECAP for this main buffer is part of the process description in the monitor. The monitor maintains a process description for each process in the system, which contains the following information:

i) The process capability—used to access the process state vector.

ii) Status indicator—an INT element which indicates the current status of the process as well as the identity of a resource causing a block, if it is the case.

iii) ECAP for the main message buffer associated with the process.

iv) A list of synchronization ECAPs being used by the process.

The status indicator is updated whenever the monitor primitives VACATE and RESUME are executed. The reason for the last set of information will be indicated shortly.

A process initiates communication with another process by an operation defined on the process capability of the receiver—COMMUNICATE (Proc. Cap, CWD). This feature is implemented in the second category of implementations (as an EPN operation) and the implementation code first executes a monitor primitive which returns the message buffer ECAP of the receiving process. This is the only portion where the monitor is used, as the implementation code for COMMUNICATE then sends a message to the corresponding main buffer using the ECAP. The
receiving process now knows the identity of the sender as this is inserted along with the codeword to the message in the main buffer of the receiving process. The sending process could, as part of the message pass the ECAP for a message buffer which the processes could share privately. The main message buffer associated with a process is of course available for messages from any process possessing the process capability. The COMMUNICATE operation is expanded as follows:

COMMUNICATE (Proc. Cap., CWD):

```
PROC_INFO (Proc Cap., code) /* this is a monitor primitive
    which depending on the code, returns the corresponding information from
    the process description. It puts it in a register say X_t. */

X_a = CWD /* the message CWD is put in X_a and
X_b = \sigma(PI) the sender's Proc. Cap into X_b. */

EVOKE (X_t, 1) /* 'SEND' operation is executed on
    the message buffer ECAP in X_t */
```

A process possesses the ECAP to its main message buffer and receives messages by applying the 'RECEIVE' operation on this ECAP.

Since we are allowing the setting up of private synchronization mechanisms, once communication has been established, we should provide for a monitor primitive for the process to inform the monitor of the synchronizing mechanism being used, so as to safeguard itself from being permanently blocked if the other process is terminated permanently. This is by executing a monitor primitive INFORM (ECAP), which results in the monitor adding the ECAP to the list of synchronizing mechanism ECAPs it maintains in the process information. The use
of this is later discussed.

The elegant implementation for the synchronization mechanisms accentuate how flexibility in synchronization can be provided by ECAPs. This coupled with the use of revokable codewords, sets up a very dynamic environment for interaction between processes. Needless to say, the use of the mechanisms are very simple and efficient and the flexibility of passing mechanisms through tagged elements maintain the simplicity of the interaction environment.

Summarizing, we have shown the implementation of various synchronizing mechanisms, all derived from the elementary synchronization of mutual exclusion provided by the protection structure. We have then indicated that the basis of process cooperation is synchronized message communication and shown how this can be effected with a minimum of monitor intervention. In the process of establishing the synchronization features, we have also introduced monitor primitives dealing with short term scheduling as well as maintaining centralized information about a process.

IV.4 Input/Output handling features

The uniform treatment of input/output with normal processes provides an efficient solution to one of the most important aspects of any operating system. First, let us enumerate the criteria to be satisfied by the nucleus' features supporting I/O operations:

1) The I/O needs to execute as independent processes.

2) The I/O process needs to be synchronized with the process requesting the I/O operation. This implies that it needs to signal completion of the I/O.
3) The I/O process needs to be accessible to all processes which are authorized to use the operations.

4) A process should also be able to secure exclusive access to the I/O process (like when using sequential devices).

5) I/O processes need to be treated equally with other processes with respect to short term scheduling. This implies that the hardware synchronization mechanism through interrupts needs to identify the I/O process it corresponds to.

As part of the protection structure, we had decided that a device would be represented by a Device capability with two operations defined on it:

\[
\text{IN(Dev. Cap.,CWD)} - \text{transfers the information from the device to the segment described by the codeword.}
\]

\[
\text{OUT(Dev. Cap.,CWD)} - \text{transfers the information from the segment to the device.}
\]

In designing the features supporting I/O handling, the conflict is, whether to consider these protection structure operations as elementary device operations, or whether to consider these as the complete description of I/O handling available to a process. The former viewpoint allows the definition of more elaborate I/O operations on the basis of ECAPs and restricting processes to use these. This also allows the treatment of I/O as a separate process. The latter viewpoint however requires the executing process to indicate further information about the nature of the operation required as part of the segment described by the codeword. The clinching factor though, in support of this viewpoint is that it provides a very concise description of I/O handling and ties in with the notion of defining operations on a
resource. So we will decide in favour of the latter.

We define a set of I/O processes in the system which are permanently resident. Groups of identical devices are governed by the same I/O process. These I/O processes are treated uniformly with normal processes, although preferential treatment is possible by assigning priorities in order to improve I/O response. The IN, OUT operations are implemented as first, the execution of a monitor primitive which given the device capability, returns the process capability of the I/O process governing the device specified. The implementation code of IN,OUT operations then executes a COMMUNICATE on this process capability passing the codeword describing the segment, as a message. The executing process can now carry on with normal execution.

The device capability, the ECAP for a semaphore, the nature of the operation required, the particulars identifying a subset of the device space to be used for the transfer, are all contained within the segment passed as message (Fig. 4.8). The I/O process services each message in its main message buffer as and when a device in the set of devices it controls, becomes free. Synchronization with respect to the process requesting the I/O is effected by means of the semaphore indicated in the message segment. Thus the user process can 'WAIT' on the semaphore it has specified, until the I/O process signals completion. The semaphore can later be used by the user process for other purposes also. The 'options' field in the device capability is used to specify the operations allowed to the process and the I/O process checks this to ensure that the operation specified in the message is allowed. The I/O process also indicates the result of the I/O operation in the message segment. Once the user process is unblocked on the synchronizing
Semaphore ECAP for synchron.
- nature of operation
- result of operation
- to specify a subset in device space
- Device capability
- Codeword to Numeric segment
- Numeric segment containing the information actually transferred.

IN (Dev Cap., CWD I) or OUT (Dev Cap., CWD I) are expanded as

**IN, OUT:**

DEV_INFO (Dev Cap.)  /* monitor primitive which returns the process capability of the I/O process in register X_t */

COMMUNICATE (X_t, CWD I)  /* results in sending the message to the main message buffer of the I/O process */

Fig. 4.8 The Input/Output operations
semaphore, it can access the message segment and continue with the necessary processing.

A user process indicates if it wants to reserve an I/O device in the 'nature of operation' field. The I/O process indicates the result in the appropriate field. If an I/O device has been reserved, the I/O process replies to messages from user processes requesting the device, by indicating that the device has already been reserved.

Although synchronization has been provided between the user and the I/O processes, the I/O process must still be synchronized with the physical devices. Since it is not necessary for an I/O process to be executing when its corresponding interrupt from the device occurs, we propose that all interrupts be handled within the monitor. We assume that the hardware interrupt feature provides for a transfer to the concerned monitor procedure when an interrupt occurs. In the monitor state, interrupts are disabled and queuing of interrupts occur. The monitor recognizes the class of the interrupt and activates the I/O process which is blocked, waiting for the interrupt, and then exits from the monitor state. Thus the I/O process needs to initiate a driver and execute a VACATE (WIOP,Dev Cap.) and wait in the WIOP (waiting for I/O interrupt) state till it is activated by the monitor. It is also possible to control access to devices (as may be required in file management) by restricting user processes to perform the I/O for these by using standard system defined ECAPs.

Summarizing, we have satisfied all the criteria required of the I/O handling features. The solution involves minimum use of monitor primitives, so that I/O can be executed concurrently in order to derive the maximum benefit from a multiprocessing environment.
IV.5 Process creation and control

The processes in the system are organized in a hierarchical tree structure, where the parent of a process in the hierarchy is the process which created it. At the root of the hierarchy is the basic operating system. The monitor is not to be confused with this basic operating system, for the monitor is just part of the nucleus which governs the environment in which the basic operating system can control all other processes.

We had introduced an operation for process creation as part of the protection structure. This was the PROCMAK operation defined on a Process Forming capability (PFC). A process was described by its process capability which indirectly referenced the Process State Vector (PSV). The PSV specifies the dynamic execution state of the process, but we also need to maintain a centralized data structure which contains other information regarding a process. Table 4.1 tabulates the information required to be present for each process. Since process creation and their control requires the update of this centralized data structure as also other such centralized structures, we need to implement them as monitor primitives. This has all along been the criteria for implementing a feature as a monitor primitive, namely the modification of some systemwide data structure (that is, centralized data structure).

The rationale behind the hierarchical structure is that each parent is responsible for the creation, medium term scheduling, termination of its child processes as also the specification or creation of bases with respect to which the process executes. Thus in effect, a parent process defines the operating system for its child processes. There are six monitor primitives which are provided for creating,
Table 4.1 The Process Description

The information maintained in a system wide process description table is:

1) The Process capability
2) Status Indicator (An INT element can contain the status code)
3) ECAP for the main message buffer
4) Process capability for the Parent process
5) List of ECAPs of synchronization mechanisms used by the process
6) List of resources created temporarily by the process

The status of a process can be any of the following:

a) CREATED - after PROCMAK operation (e)WBLO - Wait on block lockout
b) INITIALIZED - after INITIALIZE (f)WSEM - Wait on semap, block
c) EXECUTING - executing on processor (g)WEVN - Wait on event variable
d) WCPU - activated but not allocated (h)STOPPED - after STOP operation

Table 4.1b The Monitor Primitives

A) Process scheduling primitives -
   VACATE, SCAN_WBLO, RESUME
B) Process Control
   PROCMAK, INITIALIZE, START, STOP, CONTINUE, TERMINATE
C) Resource management
   GETRES
D) Base maintenance
   CREATE, APPEND, DELETE
E) Centralized Table inquiry or update
   PROC_INFO, DEV_INFO, INFORM
initializing and controlling child processes. These are:

1) PROCMAK (PFC) - This primitive makes an entry corresponding to the new process in the process information data structure and returns a process capability. The corresponding PSV is not initialized.

2) INITIALIZE (Proc Cap,CWD) - This initializes the PSV according to the information provided in the segment described in the codeword. Of course, in an actual system, procedures could be available which given the process capability to the uninitialized PSV, defines a standard PSV using this primitive. This standard PSV could possibly contain marked links in the stack set to error recovery procedures.

3) START (Proc Cap,CWD,Base Cap) - This primitive activates a process with respect to the specified base, control initiating within the control segment defined by the codeword.

4) STOP (Proc Cap) - This stops the activation of the child process indicated by the process capability. It also needs to stop all descendant processes down the hierarchy, which have not already been stopped by their parents.

5) CONTINUE (Proc Cap) - This resumes the activation of the child process indicated, as also those processes which were stopped when this child process was originally stopped.

6) TERMINATE (Proc Cap) - This terminates a process permanently and removes the process description corresponding to the process.

It is obvious as to how a parent process can determine the medium term scheduling strategy of its descendant processes using the STOP and CONTINUE primitives. More important however is the effect on synchronization and input/output. The STOP primitive has to ensure that all I/O with respect to the process is completed. Processes blocked on
synchronizing conditions with respect to the process being stopped remain unaffected. However, in the case of TERMINATE, the monitor needs to ensure that all these blocked processes are unblocked. This is where the list of synchronization mechanism ECAPs, maintained in the process description, becomes useful. The monitor scans this list and unblocks all the processes which are blocked with respect to the terminated process. This is of course the price to pay for a flexible scheme of synchronization.

We also need to establish the relationship of the process with respect to the base, in the environment provided by the nucleus. Whenever, a process is initially activated, it is done so with respect to a base. The parent process is entrusted the responsibility of creating the base and hence determining the execution environment of the process. A base is created by a CREATE operation on a base forming capability and on a codeword describing a resource segment. The new base formed, comprises of this resource segment. We need to implement CREATE as a monitor primitive. Of course, one could have a procedure available to all processes, which defines the standard portion of a base consisting of codewords to system implementation procedures, accessible system tables and some base level capabilities like CRA, CFC, Device capabilities (to the operator console etc.). The calling process can update the rest of the base with resources it possesses.

A base since it is created by an ancestor process, is a permanent entity with respect to the process' life time. Hence, the process also maintains a temporary set of resources which it obtains by calling the RESOURCE MANAGER. All these temporary resources are indicated in the process description (Table 4.1) and are automatically
destroyed when the process is terminated. However, a process can by means of the APPEND operation, add a resource to the current base, thus requiring the resource to be removed from the set of temporary resources indicated in the process description. Thus APPEND (introduced in Chapt. III) needs to be implemented as a monitor primitive as it implies that the resource added to the base becomes permanent with respect to the life time of the process. The same is true of a DELETE operation which removes a resource from a base. Thus by entrusting the responsibility of a resource to the process which requests it, we can provide a restricted solution to the 'lost object' problem. Also the base can now be considered as the permanent record of a process' activation.

IV.6 Short term Resource Management

The monitor is responsible for the short term scheduling of the processes. It does this by maintaining a queue (could be a priority queue) of processes called WCPU queue. A process, when activated either from a blocked state by the RESUME primitive or from a 'created' state by the CONTINUE primitive, gets entered into this queue (the process capability is entered). An external hardware timer provides an interrupt at fixed intervals and the interrupt handler, if it recognizes the interrupt as a timer interrupt, pre-empts the executing process and enables the first process in the WCPU to execute. The pre-empted process is now entered at the tail of the queue thus ensuring a 'round robin' discipline. If more than one processor is present, a timer could be provided for each processor. Priorities could also be assigned to processes in the WCPU queue. This enables faster I/O processing as well as improved response for processes interacting with real time events.
requiring immediate processing. Since the scheduling is automatically handled within the monitor, no primitives are necessary although VACATE and RESUME have been provided for pre-emption and unblocking.

Finally, as earlier stated, a process can obtain resources by calling the RESOURCE MANAGER. The Resource Manager needs to be implemented within the monitor as it has to return a new capability or codeword and this implies a modification of the Master Segment Table or one of the various Tables of capabilities. Without going into the details of the implementation, we state that a process obtains a resource from the resource manager by executing the monitor primitive GETRES (INT), where the integer is a code for the type of resource required. We could also have the facility by which a process, when it creates a child process, can specify limit values for various types of resources which seals the total quantity of a resource which the child process can get from the Resource Manager. The limit values, to which the parent is constrained are decremented by this amount. The limit values of the parent can be reset to the original values when the child process is terminated. Table 4.1b summarizes the nucleus' features which are implemented as monitor primitives.

IV.7 Conclusion

We have, in this chapter, organized the operations constituting the protection structure of chapter III into a set of features which form a framework for designing operating systems. We have recognized the four classes of features - process cooperation, input/output, process creation and control, short term resource management; which are essential to any operating system and indicated a scheme for implemen-
iting them. This implementation concentrates on the support of an environment, where the policy defining part of the operating system is treated as a set of concurrent cooperating processes. Furthermore, the protection structure features of flexibility (as provided by revocation) and extensibility can be applied to enable the nucleus to provide a dynamic environment. Finally, the nucleus also stresses on the simultaneity of as many features as possible. Tables 4.2a,b,c tabulate all the operations and features suggested in this thesis.
<table>
<thead>
<tr>
<th>Operand Class</th>
<th>Level of Implementation</th>
<th>Is Monitor Prim. Used</th>
<th>Operation</th>
<th>Function of the Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Hardware Instructions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>see Table 3.1</td>
<td>Level 0</td>
<td>No</td>
<td>MPN instrs. including MARK &amp; CALL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>No</td>
<td>RETURN n</td>
<td>Procedure return to stacked link.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No</td>
<td>TST(Integer)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No</td>
<td>REDINC(Integer)</td>
<td>Multiprocessor instrs.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No</td>
<td>SETPRIV</td>
<td>to enter privileged mode.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>No</td>
<td>REMPRIV</td>
<td>to exit privileged mode.</td>
</tr>
<tr>
<td><strong>Operations defined on codewords and revokable codewords (CWD &amp; RCOD)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CWD or RCOD</td>
<td>1</td>
<td>Yes</td>
<td>EVAL(COD) $'$COD = CWD or RCOD$'$</td>
<td>to form a pointer (possibly for exclusive use)</td>
</tr>
<tr>
<td>not Rel. do.</td>
<td>1</td>
<td>No</td>
<td>RELEASE(COD)</td>
<td>to release a segment after excl. access</td>
</tr>
<tr>
<td>Res Cwd</td>
<td>3</td>
<td>Yes</td>
<td>GROUP(Res Cwd)</td>
<td>to collapse one level of a tree structure</td>
</tr>
<tr>
<td>Rel Cwd</td>
<td>2</td>
<td>No</td>
<td>EXPAN(Res Ptr.) points to Rel Cwd</td>
<td>to form the pointer describing sub-seg.</td>
</tr>
<tr>
<td><strong>Revocation Features for codewords, base level capabilities, extended caps.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all Res elements (no Int)</td>
<td>2</td>
<td>Yes</td>
<td>SEAL(CWD/RCOD/CAP/RCAP/ECAP/RCAX)</td>
<td>to form revokable version of a resource.</td>
</tr>
<tr>
<td>RCOD, RCAP, RCAX</td>
<td>2</td>
<td>No</td>
<td>REVOKE(RCOD/RCAP/RCAX, Int)</td>
<td>to revoke privileges</td>
</tr>
<tr>
<td>do.</td>
<td>2</td>
<td>No</td>
<td>LOCK(RCOD/RCAP/RCAX)</td>
<td>to localize the effect of revocation on the revokable version.</td>
</tr>
</tbody>
</table>
Table 4.2b (contd.) Table of Operations and Nucleus' Features

<table>
<thead>
<tr>
<th>operand class</th>
<th>is monitor prim.</th>
<th>Operation</th>
<th>function of the operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations defined for base level Capabilities (Cap &amp; Rcap)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null</td>
<td>No</td>
<td>TAG(CFC)</td>
<td>to create a tagged element.</td>
</tr>
<tr>
<td>CFC</td>
<td>Level 0</td>
<td>No</td>
<td>IN(Dev Cap,COD)</td>
</tr>
<tr>
<td>Device</td>
<td>3</td>
<td>Yes</td>
<td>OUT(Dev Cap,COD)</td>
</tr>
<tr>
<td>Process</td>
<td>2</td>
<td>Yes</td>
<td>COMMUNICATE(Pro Cap,COD)</td>
</tr>
<tr>
<td>do. Monitor</td>
<td>-</td>
<td>INITIALIZE(Pro Cap,COD)</td>
<td>to initialize the PSV.</td>
</tr>
<tr>
<td>do. do.</td>
<td>-</td>
<td>START(Pro Cap,COD,Base Cap)</td>
<td>to start process activation.</td>
</tr>
<tr>
<td>do. do.</td>
<td>-</td>
<td>STOP(Pro Cap)</td>
<td>to temporarily stop a process.</td>
</tr>
<tr>
<td>do. do.</td>
<td>-</td>
<td>CONTINUE(Pro Cap)</td>
<td>to restart process after stop.</td>
</tr>
<tr>
<td>do. do.</td>
<td>-</td>
<td>TERMINATE(Pro Cap)</td>
<td>to terminate activation.</td>
</tr>
<tr>
<td>do. do.</td>
<td>-</td>
<td>VALIDATE(Pro Cap,COD)</td>
<td>to validate pointers in PSV, corres. to CWD.</td>
</tr>
<tr>
<td>Base</td>
<td>Level 2</td>
<td>No</td>
<td>CHANGET(Bas Cap)</td>
</tr>
<tr>
<td>do.</td>
<td>3</td>
<td>No</td>
<td>TRANSD(Bas Cap,COD)</td>
</tr>
<tr>
<td>do. Monitor</td>
<td>-</td>
<td>APPEND(Bas Cap,resource)</td>
<td>to add to a base.</td>
</tr>
<tr>
<td>do. do.</td>
<td>-</td>
<td>DELETE(Bas Cap,Int)</td>
<td>to remove from a base.</td>
</tr>
<tr>
<td>Name</td>
<td>Level 2</td>
<td>No</td>
<td>CONNECT(Name Cap,Bas Cap,Int)</td>
</tr>
<tr>
<td>do.</td>
<td>2</td>
<td>No</td>
<td>NAMF(Name Cap)</td>
</tr>
<tr>
<td>CRA</td>
<td>0</td>
<td>No</td>
<td>REG(CRA Cap)</td>
</tr>
<tr>
<td>PFC Monitor</td>
<td>-</td>
<td>PROCMAK(PFC Cap,COD)</td>
<td>to create new process.</td>
</tr>
<tr>
<td>BFC do.</td>
<td>-</td>
<td>CREATE(BFC Cap,COD)</td>
<td>to create new base.</td>
</tr>
<tr>
<td>EFC</td>
<td>Level 3</td>
<td>Yes</td>
<td>EXTEND(EFC Cap,COD)</td>
</tr>
<tr>
<td>INC</td>
<td>2</td>
<td>No</td>
<td>DOWNGRADE(INC Cap,Int)</td>
</tr>
<tr>
<td>do.</td>
<td>2</td>
<td>No</td>
<td>UPGRADE(INC Cap,Int)</td>
</tr>
</tbody>
</table>
### Table 4.2c (contd.) Table of Operations and Nucleus' Features

<table>
<thead>
<tr>
<th>operand class</th>
<th>level of implem.</th>
<th>is monitor prim. used</th>
<th>Operation</th>
<th>function of the operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'data proc' ECAP</td>
<td>Level 3</td>
<td>No</td>
<td>EVOKE(Ecap,Int)</td>
<td>apply an 'operation' on the struct. of ECAP.</td>
</tr>
<tr>
<td>'data struc' ECAP</td>
<td>2</td>
<td>No</td>
<td>STRUCT(Ecap)</td>
<td>to form pointer to the structure.</td>
</tr>
<tr>
<td>do.</td>
<td>3</td>
<td>Yes</td>
<td>CONVERT(Ecap,CWD/EFC)</td>
<td>to convert 'data struc' to 'data proc'.</td>
</tr>
</tbody>
</table>

**Rest of Monitor primitives**

<table>
<thead>
<tr>
<th>Privileged mode</th>
<th>Monitor</th>
<th>VACATE(status,resource)</th>
<th>pre-empts executing proc. in given status.</th>
</tr>
</thead>
<tbody>
<tr>
<td>do.</td>
<td>RESUME(Pro Cap)</td>
<td>resumes the blocked process.</td>
<td></td>
</tr>
<tr>
<td>do.</td>
<td>SCAN_WBLO(COD)</td>
<td>scans WBLO Q and frees processes blocked on COD.</td>
<td></td>
</tr>
<tr>
<td>do.</td>
<td>DEV_INFO(Dev Cap)</td>
<td>returns Pro Cap of the I/O process.</td>
<td></td>
</tr>
<tr>
<td>do.</td>
<td>PROC_INFO(Pro Cap)</td>
<td>returns ECAP of main message buffer of proc.</td>
<td></td>
</tr>
<tr>
<td>do.</td>
<td>GETRES(Int)</td>
<td>returns resource indicated by code.</td>
<td></td>
</tr>
<tr>
<td>Any mode</td>
<td>INFORM(Ecap,Pro Cap)</td>
<td>informs monitor of synchron. ECAP.</td>
<td></td>
</tr>
</tbody>
</table>

**Synchronization Mechanisms**

<table>
<thead>
<tr>
<th>Semaph.</th>
<th>Ecap</th>
<th>Yes</th>
<th>SIGNAL(Ecap)</th>
<th>'signals' semaphore.</th>
</tr>
</thead>
<tbody>
<tr>
<td>do.</td>
<td>Ecap</td>
<td>Yes</td>
<td>WAIT(Ecap)</td>
<td>'waits' for sem. signal</td>
</tr>
<tr>
<td>Message buffer</td>
<td>Ecap</td>
<td>No</td>
<td>SEND(Ecap)</td>
<td>'send' msg. to buffer.</td>
</tr>
<tr>
<td>do.</td>
<td>Ecap</td>
<td>No</td>
<td>RECEIVE(Ecap)</td>
<td>wait for msg. in buff.</td>
</tr>
<tr>
<td>Event var.</td>
<td>Ecap</td>
<td>Yes</td>
<td>AWAIT(Ecap)</td>
<td>wait in event queue.</td>
</tr>
<tr>
<td>do.</td>
<td>Ecap</td>
<td>Yes</td>
<td>CAUSE(Ecap)</td>
<td>re-activate all procs. in event queue.</td>
</tr>
</tbody>
</table>
Chapter V

Analysis of the System

An analysis of computer systems like the one proposed can only be meaningful if conducted on the basis of a simulation study. This requires a simulation of the entire architecture as well as the overlaid software which can be considered as a separate thesis topic in itself. Furthermore, as there are so many 'design criteria' in such a complete system, it is difficult to evaluate the relevance of each in a separate analysis. Throughout the progress of this thesis, we have, at each stage, endeavoured to point out the motivation and rationale for each of the design decisions made.

In conducting an analysis of the system, we will follow the strategy of enumerating a list of 'apparent disadvantages' to the system. The underlying idea is that these features will be the ones to attract the attention of a critical evaluator of the system. We have indicated these as 'apparent disadvantages' because they are not inherent in the system, but arise because of a tradeoff with respect to other criteria. To continue the analysis, we have to show the nature of this tradeoff, leaving the question of striking a suitable balance in this tradeoff as an implementation aspect depending on the specific utilitarian value the system is put to. Apart from the apparent disadvantages, the intrinsic limitations in the system are also enumerated and finally the analysis completed by highlighting the advantages of the system.

Enumerating the apparent disadvantages in the system:

1) The hardware checks of the operand tags for each operation seems to be an expensive overhead, as in most cases the check does
In the basic hardware, we are using a register extension for each of the processor registers. Apart from the hardware cost, the additional complication arises, of specifying the effect on the extension for each operation defined.

3) The stack convention of stacking the register extension when the DUMP operation is applied on a POINTER seems unnecessary.

4) The definition and use of a privileged mode of operation is contradictory to the treatment of protection as the control of accessible objects.

5) The use of different tags (RCOD,RCAP,RCAX) for the revokable versions of codewords and capabilities implies that a revokable version is treated differently from the original form.

6) The GROUP operation is restricted in its flexibility, as it does not apply on revokable versions and also, it is implemented by copying the second level data structure.

7) The use of the privileged mode in certain ECAP (extended capability) implementations, specifically semaphores and event queues, is contradictory to the interpretation of an ECAP as a normal procedure applying on a data structure.

8) With respect to the synchronizing mechanisms, two cooperating processes are dependent on each other if they use private means of synchronization.

9) A resource obtained by a process is automatically deleted when the process is terminated.

We now provide an explanation for the tradeoffs involved in each of the above disadvantages.
1) The basis for the evolution of the entire system has been a restriction of the operations valid on an operand. Thus it is essential for the integrity of the entire system, to maintain the hardware check on tags. Of course, schemes to minimize the overhead can be developed, like a dedicated processor to conduct the check of the tag validity in parallel or by using a pipelined processor which conducts the various stages of instruction execution in parallel. The latter scheme proves to be more attractive.

2) The MPN architecture relies on a scheme of restricting the creation and maintenance of pointers. As stated in chapter III, the structure of a pointer necessitates that an extension be specified so as to indicate the context in which the pointer is used by containing the codeword (or its revokable version) from which the pointer was derived. Thus for proper maintenance, this extension is necessary, although it need not be present in hardware but remain as part of the Process State Vector in memory. The specification of an operation on this extension is not very critical as illustrated by the following observations:

i) An extension needs to be updated, only if a pointer is the result of an operation.

ii) A pointer can be the result of an operation only in the following cases - COPY; Address manipulating instructions like MOD, LIM and PTS; REG; UNDU; EVAL and TAG.

iii) For COPY and Address instructions, the same or restricted version of the operand pointer is formed, so that the extension just needs to be copied from that corresponding to the operand.

iv) For REG, control registers have their extensions and the
corresponding one needs to be copied.

v) For UNDU, the extension is stored as the next element to the pointer, in the stack. The use of this will be indicated shortly.

vi) For EVAL, the operand is itself copied to the extension.

vii) Finally TAG operations are restricted by CFC capabilities to prevent them from forming pointers.

Thus the additional specification needed, simplifies to an indication of where the extension is to be copied from, for a small sub-set of operations.

3) Apart from the fact that a pointer extension is needed to update the register extension when unstacking using UNDU, the codeword, stored with a pointer in the stack, serves a more important purpose. This is brought about because a pointer is the only means of accessing memory. Hence, if a process obtains access to a 'mutually exclusive' segment by forming a pointer through the EVAL operation, it later releases the segment by a RELEASE operation. It is crucial for this RELEASE operation to invalidate all pointers to the segment which the process may possess. The assumption is that the implementation of the RELEASE results in a scan of the extensions to the registers and the invalidation of all those pointers (by setting to NULL) which correspond to the codeword being released. However, in order to account for pointers which may be stacked, it also needs to scan the complete stack. Rather than performing this costly operation, we solve the problem by implementing UNDU so that whenever it has to unstack a pointer, it establishes its validity by performing an EVAL like operation on the associated codeword which has also been stacked.

The above implementation also supports the immediate revoc-
ation of codewords (Section III.7). Here the monitor primitive VALIDATE is available to all processes, the operands being a process capability and a codeword. When a process, specifically wants to ensure that another process does not have any pointers to a segment it no longer has access to; it executes VALIDATE (Proc Cap, CWD) which results in a revalidation (by an EVAL like operation) of all pointers, corresponding to the indicated codeword, in the registers of the process specified. Automatic validation of pointers in the stack is taken care of by the UNDU operation. It needs to be pointed out that both UNDU and VALIDATE need to execute only an EVAL like operation, in that the indirection path to the Master Segment Table entry needs to be followed and the pointer validated against this entry without the subsequent problems of synchronization as enforced by EVAL. Thus the need for a register extension is a tradeoff with the requirement of an elementary operation which can provide synchronization, as also with the need for immediate revocation.

4) The privileged mode of operation is used only for implementation code corresponding to an EPN operation. It is not available to the process hierarchy including the basic operating system, but only used to make the protection structure an extension of the hardware. Thus the treatment of protection remains unchanged.

5) The implementation of the revocation mechanism calls for the distinction between a resource representation (codeword or capability) and its revokable version. The operation of revocation of privileges needs to be applicable only on revokable versions. On the other hand, the GROUP operation on a codeword is to be restricted to non revokable versions in the first level structure. However, in the context of a
resource representation being used to identify the resource and to define the actions allowed on the resource, the normal and revokable versions are undifferentiated.

6) Since pointers are not allowed in segments in memory, relative codewords were introduced to define intra segment structure. But these relative codewords are not codewords in the normal sense as they do not refer to a segment through an indirection scheme but actually point to a subsegment within the segment in which they are present. Thus they serve the role of intra segment pointers. Now, since a segment in the second level of a structure (the concept of codewords defines all structures as tree structures) is actually added to the first level segment, by the GROUP operation; we cannot have the second level segment defined by a revokable codeword, because this revokable codeword has to be converted into a relative form, implying that there is no revocation control for the segment. Furthermore, since there may be other codewords to this second level segment, we can only have a copy of this segment added to the first level segment.

7) We had earlier indicated that the privileged mode is only used for implementation code corresponding to an EPN operation. Therefore, the use of the privilege mode in certain ECAP procedures, specifically those corresponding to semaphores and event queues, appears to contradict this statement. The confusion arises because semaphores and event queues are nucleus’ features implemented as an ECAP so that any process can avail of this feature by obtaining an ECAP for it. We need procedures for these to be executed in privileged mode as the use of the two short term scheduling monitor primitives - VACATE and RESUME, has to be restricted. Thus, we modify our contention to state that privileged
code is only used to implement nucleus* features in the form of EPN operations, ECAPs and monitor primitives. As far as processes are concerned, they are unaware of the existence of a privileged mode. In ECAPs the type field is used to indicate that a nucleus* feature is implemented, whereupon the EVOKE expansion takes care to transfer to the procedure in privileged mode.

8) The idea in providing synchronization mechanisms as ECAPs is to facilitate processes to obtain these as resources for private communication between cooperating processes. A process can still communicate using the COMMUNICATE operation if suspicion between processes exists. Thus private communication (this also includes synchronization) is used only if the processes trust each other and to provide some protection they inform the monitor of their communication mechanism by the INFORM primitive.

9) The temporary nature of a resource ties in with the concept of regarding a base as a permanent record of a process* execution. Thus the parent process which creates the base, can allow the child process to add to the base, whereupon the resource becomes permanent with respect to the life time of the base. Since a base itself is a resource, it only exists during the lifetime of the process which created it. Thus the basic operating system, at the root of the process hierarchy, is permanently resident and maintains a record of all process activations in the set of bases it has created.

Apart from the 'apparent disadvantages', the proposed system has some intrinsic limitations:

1) The capability concept relies on the possession of unique IDs to reference an object. This is to ensure that once an object is
destroyed, then any existing capabilities automatically become invalid. The finite nature of the ID field implies that it is not possible in the system's life time. Thus the system has to provide for a procedure which scans all resource descriptions in the system and nullifies those referring to non existent objects and then reassigns IDs. This can be facilitated by executing this procedure when only the basic operating system is resident, all other processes being terminated. Thus the procedure needs to consider the set of bases possessed by the basic operating system as the roots of trees and traverse down each one of them.

2) There are also limitations with respect to store management. Throughout the discussion, the implicit assumption is that store management in terms of a virtual store, segmentation and paging are automatically taken care of. The conflict arises though, because EPN segments are small compared to normal page sizes. Moreover, when a segment has to be expanded, the store manager needs to take care of this, perhaps by relocating the segment. This gives rise to the additional complication that all pointers to the old segment area have to be invalidated if it has to be used again.

3) Finally, the operating system implementation as developed is not transferable to other architectures. Of course, the basic architecture itself could be simulated but this is hardly a practical scheme.

The various features of the system have been introduced in earlier chapters of this thesis. We now enumerate those which enhance the applicability, flexibility and efficiency of the whole system:

1) The system provides a novel solution to reducing the overhead
of affixing tags to elements in storage by making an element in memory identifiable, not by a tag associated with it but by the pointer with which it can be accessed.

2) The dynamic chain maintained in the stack provides a scheme for return of control to pre-determined restart points. Thus one can define restart points corresponding to error recovery procedures for various exceptional conditions. In addition, the hardware check of the failure of an operation on an operand could itself be implemented as a return to one of these restart points.

3) The revocation feature, provided at little overhead, meets one of the foremost objections to using capability structures. This is provided at the cost of a very marginal loss in flexibility of capability handling. The revocation scheme recognizes the need for obtaining both a copy of a capability (or codeword) with all the facilities associated with the original one; as well as a revokable version of a capability wherein the original capability maintains a direct control over the revokable version.

4) The GROUP operation provides an elegant mechanism for compacting a structure. All structures in storage are defined as tree structures. Of course, as shown in various implementations, this is not a serious restriction as any other structure can be mapped into this. The GROUP operation, by forming relative codewords, is able to provide a concise definition of such structures within a segment.

5) The concept of extended capabilities (ECAP) is one of the most powerful features provided by the system. It enables a generalization of the basic idea underlying the system - 'the restriction of an operation on an operand', to 'the restriction of a sequence of operations
on a set of operands*. In addition, it is also possible to represent a data structure as one manipulable entity by using the 'data structure' version of an ECAP (the 'S' bit is zero). In this case, the 'options' field in this ECAP version could be used to apply in conjunction with the codeword defining the data structure.

6) The system can be complemented with a set of procedures available universally to all processes, which aid in the formation of new processes, bases and the definition segments for standard system defined ECAPs. This creates a more user oriented environment. The system can also provide software facilities like file handling procedures, application packages in the form of ECAPs thus ensuring that they are protected against user processes.

7) Most of the system design has been made to ensure the protection of system features from user processes. However, user processes can also protect themselves by using the TRANSD operation. This facility enables users which have created a restricted base to transfer to a procedure and execute in this restricted base. Alternatively a procedure can be constrained to execute in a special base associated with it and which is the only base which allows the execution of the procedure. Thus cooperation between mutually suspicious processes is also possible. We can therefore conclude that the system supports protection facilities to handle user interactions, although this was not the primary motivation for the protection mechanisms. It needs to be pointed out that there are two aspects of protection in the system. The first is the physical aspect of protection which the basic hardware enforces with respect to segments in storage; the second is the logical aspect of protection which the protection structure enforces with respect to
8) The system supports a scheme of differentiated medium term and long term resource handling. This is provided by the dynamic nature of the process base linkage. A process executes only on a medium term basis and can obtain resources (within a limit set by the parent process) during its lifetime. It may maintain these by simply having a resource segment (pointed to by a fixed processor register) which it considers as its variable base. The parent process has of course provided a permanent base in which the process maintains a record of its activation. This provides the long term resource handling feature. The resource segment serving as a temporary base is destroyed along with the process, thus providing medium term resource handling.

We have analysed the system by considering the apparent disadvantages in the system and stating the reason for these. We have also considered a set of intrinsic limitations to the system and finally pointed out the highlights of the system. This completes the analysis.
Chapter VI

Conclusion

VI.1 Summary

This thesis has integrated the ideas of protection, descriptor oriented architectures, and the nucleus of a multiprocessing operating system to develop a flexible and generalized computer system. The initial emphasis was on the development of a capability model of protection which supported the generalized notion of a resource. This was followed by the recognition of an architecture which naturally adapted itself to this model. Once the applicability of a descriptor oriented architecture to the protection model was shown, the architecture was enhanced to satisfy the requirements of flexibility and extensibility. This enhanced architecture was then adapted to develop, in a multiprocessing environment, a framework for the systematic design of any operating system.

The efforts in this thesis have been directed towards suggesting a practical utilization of the architecture. The motivation for this has been to prove the practicality of many concepts and models which have been developed. Moreover, it illustrates the feasibility of applying the class of architectures discussed as a basic tool in system development. The practicality consideration involved the detailed specification of the protection structure as evolved in chapter III.

Finally, the application of this practical implementation to the specific problem of developing a flexible and viable framework for designing any operating system, was considered. Thus, the thrust of this thesis is the formulation of a practical scheme to use a descriptor oriented architecture as a natural support for developing operating
systems in a multiprocessing environment.

VI.2 Areas for future development

No research is complete if it does not open up avenues for future development. We have therefore identified four areas of further research, which can be investigated and developed on the basis of the computer system as evolved till now:

1) The motivation for the protection structure was to provide a recognition of higher level constructs at the assembly language level in order to facilitate system programming. Later on, specific higher level constructs, in connection with data structure specification and synchronization, were introduced. But still the possibility of errors in assembly language coding remained. Therefore, the development of a system programming language, which fully utilizes the protection constraints dictated by the architecture seems attractive. This may, by compiler checks, detect possible protection violations, but more important, by defining specific language constructs, time dependent errors caused by synchronization could be prevented.

2) A more attractive user environment could be provided, if by using the protection mechanisms, a scheme to associate environments with procedures is developed. Procedure associated environments need to be dynamic and the system needs to ensure the automatic detection of these execution environments. Thus, the resolution of protection is to be shifted from a 'process' basis to a 'procedure' basis.

3) The power and flexibility of the extended capability concept has not yet been fully exploited. The applicability of this concept now needs to focus on the important consideration of parallel processing.
In the proposed system, parallel processing has been provided for on a process basis in that the process is defined as an 'independently schedulable unit'. The limitation here, is that the overhead of process creation and maintenance prevents the feasibility of parallel processing on a 'intra statement' or even on a 'intra procedure' level. The extended capability concept suggests the possibility of an attractive scheme, wherein the application of an operation within the extended capability can be considered to execute concurrently, synchronization being provided by a built in semaphore. Thus one may even extend the extended capability idea to include numeric operations on arrays and such structures, thus providing a controlled and synchronized scheme for concurrent execution.

4) In the development of the nucleus of the operating system, we have pointed out that we need to provide many features outside the monitor. This is to prevent the monitor from being a bottleneck within the system. With the accent on parallel processing, the next stage of system development naturally follows into the area of distributed operating systems. We have provided for this at a very basic level by disassociating some features of the nucleus from the 'monitor' and by indicating the separation of the policy defining operating system from the mechanism implementing nucleus. Further resolution of this scheme is called for, wherein different processes executing concurrently can simultaneously avail of various features provided by different operating systems. It may be necessary, for the model developed for distributed operating systems, to be applicable on existing architectures. This implies that the study should concentrate on deriving the essential features of descriptor oriented architectures as applied to the support
of the model and then showing the universality of these features to other architectures.

Thus the proposed system calls for specific investigations into the four broad areas of language development, protection, parallel processing, and operating systems.
Appendix A.1

The Micro PN machine

This section of the appendix describes a machine structure called the Micro Pointer Number machine (micro PN) [II 75], a modified version of which serves as the base for the implementation of the protection structure described in Chapter III. The machine was originally proposed as the specification of a secure and efficient form of microcode which would support the direct interpretation of high level languages. The design was in the context of a microprocessor, in that the execution rate of the machine instruction set was geared to the speed of an elementary arithmetic and logic unit. The microprocessor retains the space and speed advantages of language oriented design while meeting the general microsystem requirements of extensibility, security and register allocation.

A) Micro PN tagged elements

The micro PN machine is a particular instance of the generalized pointer number machine described in Chapter II. It is defined in terms of operations on a set of 16 general purpose and 4 control registers, all of which are 32 bits width. A byte addressable program space of 64K bytes is assumed, accessible as 8 bit bytes, 16 bit half-words and 32 bit words. Table A.1 illustrates the possible formats of the elements in the registers. The tagged elements are:

\[
\text{INTEGER } (u_4, v_1, n_8) \text{ (The convention for field specification is } \text{ } u_4 \text{ - implies a 4 bit field called the 'u' field) The integer value is contained in 'n'; the 'v' field is used as an overflow byte for multi-length arithmetic and field extraction. The 'u' field can be used as a user tag and is not interpreted by hardware.} \]
Table A.1 The Micro PN tagged elements

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td></td>
<td>Indicates privilege mode of operation</td>
</tr>
<tr>
<td>m</td>
<td></td>
<td></td>
<td>Link* used when a control pointer is stored</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td>b: object/segment number</td>
</tr>
<tr>
<td>v</td>
<td></td>
<td></td>
<td>v: overflow byte</td>
</tr>
<tr>
<td>u</td>
<td></td>
<td></td>
<td>u: user tag</td>
</tr>
<tr>
<td>Typ</td>
<td></td>
<td></td>
<td>Typ: capability or codeword</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td>Class</td>
</tr>
<tr>
<td>FBL</td>
<td></td>
<td></td>
<td>FBL: first byte location</td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td>SP: Stack Pointer (defines the current stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>frame)</td>
</tr>
<tr>
<td>IP</td>
<td></td>
<td></td>
<td>IP: Instruction Pointer (indicates the next</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>instruction)</td>
</tr>
<tr>
<td>MP</td>
<td></td>
<td></td>
<td>MP: Mark Pointer (defines the previous stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>frame)</td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td>BP: Base Pointer (defines the set of resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>available)</td>
</tr>
</tbody>
</table>

Address Pointer Tags: BYTE, BYTEER, HWD, HWDR, RES, RESR, MIX, MIXR

Control Registers:

IP : Instruction Pointer (indicates the next instruction)
MP : Mark Pointer (defines the previous stack frame)
SP : Stack Pointer (defines the current stack frame)
BP : Base Pointer (defines the set of resources available)
CAPABILITY \((h_4, b_8)\) The 'h' field indicates the capability class while the 'b' field serves as a capability ID. The resources defined by a capability are:

a) Null - it has no attributes but is convenient to represent an undefined element.

b) Process - provides the reference to the process description which may contain the program registers, register stack, control pointer, base pointer and the process status.

c) Device - provides the device definition.

d) Name - formalizes the part played by identifiers in establishing connections between program modules. It can be associated with a descriptor by means of a microsystem function.

e) Base - defines the execution environment of a process. It refers to a collection of resources (i.e. capabilities, codewords and integers).

f) CFC - (capability forming capability) enables the formation of a new tagged element as described in the 'TAG' instruction.

g) CRA - (control register access) enables the access of a control register, the nature of access being indicated by the 'b' field of the CRA capability. The 'REG' instruction clarifies the idea.

CODEWORD \((b_12)\) (CWD or CWDR, where R indicates a 'read only' access) A codeword is a specialized form of capability used in program store management. The store can contain a maximum of 4096 segments of upto 4096 bytes each. The segment class (Control, Numeric, Resource or Mixed) is contained as part of the segment and not in the codeword. The 'b' field indexes into an entry in the Master Segment Table (MST), which indicates the actual location of the segment. The segment begins
with a descriptive half word containing the length of the segment and the class to which it belongs. This idea is illustrated when the 'EVAL' function is considered (Fig. A.1).

**CONTROL POINTER (m12, f16) (CTRL)** This element refers to a 16 bit microinstruction through the 'f' field. The privileged mode of operation is indicated by the 'p' field. When used as the next instruction pointer -IP or as a return link in the process stack, two more fields are used, namely the 'm' field to contain the mark value and a 'd' field to serve as a dynamic link reference in the process stack. In the IP, the 'd' field is used to contain the condition codes.

**BYTE POINTER (m12, f16)** - (BYTE, BYTER, where for all address pointers the 'R' indicates 'read only' access) The Byte address pointer refers to a numeric sequence in program store starting at address 'f' and ending at address 'f+m'. Thus the limit field 'm' is one less than the number of elements.

**HALFWORD POINTER (m12, f16)** - (HWD, HWDR) These address pointers refer to halfword accessible numeric sequences starting at address 'f' and ending at address 'f+2m+1'. The 'f' field is constrained to be even.

**RESOURCE POINTER (m12, f16)** - (RES, RESR) The elements of a resource segment are codewords, capabilities or short integers (u4, v8). The resource address pointers refer to sequences of 16 bit resource descriptors.

**MIXED POINTER (m12, f16)** - (MIX, MIXR) The mixed pointer refers to a sequence of tagged elements in store. Mixed segments are allowed to contain the resource descriptors allowed in a resource segment as also other pointers. Each element is 32 bits in size, identical to the
format in registers. The segment extends from address 'f' to 'f+4m+3'.

The sixteen general purpose registers are designated as X0, X1 .... Xf. The four control registers contain pointers and are CFCTRL, BPERES, MPIMIX and SPIMIX. The latter two pointers are used to access a process stack which is associated with each process. The operation of this stack (referred to as register stack earlier) is elaborated in Section III.3.

B) Micro PN instructions

The instruction set of a tagged machine provides for the normal arithmetic and logic operations on numeric data as well as a set of special functions for operating on capabilities, codewords and pointers. A fuller discussion of the instruction set of a tagged machine can be found in the reference [Il 72]. The following points amplify the design decisions in the micro PN instruction set specified in Table A.2.

IF...GOTO... The set (f=0,3 and 7B) provides for transfer of control within the executing control segment, with the offset value being checked by the assembler. Control can also be transferred to a new control segment indicated by the CTRL element by using a 7C or 7D instruction.

DATA (f=0) The offset 'n' relative to the CP is checked by assembler. The instruction provides a means of loading into the XO register a pointer to data or instructions in the control segment. Only Name, CFC and CRA capabilities can be held in control segments.

Arithmetic Functions (f=6 0 to 6 9) Arithmetic is 16 bit with overflow into the 'v' byte. The user tag in X is unchanged. The second
### Table A.2 Micro PN Instructions

<table>
<thead>
<tr>
<th>fg</th>
<th>h</th>
<th>Mnemonic</th>
<th>Validity</th>
<th>Operation</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>GOTO</td>
<td>Assemb. IP = IP + n_s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>DATA</td>
<td>Assemb. X_{0h} = (IP+n_s); X_{0l} = IP+n_s+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>MOD'</td>
<td>X Addr X_0 = X' n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2c</td>
<td>IFGO</td>
<td>Assemb. If c, IP = IP - 2n + 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Shifts</td>
<td>X*Int Integer shift etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>COPY</td>
<td>X = n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6g</td>
<td>3</td>
<td>Arith</td>
<td>X*Int ADD AND MASK NEQ Multstep</td>
<td>OV ZE MI</td>
<td></td>
</tr>
<tr>
<td>0c</td>
<td>3</td>
<td>Y*Numptr</td>
<td>SUB OR COMP MVN Divstep</td>
<td>OV ZE MI</td>
<td></td>
</tr>
<tr>
<td>6g</td>
<td>3</td>
<td>Address</td>
<td>X*Addr MOD LIM MOD' PTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>LOAD</td>
<td>Y*Addr X = \phi(Y), first element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6F</td>
<td>3</td>
<td>STORE</td>
<td>Y*Addw \phi(Y) = X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>3</td>
<td>COPY</td>
<td>X = Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>3</td>
<td>TAG</td>
<td>Y*CFC (See text)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>3</td>
<td>CHL</td>
<td>X_{h} = 0; X_{l} = Y_{h}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>3</td>
<td>CLH</td>
<td>X_{h} = 0(Y_{1}(bits 4 to 15); X_{tag} = INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>3n</td>
<td>MARK</td>
<td>IP(m) = n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>3n</td>
<td>MOD</td>
<td>X*Addr X = X' n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>3n</td>
<td>UNDU</td>
<td>Unstack X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>3n</td>
<td>CASE</td>
<td>X*Int If 0&lt;X&lt;n, IP = IP + 2X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>3n</td>
<td>RETURN</td>
<td>Return to mark n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>3n</td>
<td>ADD</td>
<td>X*Int X = X + n_s</td>
<td>OV ZE MI</td>
<td></td>
</tr>
<tr>
<td>7A</td>
<td>3n</td>
<td>DUMP</td>
<td>Stack X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7B</td>
<td>3nc</td>
<td>SKIP</td>
<td>Assemb. If c, IP = IP + 2n + 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7C</td>
<td>3n</td>
<td>CALL</td>
<td>X*Ctl If n, Stack link and IP = X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7D</td>
<td>3n</td>
<td>GOTO</td>
<td>X*Ctl If n, IP = X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7E</td>
<td>3</td>
<td>REG</td>
<td>X*CRA Access control registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7F</td>
<td>3nc</td>
<td>MSYS</td>
<td>Call microsystem function</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \phi() = \text{'contents of'} \)
X' n = MOD on the pointer in X, by n

#### Instruction formats (h)

<table>
<thead>
<tr>
<th>f</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3n</td>
</tr>
<tr>
<td>2c</td>
<td>3nc</td>
</tr>
</tbody>
</table>

#### Operand classes

- Addr : BYTEv BYTRw HWDw HWDRw RESw RESRw MIXw MIXR
- Addw : BYTEv HWDw RESw MIX
- Numptr : INTEGERv BYTRw HWDw HWDR
- Int : INTEGER
argument is fetched from store if the Y register is an address pointer of the numeric type. However X must be an integer element.

**Addressing Functions** The MOD functions (f=2,6A,6C and 75) modify the 'f' field of the address pointer on which it acts, by adding the number specified (as an integer or as 'n' in the instruction). The function fails if the number specified exceeds the limit field. The MOD* version leaves the result in X0. The LIM function (f=6B) reduces the limit field by the number specified, but the resultant field is not allowed to become negative. The PTS (f=6D) function changes the 'tag' of the address pointer on which it acts by restricting it to 'read only' access or by reducing the element size in the case of numeric pointers (HWD to BYTE, or HWDR to BYTER). The action to be done is specified by the number code in the Y register.

**LOAD and STORE** (f=6E and 6F) In LOAD operations referring to a numeric element, the element is loaded into the least significant portion of the value field of an INTEGER element which is formed in X. The rest of the fields in X are set to zero. In non-numeric loads, the operand carries its own tag. In STORE operations, the destination segment restricts the value which can be stored as follows:

Destination Y BYTE or HWD requires source X INTEGER

do. Y RES do. X INTEGER or CAPABILITY or CWD or CWDR

do. Y MIX allows any register 'tag' to be stored.

Otherwise, a 'type' error causes function failure.

**TAG** (f=71) Here the Y register contains a CFC capability which contains with it, the new 'tag' of the element to be formed. The TAG function changes the tag of the element in the X register to the new
value. The CFC capability is found only in control segments and validity is determined by assembly check.

**CHL and CLH** *(f=72 and 73)* These instructions provide for transfer of the higher halfword of one register into the lower half of another or vice versa. In both cases, the new element formed has its 'tag' set to INTEGER.

**CALL, MARK and RETURN** *(f=7C, 7 4 and 7 8)* The MARK instruction sets the mark field in the IP to a new value. The CALL instruction stores the IP, together with its mark value, in the process stack to form part of a dynamic chain. The Stack pointer (SP) is prevented from unstacking a previously stored link in the chain. The RETURN instruction results in a scan of the dynamic chain in reverse order, obeying the first link with a mark value greater than that specified in the RETURN instruction. When this is found, the SP and IP are set to the value they had when this link was created.

**DUMP and UNDU** *(f=7 5, 7 6)* These instructions provide for stacking and unstacking an element between a register and the stack. The SP's 'Limit' and 'f' fields indicate the available area. The Mark pointer (MP) defines the extent of the whole process stack.*

**CASE** *(f=7 7)* This instruction provides efficient handling of multiway branches but control still remains within the control segment.

**REG** *(f=7E)* Since access to control registers is to be restricted the CRA capability is required as an operand. The 'b' field in the capability specifies the nature of access according to the code below:

- \( b = 0 \) \( \Rightarrow \) COPY EP to X
- \( b = 4 \) \( \Rightarrow \) COPY X to EP

* It should be noted that the Mark pointer is used differently in the modified version, as will be shown in chapter III.
b = 1 \Rightarrow \text{COPY SP to X} ; b = 5 \Rightarrow \text{COPY X to SP}

b = 2 \text{ do. MP do.} ; b = 6 \text{ do. MP}

b = 3 \text{ do. IP do.} ; b = 7 \text{ do. IP}

**Microsystem Functions** (f=7F) The set of functions implemented as microsystem functions are characterized by complexity and variability which prevent them from being included in the 'hardwired' set. They are implemented by providing a small control memory and a shift register stack for the IP (f,p fields). Execution of the instruction stacks the IP and branches to a control memory location dependent on the 'n' value in the instruction, after setting the privilege bit.

The first function in this class is the CONNECT function which associates a Name capability in the control segment with a pointer to the program base. Thus shared control segments can refer to a resource by a Name capability, thus providing a uniform basis for identifying resources.

The EVAL function (Fig. A.1) is needed to convert any code-word into the corresponding pointer. The 'b' field of the codeword indicates that the segment is defined by the 'b-th' entry in the Master Segment Table. The entry contains a 2 bit 't' field for synchronization purposes and a 14 bit word address of the segment. If the codeword is evaluated (EVALed) and the 't' field is zero then the pointer is formed and the 't' field set to one. However, if this field is already set, then the requesting process is suspended until the field is reset.

This is a simple but not secure means of synchronization as no check is made once a pointer is formed.

The RELEASE function removes the blocking condition on a segment to which the process has exclusive access. Waiting processes are freed.
If EVAL is applied on \texttt{Gwd b} then,

The new address pointer formed by the EVAL operation is constructed by loading the first element of the segment referred to, in the higher value (more significant) halfword. The address pointer \texttt{f} field is made to point to the next element.

\textit{Fig. A.1 The micro PN EVAL operation}
Appendix A.2

The modified Micro PN machine (MPN)

The basis for the hardware support is the modified micro PN machine (MPN). The specification of the original machine, as proposed by Iliffe [Il 75], is given in Appendix A.1. Here we discuss a slightly modified version of this machine, which forms the hardware base for the protection structure of chapter III. The modified MPN instruction set is specified in Table A.3. The modifications introduce new instructions which could have been implemented by the instruction set of the unmodified version. The accent now is on a base architecture, rather than a microprocessor (which was the rationale behind the original micro PN). As a result, the operations are regarded differently with respect to the validity of the operands, as will be shown. The tagged element format is assumed to be the same as in the original version (Table A.1). In chapter III, a new set of tagged elements will be introduced to support the protection structure extensions. The revised MPN instruction set is not minimal but provided for flexibility and ease of use.

Considering the modifications to the instruction set, we state that in Table A.3, the restrictions on operands are as indicated by the 'validity' column, while the nature of the operation is specified in the 'operation' column. The modified version includes a specification of SHIFT functions. The SHIFT instructions operate on the 16 bit value part of an INTEGER element and set the corresponding condition code indicators. In addition, a set of field manipulating instructions (4.5, 4.6, 4.7) have been specified. The X register specifies the offset in field units, from the leftmost field unit of the Y register. The
### Table A.3 MPN Instructions

<table>
<thead>
<tr>
<th>f, g</th>
<th>Mnem.</th>
<th>Validity</th>
<th>Operation</th>
<th>Cond. Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>GOTO</td>
<td>CURR.Lim</td>
<td>IP = IP + n_s</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>DATA</td>
<td>X0.Tag=Cap,X0.Typ=Nam,X0_L=φ(IP+n_s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2</td>
<td>MOD'</td>
<td>X≡Aptr</td>
<td>X0 = X'n</td>
<td></td>
</tr>
<tr>
<td>3 2d</td>
<td>IFGO</td>
<td>CURR.Fbl</td>
<td>If c, IP = IP - 2n + 2</td>
<td>OV, ZE, MI</td>
</tr>
<tr>
<td>4g 3</td>
<td>Shifts</td>
<td>X≡Int</td>
<td>Left, Right, Right Circular, Arith. left</td>
<td></td>
</tr>
<tr>
<td>0g 4</td>
<td>Move</td>
<td>X≡Int</td>
<td>MOVEF: 4 bit field, MOVEB: byte</td>
<td></td>
</tr>
<tr>
<td>5g 7</td>
<td>Field</td>
<td>MOVEH: half word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.83</td>
<td>TAG</td>
<td>Y≡CFC</td>
<td>X.Tag=Y(29:32), X.Typ = Y(25:28)</td>
<td></td>
</tr>
<tr>
<td>5 2</td>
<td>COPYN</td>
<td>X.Tag=Int, X(25:32) = n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6g 3</td>
<td>Arith</td>
<td>X≡Int</td>
<td>ADD, AND, MASK, NEQ, SUBTRACT, OR, MVN</td>
<td>OV, ZE, MI</td>
</tr>
<tr>
<td>0g 8g</td>
<td>Y≡Mptr</td>
<td>COMPARE, Multi. step, Division step,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.E3</td>
<td>LOAD</td>
<td>Y≡Mptr</td>
<td>X = g(Y), first element</td>
<td></td>
</tr>
<tr>
<td>6.E3</td>
<td>STORE</td>
<td>Y≡Mptw</td>
<td>g(Y) = X</td>
<td></td>
</tr>
<tr>
<td>7.03</td>
<td>COPY</td>
<td>X = Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.13</td>
<td>TTAG</td>
<td>X(29:32)=Y.Tag,X.Tag = Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.23</td>
<td>CHL</td>
<td>X_H=0,X_L=Y_H, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.33</td>
<td>CLH</td>
<td>X_H=0,Y_L, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.4n</td>
<td>MARK</td>
<td>IP.m = n,Mp = SP, SP= null pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.5n</td>
<td>MOD</td>
<td>X≡Aptr</td>
<td>X = X'n</td>
<td></td>
</tr>
<tr>
<td>7.6n</td>
<td>UNDU</td>
<td>SP</td>
<td>Unstack element into X</td>
<td></td>
</tr>
<tr>
<td>7.7n</td>
<td>CASE</td>
<td>X≡Int</td>
<td>If 0&lt;X&lt;n, IP = IP + 2n</td>
<td>OV, ZE, MI</td>
</tr>
<tr>
<td>7.83n</td>
<td>RETURN</td>
<td></td>
<td>Return control to mark n</td>
<td></td>
</tr>
<tr>
<td>7.93n</td>
<td>ADDI</td>
<td>X≡Int</td>
<td>X = X + n_s</td>
<td></td>
</tr>
<tr>
<td>7.A3n</td>
<td>Dump</td>
<td></td>
<td>Stack element from X</td>
<td></td>
</tr>
<tr>
<td>7.B3n</td>
<td>SKIP</td>
<td>CURR.Lim</td>
<td>If c, IP = IP + 2n + 2</td>
<td>OV, ZE, MI</td>
</tr>
<tr>
<td>7.C3n</td>
<td>CALL</td>
<td>X≡Cptr</td>
<td>If n, Stack link, IP = X</td>
<td></td>
</tr>
<tr>
<td>7.D3n</td>
<td>GOTO</td>
<td>X≡Cptr</td>
<td>If n, IP = X</td>
<td></td>
</tr>
<tr>
<td>7.E3</td>
<td>ITP</td>
<td>X(29:32) = Y.Typ, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.F3n</td>
<td>INDEX</td>
<td>X(21:32) = Y.Lim, X.Tag = Int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Formats: f, g, h

- **Operand Classes:**
  - Mptr = Byte, ByteHw, HwHw, HwR Res, Resr, Cptr
  - Mptw = Byte, HwHw
  - Nptr = Integer, Byte, ByteHw, HwHw, HwR, Res or Resr pointing to Integer
  - Int = Integer
  - Aptr = Mptr, Mix, Mixr
  - Cptr = Control pointer

#### Key:
- X.Tag = Tag of the X register
- X.Typ = Type field of the X register
- X.Lim = Limit field if the X register contains a pointer
- X(n1:n2) = Bits n1 to n2 of the X register
- g(X) = element pointed to by pointer in X register
- X'n = Fbl field of the pointer in X is modified by n
resulting field is placed in the least significant part of the X register and the X register TAG is set to INTEGER (Fig. A.2).

A set of interrogative instructions have been provided - ITAG, ITYP, INDEX (7,1,7,E,7,F), to interrogate the TAG, TYPE and LIMIT value of an element. The REG and Microsystem (7E,7F) instructions of the unmodified version are now included as part of the protection structure. The CALL, GOTOC, RETURN instructions provide for inter-segment transfers and subroutine linkage.

Chapter II had introduced the theoretical basis for a generalized Pointer Number machine. Appendix A.1 illustrates the salient features of a proposed machine and this section of the appendix describes a modified version of this machine. The discussion of chapter III assumes that these sections provide sufficient background to familiarize the reader with the MPN operations vital to the maintenance of tagged elements and transfer of control. The following comments provide some clarification with respect to the modified instruction set of Table A.3:

1) \( f = 0 \) The Instruction Pointer (IP) indicates the next instruction to be executed and unlike address pointers, does not describe a set of elements using a 'base limit' descriptor format (Table A.1), but refers to the next instruction only. So the processor has a CURR register which contains the 'base limit' description of the current control segment and has a TAG value of HWDR, as each instruction is halfword in size. The GOTO instruction is used for transfer of control within the current control segment, which is ensured by a check against the CURR register.

2) \( f = 2,6.A \) to \( 6.D,7,5 \) The descriptor manipulating functions can produce only a restricted version of the Address Pointer which is an
Move Operations

If the register $Y$ contains $\text{Int} u \ v \ n$ then:

- no. of hits 4 4 8 16

\[
\begin{align*}
&X \text{ Int} a \ p \ l \xrightarrow{\text{MOVE}} X \text{ Int} a \ p \ 0;0;0;u \\
&X \text{ Int} b \ q \ 0 \xrightarrow{\text{MOVEB}} X \text{ Int} b \ q \ 0;\text{Int} u \\
&X \text{ Int} c \ r \ l \xrightarrow{\text{MOVEH}} X \text{ Int} c \ r \ n
\end{align*}
\]

TAG operation

If the register $Y$ contains $\text{Cap} \text{Cfd} \ b \ Hwd \ L$

- no. of bits 4 4 8 4 12

\[
\begin{align*}
&X \text{ Int} \ F \xrightarrow{\text{TAG}} X \text{ Hwd} \ L \ F
\end{align*}
\]

Interrogative Operations

\[
\begin{align*}
&Y \text{ Cap} \ t \ b \xrightarrow{\text{ITAG}} X \text{ Int} \ 0 \ 0 \ 0;\text{Cap} \\
&Y \text{ Cwd} \ c \ b^* \xrightarrow{\text{ITYP}} X \text{ Int} \ 0 \ 0 \ 0;\ c \\
&Y \text{ Res} \ n \ f \xrightarrow{\text{INDEX}} X \text{ Int} \ 0;\ n
\end{align*}
\]

Fig. A.2 The Move, TAG and Interrogative Operations
operand. The MOD instruction increases the FBL value of the pointer by ‘n’ size units, but checks that the LIMIT field is not exceeded. The LIM instruction can only reduce the LIMIT size of a pointer.

3) $f=1$ The DATA instruction is completely changed from the micro PN version. This is because CONTROL segments are now going to be restricted to contain only Name capabilities* and so rather than containing the whole Name capability**, it just contains the ID part. The DATA instruction’s operand specifies the offset of this ID part from the instruction pointed to by IP, and the DATA instruction uses this ID part to form a Name capability in the XO register.

4) $f=3$ IFGO is provided for conditional backward jumps and the FBL field of the CURR register, which specifies the starting location of the control segment, ensures that control remains within the current segment.

5) $f=4.8$ A capability allowing formation of a new tagged element is necessary to ensure the validity of the TAG operation. With reference to the element format of Table A.1, the assumption is that the CFC capability occupies the high order halfword of the register, while the lower halfword contains the new TAG, TYPE required. With the revised format for tagged elements, necessitated by the protection structure (Table 3.2), all elements are a full word in length and hence the CFC contains the new TAG, TYPE in its low order 16 bits.

6) $f=7.4$ The MARK instruction updates the mark value in the IP. It also sets the Mark pointer (MP) to the value in the Stack pointer (SP) and then sets the SP to null.

---

* The micro PN machine allowed CFC and CRA capabilities also.
** The revised tagged element format calls for capabilities of 32 bits in width. Consequently the Name capability has a 16 bit ID field.
7) On executing CALL, the IP (instruction pointer) is stored as a link in the location pointed to by the MP and control is then transferred.

8) The RETURN instruction necessitates a scan of the stack for a link with a mark value greater than or equal to 'n'. The IP is then updated, the SP is made to define the appropriate portion of the stack, while the MP is made to point to the previous stack frame. All these operations are elaborated in section III.3 dealing with the stack operation.

9) GOTOC provides an inter segment transfer of control, but still within the executing domain.
Appendix A.3

Analysis of the MPN machine

In the last section, we have described a machine which provides a restricted mapping between the set of instructions and the set of operands (tagged elements). It is essential to differentiate between the various classes of operations possible on the tagged elements. The term 'operation' is used when it is necessary to stress the transformation of an operand by the application of an instruction. An instruction is the static definition of an operation, the operation being the execution of the instruction. The classification of MPN instructions in Table A.4 exemplifies the basis for the classification of instructions in any generalized Pointer Number machine.

Class A operations are the set of arithmetic operations, which provide the functional mapping between sets of INTEGER elements and are as defined in conventional architectures. Class B operations provide a transfer of control by altering the Instruction Pointer. Class C operations are tag independent operations and provide for copying the whole or part of an element between registers and the memory or just between registers. Class D operations modify descriptor elements.

In the context of the MPN machine, we will now arrive at a criterion for protection. The following conventions are established:

1) An element is considered to be accessed, if it exists in one of the processor registers or if it can be transferred from memory to a processor register.

2) An element can be transferred from memory to a processor register, only if it is among the set of elements defined by a POINTER in
### Table A.4 Classification of MPN Instructions

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mnemonic</td>
<td>f.g</td>
<td>Mnemonic</td>
<td>f.g</td>
</tr>
<tr>
<td>Shifts 4.0-4.4</td>
<td>B₁ GOTO 0</td>
<td>C₁ COPYN 5</td>
<td>D₁ MODₙ 2</td>
</tr>
<tr>
<td>Arith 6.0-6.9</td>
<td>CASE 7.7</td>
<td>COPY 7.0</td>
<td>MOD 6.A</td>
</tr>
<tr>
<td></td>
<td>B₂ IFGO 3</td>
<td>CLH 7.3</td>
<td>LIM 6.C</td>
</tr>
<tr>
<td></td>
<td>B₃ CALL 7.C</td>
<td>Move 4.6-4.8</td>
<td>MODₙ 7.5</td>
</tr>
<tr>
<td></td>
<td>GOTO 7.D</td>
<td>UNDU 7.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B₄ MARK 7.4</td>
<td>DUMP 7.A</td>
<td>D₂ PTS 6.D</td>
</tr>
<tr>
<td></td>
<td>B₅ DATA 1</td>
<td>C₂ LOAD 6.E</td>
<td>D₃ ITAG 7.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C₃ STORE 6.F</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INDEX 7.F</td>
</tr>
</tbody>
</table>

TAG is considered as a Class E operation.
RETURN is unclassified for present purposes.

<table>
<thead>
<tr>
<th>Class A Operations</th>
<th>apply on INT&amp;NUMPTR and result in INT in reg. X</th>
</tr>
</thead>
<tbody>
<tr>
<td>do. C₁</td>
<td>do. ANY element</td>
</tr>
<tr>
<td>do. C₂</td>
<td>do. MPTR</td>
</tr>
<tr>
<td>do. C₃</td>
<td>do. MPTW</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>do. D₁</td>
<td>do. APTR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>do. D₂</td>
<td>do. APTR</td>
</tr>
<tr>
<td>do. D₃</td>
<td>do. ANY element</td>
</tr>
<tr>
<td>do. B₁</td>
<td>apply on INT&amp;n and change IP with restriction defined by CURR.Lim</td>
</tr>
<tr>
<td>do. B₂</td>
<td>do. n</td>
</tr>
<tr>
<td>do. B₃</td>
<td>do. CPTR&amp;INT</td>
</tr>
<tr>
<td>do. B₄</td>
<td>do. n</td>
</tr>
<tr>
<td>do. B₅</td>
<td>do. n and form in XO, a Name capability</td>
</tr>
<tr>
<td>do. E</td>
<td>do. ANY&amp;CFC to form new TAG,TYPE in X as allowed by the CFC</td>
</tr>
</tbody>
</table>
one of the processor registers.

3) Pointers can only exist in the processor registers, control registers or the processor stack (but not in memory).

4) Transfer of control is confined to the current control segment (defined by the CURR register) or to a control segment for which a control Pointer exists in the space allowed for pointers.

5) The set of all integer values is considered to be always accessible as an INTEGER element corresponding to any value can be generated.

6) The restricted version of a Pointer is a pointer which defines a subset of the set of elements defined by the original pointer.

The **DOMAIN** of a process is then defined to be the set of all elements which can be accessed and the set of all control segments to which control can be transferred by the process. Protection is then defined as the constraints maintained with respect to the transformation of operands by operations, such that the domain of the process does not **increase**. Violation of protection is then the formation of a pointer which enables access to an element not present in the domain, or by which control is transferred to a control segment not defined within that domain. The most important aspect of the protection system is the ability to verify that an address generated by a program lies within the access domain defined in the context of a particular process.

The rigorous definition of protection enables us to show that the various MPN instruction classes do not violate protection. Class A operations (Table A.4) produce INTEGER elements from INTEGER elements, thus ensuring that the results are defined within the domain.

Class $B_1, B_2$ operations transfer control within the current
control segment. Class B₃ operations transfer control to a control
segment which is defined within the domain as a Control pointer describes it. Class B₄ operations modify pointers to the stack but are
still constrained by the PSVR register defining the Process State
Vector. The class B⁵ operation forms a Name capability which exists
in the control segment without restriction as a process can use any
Name capability as an identifier. Thus no class B operation violates
protection.

Class C₁ operations can only produce the exact copy of an
existing element or produce an INTEGER element. Class C₂ operations
transfer an element from memory and since the memory cannot contain
pointers, protection is not violated. Class C₃ operations store an
element in memory but subject to the constraints of the pointer indicating the memory location. The constraints dictated by MPTR and MPTW
(Table A.3) allows the check that a pointer cannot exist in memory,
as a segment containing a pointer can only be defined by a MIXED poi-
neter (MIX,MIXR).

Class D operations are the pointer manipulating functions
and produce a pointer as the result of an operation. Class D₃ operations produce an INTEGER element and hence cause no violation. Class D₄ operations produce a restricted version of an existing pointer and thus
the domain cannot increase. The restricted version is ensured by subj-
ecting the operations to the constraints - for the MOD operations, the
'MOD value' is to be less than the 'Limit' field of the pointer operand and for LIM operations the 'Limit' field of the operand can only be
reduced. Fig. A.3 clarifies the constraints on this class of operations.
Finally, class D₂ operations produce a more restricted pointer type
Class \( D_1 \) Operations

Initially \( X \) 

\[
\text{COPY } Y_1, X
\]

\[
\text{MOD } Y_1, q \quad (\text{fail if } p < q)
\]

results in \( Y_1 \)

\[
\text{COPY } Y_2, Y_1
\]

\[
\text{LIM } Y_2, r \quad (\text{fail if } p - q < r)
\]

results in \( Y_2 \)

Both MOD and LIM contract the space defined by a pointer.

Class \( D_2 \) Operation - PTS

PTS allows the change of the TAG by forming a restricted pointer TYPE. The integer in \( Y \) indicates the conversion.

<table>
<thead>
<tr>
<th>Conversion</th>
<th>( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.Tag :       BYTE BYTR 1</td>
<td>In all cases either the 'read' option is removed or the Numeric element size is reduced.</td>
</tr>
<tr>
<td>HWD HWDR 2</td>
<td></td>
</tr>
<tr>
<td>RES RESR 3</td>
<td></td>
</tr>
<tr>
<td>MIX MIXR 4</td>
<td></td>
</tr>
<tr>
<td>HWD BYTE 5</td>
<td></td>
</tr>
<tr>
<td>HWDR BYTR 6</td>
<td></td>
</tr>
</tbody>
</table>

Fig. A.3 Class \( D_1 \) and \( D_2 \) Operations
which does not increase the address space (Fig. A.3). Thus it is shown that the class of MPN instructions (A to D) do not violate protection and hence form a secure set.

The class E function, TAG is the only instruction capable of violating protection. But here, the protection is implemented by constraining its use with a CFC capability, which defines the TAG, TYPE of the new element which can be formed. Thus, by controlling the generation of this CFC capability, protection is enforced. The MPN machine assumes that pointers are already existing and maintains the security of the domain defined as the accessible space with respect to the pointers. In the protection structure extension of Chapter III, the concept of domain will be expanded to include the access space defined by resource descriptors like codewords and capabilities. An additional set of operations will be defined and the domain will then correspond to the elements which can be accessed by executing any sequence of operations from those available.

Table A.5 indicates the allowed operations on the various classes of tagged elements, in the context of the MPN machine. This is an inverse mapping of Table A.4. The result of operations which are not affected by the original element in the target register have also been indicated.
### Table A.5 Allowed Operations on Elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Mode</th>
<th>Operation</th>
<th>Class</th>
<th>Is the reg. changed?</th>
<th>If so the new element is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>Unary</td>
<td>Shift</td>
<td>A</td>
<td>Yes</td>
<td>INT</td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>ADDI</td>
<td>A</td>
<td>Yes</td>
<td>INT</td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>CASE</td>
<td>B₁</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td>Arith</td>
<td>A</td>
<td>Yes</td>
<td>INT</td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>MOD,LIM,MOD'</td>
<td>D₁</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>PTS</td>
<td>D₂</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>Move</td>
<td>C₁</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>APTR</td>
<td>Unary</td>
<td>MODₙ,MODₙ'</td>
<td>D₁</td>
<td>Yes (or XO)</td>
<td>Restricted (APTR)</td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td>MOD,LIM,MOD'</td>
<td>D₁</td>
<td>Yes (or XO)</td>
<td>Restricted (APTR)</td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>PTS</td>
<td>D₂</td>
<td>Yes</td>
<td>Restd. Type (APTR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ITAG,ITYP,INDEX</td>
<td>D₃</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>MPTR</td>
<td>Unary</td>
<td>LOAD</td>
<td>C₂</td>
<td>No (other reg. updated from)</td>
<td></td>
</tr>
<tr>
<td>MPTW</td>
<td>do.</td>
<td>STORE</td>
<td>C₃</td>
<td>No (memory changed with constraints)</td>
<td></td>
</tr>
<tr>
<td>NUMPTR</td>
<td>Binary</td>
<td>MOD,LIM,MOD',PTS</td>
<td>D₁</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>Arith</td>
<td>A</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>CPTR</td>
<td>Unary</td>
<td>CALL</td>
<td>B₃</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO</td>
<td>B₃</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ANY</td>
<td>Unary</td>
<td>COPY</td>
<td>C₁</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>CLH,CHL</td>
<td>C₁</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>Move</td>
<td>C₁</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>do.</td>
<td>DUMP</td>
<td>C₁</td>
<td>No (copy of element is stacked)</td>
<td></td>
</tr>
</tbody>
</table>

Registers changed when used as the target register for an operation on an element in another register:

- **XO** ← Restricted (APTR) by MOD',LIM'
- **XO** ← INT by COPYN
- **X** ← element from stack by UNDU
- **X** ← copy of another element by COPY
- **X** ← INT by CLH,CHL,ITAG,ITYP,INDEX and Move operations
- **X** ← Any resource by LOAD (except pointers)

- CP changed by GOTO, IFGO
- CASE, SKIP
- MARK
- CALL, GOTOC
- SP changed by MARK
- DUMP, UNDU
- MP changed by MARK
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aptr</td>
<td>Address Pointer</td>
</tr>
<tr>
<td>Bas</td>
<td>base capability</td>
</tr>
<tr>
<td>BFC</td>
<td>base forming capability</td>
</tr>
<tr>
<td>BF</td>
<td>Base pointer</td>
</tr>
<tr>
<td>Byte</td>
<td>byte address pointer</td>
</tr>
<tr>
<td>Byter(or Bytr)</td>
<td>do. (read only)</td>
</tr>
<tr>
<td>Cap</td>
<td>capability</td>
</tr>
<tr>
<td>CFC</td>
<td>capability forming cap.</td>
</tr>
<tr>
<td>Cnt</td>
<td>control segment class</td>
</tr>
<tr>
<td>Cptr</td>
<td>control pointer</td>
</tr>
<tr>
<td>CRA</td>
<td>control register access</td>
</tr>
<tr>
<td>CURR</td>
<td>reg. defining current control seg.</td>
</tr>
<tr>
<td>Cwd</td>
<td>codeword</td>
</tr>
<tr>
<td>Dev</td>
<td>device capability</td>
</tr>
<tr>
<td>Ecap</td>
<td>extended capability</td>
</tr>
<tr>
<td>EFC</td>
<td>extension forming cap.</td>
</tr>
<tr>
<td>EFN</td>
<td>extended pointer number</td>
</tr>
<tr>
<td>Fbl</td>
<td>first byte location</td>
</tr>
<tr>
<td>Gpn</td>
<td>generalized pointer number</td>
</tr>
<tr>
<td>GPR</td>
<td>general purpose register</td>
</tr>
<tr>
<td>Hwd</td>
<td>halfword address pointer</td>
</tr>
<tr>
<td>Hwdr</td>
<td>do. (read only)</td>
</tr>
<tr>
<td>ID</td>
<td>identity field</td>
</tr>
<tr>
<td>IP</td>
<td>Instruction pointer</td>
</tr>
<tr>
<td>Inc</td>
<td>increment capability</td>
</tr>
<tr>
<td>Inst</td>
<td>instruction register</td>
</tr>
<tr>
<td>Int</td>
<td>integer element</td>
</tr>
<tr>
<td>I/O</td>
<td>input/output</td>
</tr>
<tr>
<td>Mix</td>
<td>mixed segment pointer</td>
</tr>
<tr>
<td>Mixr</td>
<td>do. (read only)</td>
</tr>
<tr>
<td>MP</td>
<td>Mark pointer</td>
</tr>
<tr>
<td>MPN</td>
<td>modified micro PN</td>
</tr>
<tr>
<td>Mptr</td>
<td>all address pointers except mixed pointers</td>
</tr>
<tr>
<td>Mptw</td>
<td>all Mptrs with 'read access'</td>
</tr>
<tr>
<td>Nam</td>
<td>name capability</td>
</tr>
<tr>
<td>Nptr</td>
<td>numeric pointer</td>
</tr>
<tr>
<td>Nul</td>
<td>null capability</td>
</tr>
<tr>
<td>Num</td>
<td>numeric segment class</td>
</tr>
<tr>
<td>PI</td>
<td>process identity register</td>
</tr>
<tr>
<td>PFC</td>
<td>process forming capability</td>
</tr>
<tr>
<td>Pro</td>
<td>process capability</td>
</tr>
<tr>
<td>PSV</td>
<td>process state vector</td>
</tr>
<tr>
<td>PSVR</td>
<td>do. register</td>
</tr>
<tr>
<td>Rcap</td>
<td>revokable capability</td>
</tr>
<tr>
<td>Rcax</td>
<td>revokable extended capability</td>
</tr>
<tr>
<td>Rcod</td>
<td>revokable codeword</td>
</tr>
<tr>
<td>Res</td>
<td>resource address pointer</td>
</tr>
<tr>
<td>Resr</td>
<td>do. (read only)</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>Str</td>
<td>struct. segment class</td>
</tr>
<tr>
<td>WBLO</td>
<td>wait on block lockout</td>
</tr>
<tr>
<td>WEVN</td>
<td>wait on event variable block</td>
</tr>
<tr>
<td>WSEM</td>
<td>wait on semaphore block</td>
</tr>
</tbody>
</table>
Glossary

base: a list of resources which a process can access. Every process executes with respect to a base.

base limit format: scheme of describing a set of elements by specifying the starting location of the first element and the index of the last with respect to the first.

capability: controls access to all resources except storage.

codeword: controls access to storage.

descriptor: an element which describes a set of elements.

domain: the dynamic version of a base, defined as the total accessible space available to a process by performing sequences of 'operations'.

event queue: synchronization mechanism.

execution environment: see 'domain'.

hierarchical structure: organization into components, the interrelationship between which is defined by a partial ordering relation.

kernel: see 'nucleus'.

limit: the index of the last element when using the 'base limit' format.

lock bit: bit used in revokable versions to localize the effect of revocation.

mechanisms: the set of features common to all operating systems.

message buffer: synchronization mechanism.

micro PN: a practical instance of a descriptor oriented architecture, a modified version of which serves as the hardware support.

monitor: the set of features which are constrained to be used exclusively by the processes.
mutual exclusion: constraint on the simultaneity of execution points between two processes.

nucleus: the collection of features which form the environment for the definition of specific operating systems as cooperating concurrent processes.

object: the entity (or resource) whose access is controlled by a cap.

options: specification of the allowed operations on an 'object' controlled by a capability.

pointer: a 'descriptor' which directly refers to storage.

policies: the features which distinguish specific operating systems and which make use of the 'mechanisms' to do this.

privileges: see 'options'.

process: defined as an independently schedulable unit which represents a user active in the computer utility.

queue D structure: a queue structure which inserts and removes two elements at a time.

reference: indicates the localizing information by which the object, referred to by a capability, can be located.

revocation: reduction of the options in a capability.

segment: a logical unit of information.

semaphore: synchronization mechanism.
References


[Il 75] Iliffe, J.K., "A secure and efficient form of microcode," Class notes of course on 'High Level Computer Design', Sec. 2.5, Department of Electrical Engineering, Rice University, Houston, Texas.


