INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
JAGANNATHAN, ANAND
THE STRUCTURED DESIGN OF A SOFTWARE DIRECTED
ARCHITECTURE.

RICE UNIVERSITY, PH.D., 1979
RICE UNIVERSITY

THE STRUCTURED DESIGN OF A SOFTWARE DIRECTED ARCHITECTURE

by

ANAND JAGANNATHAN

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

DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

Edward Alvin Feustel
Dr. Edward A. Feustel (Associate Professor)
Chairman

James B. Sinclair
Dr. James B. Sinclair (Assistant Professor)

Meera M. Blattner
Dr. Meera Blattner (Assistant Professor)

HOUSTON, TEXAS

MAY, 1979
ABSTRACT

THE STRUCTURED DESIGN OF A SOFTWARE DIRECTED ARCHITECTURE

by

Anand Jagannathan

This thesis presents a model of a computer architecture in which the design is strongly influenced by software considerations. Such an architecture is called software directed. In the proposed architecture, there are two ways in which software influences the design. First, the architecture supports the design methodology of 'data abstraction'. This methodology is becoming increasingly prevalent in software design. Second, the architecture is modeled such that any software subsystem can be implemented as part of the architecture. This is called 'incorporating' the subsystems within the architecture. We distinguish between the implementation of a subsystem as software and its implementation as part of the architecture.

The thesis describes a set of features for the use of 'abstraction' in a programming language. This description is formalized by means of a linguistic model. The model emphasizes generalized and dynamic features for defining 'parameterized abstractions'. From the design of the mechanisms supporting these features, we establish a set of requirements with respect to the architecture.

We use two operating system subsystems—the virtual memory and virtual processor—as examples of incorporation. Our approach to the problem is based on internally structuring the two subsystems. The technique used for structuring is called 'type extension'.

The benefit of structuring the virtual memory subsystem is that we are able to decompose it into small and easily understood components. The rationale for this fine level structuring is that each of these com-
ponents can be designed and implemented independently. Many of the compo-
ponents are simple enough to be directly implemented as 'hardware'.

In the structuring of the virtual processor subsystem, there are two
issues concerning the role of the components of the structure. The first
issue concerns the interactions with the components of the virtual memory
subsystem. The second issue concerns the role of the components in the
internal implementation of the architecture.

The entire internal structure of the architecture is based on the
use of 'type extension'. The components of the structure are small and
simple. The crux of our approach to implementing a subsystem within the
architecture is to associate an interpreter with, and choose one of sev-
eral forms of implementation for, each component of the subsystem. In a
conventional 'software' implementation, the components are restricted to
use the 'hardware architecture' as the sole interpreter for implementa-
tion programs. The important result of this research is that we identi-
fy and characterize several forms of implementation within the architec-
ture. The conventional forms of implementation are only a subset of these.

To summarize, we have designed an architecture which supports 'data
abstraction'. Using this design as a case study, we have presented an
architecture model which shows how a software subsystem can be implemen-
ted as part of the architecture. The significance of our approach is
that it bridges the gap between architecture design and software design.
The impact is on the total system design, which can now benefit at all
levels from an evolving hardware technology.
ACKNOWLEDGEMENTS

I wish to express my appreciation and gratitude to my thesis advisor, Dr. Edward A. Feustel, for his supervision and guidance. I wish to thank Dr. J.B. Sinclair and Dr. M. Blattner for serving on my thesis committee.

My study here at Rice University has been pleasant and rewarding thanks to my colleagues and friends. My special thanks to my fellow computer scientists, Daniel Dias, Bart Sinclair, William Bain, Timothy Gonsalves, Naresh Jotwani and Sudhir Ahuja for an enriching experience. I extend my thanks to my ever-helpful 'desi' friends, the 'Milam' and 'Ashby' gangs in particular. My thanks also goes to Joe Gonsalves, my roommate of long standing.

The individual most crucial to the final production of this thesis is Mrs. Pat Angel. There can be no dearth of tributes commending her for her diligence and expertise in typing the final draft of the thesis.

Finally, I express my sincere and wholehearted gratitude to my parents for their support and never-failing confidence in me. I also admire the courage that my mother and sister have shown, under most trying circumstances, in the past four years. This thesis is dedicated to my parents.

This research was supported by the National Science Foundation under the grant NSF-MCS-72-03609 A02.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>I.1 Structuring of Operating Systems</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>I.2 Linguistic Abstraction and Type Extension</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>I.3 Incorporating Software Subsystems</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>I.4 Plan of the Thesis</td>
<td>11</td>
</tr>
<tr>
<td>II</td>
<td>Architectural Features to Support Linguistic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>II.1 Introduction</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>II.2 The Generalized Linguistic Model</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>II.3 The Design of the Abstraction Mechanisms</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>II.4 The Required Architectural Features</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>II.5 Summary</td>
<td>53</td>
</tr>
<tr>
<td>III</td>
<td>The Virtual Memory Design</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>III.1 Introduction</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>III.2 The Use of Type Extension in Internal Structuring</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>III.3 Characterizations of Virtual Memory</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>III.4 The Paging Abstraction (VM2)</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>III.5 The Address Space Abstraction (VM4)</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>III.6 The Final Abstractions of the Virtual Memory</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>III.7 Summary</td>
<td>125</td>
</tr>
<tr>
<td>IV</td>
<td>The Virtual Processor Design</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>IV.1 Introduction</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>IV.2 The Virtual Processor Mainstream</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>IV.3 The Physical Processor Abstraction (VP1)</td>
<td>134</td>
</tr>
<tr>
<td>IV.4</td>
<td>The Base Processor Abstraction (VP2)</td>
<td>138</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>IV.5</td>
<td>The Message Communication Abstractions</td>
<td>146</td>
</tr>
<tr>
<td>IV.6</td>
<td>The Interpretive Role of the VP Abstractions</td>
<td>151</td>
</tr>
<tr>
<td>IV.7</td>
<td>The STOP Processor Operation</td>
<td>158</td>
</tr>
<tr>
<td>IV.8</td>
<td>The System Processor Abstraction (VP3)</td>
<td>165</td>
</tr>
<tr>
<td>IV.9</td>
<td>Summary</td>
<td>168</td>
</tr>
<tr>
<td>Chapter V.</td>
<td>The Architecture Interface</td>
<td>171</td>
</tr>
<tr>
<td>V.1</td>
<td>Introduction</td>
<td>171</td>
</tr>
<tr>
<td>V.2</td>
<td>The Internal Structure of the Architecture</td>
<td>176</td>
</tr>
<tr>
<td>V.3</td>
<td>The Meta-Implementation</td>
<td>181</td>
</tr>
<tr>
<td>V.4</td>
<td>The Operations at the Architecture Interface</td>
<td>197</td>
</tr>
<tr>
<td>V.5</td>
<td>The Addressing Structure of the Architecture</td>
<td>228</td>
</tr>
<tr>
<td>V.6</td>
<td>The Abstraction Features at the Architecture Interface</td>
<td>239</td>
</tr>
<tr>
<td>V.7</td>
<td>Summary</td>
<td>241</td>
</tr>
<tr>
<td>Chapter VI.</td>
<td>Summary and Conclusions</td>
<td>244</td>
</tr>
<tr>
<td>VI.1</td>
<td>Summary</td>
<td>244</td>
</tr>
<tr>
<td>VI.2</td>
<td>Results</td>
<td>247</td>
</tr>
<tr>
<td>VI.3</td>
<td>Future Research</td>
<td>252</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>255</td>
</tr>
<tr>
<td>Appendix 1</td>
<td></td>
<td>258</td>
</tr>
<tr>
<td>Appendix 2</td>
<td></td>
<td>269</td>
</tr>
<tr>
<td>Appendix 3</td>
<td></td>
<td>336</td>
</tr>
<tr>
<td>Appendix 4</td>
<td></td>
<td>346</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

This thesis presents a model of a computer architecture in which the design is strongly influenced by software considerations. Traditionally, the design of computer architecture is considered to be independent of the design of the software that it supports. In the software of today, the levels of design—problem solution, programming language, operating system—are being unified with respect to a common design methodology. Thus, languages which support and encourage the use of a particular programming methodology for problem solution are being designed. We extend this unifying methodology to the level of the architecture. This is the first way by which software influences the design of our architecture.

The conventional notion of a computer architecture is that of a system which interprets software programs and which is implemented in hardware. We also take the view that an architecture serves as an interpreter for software programs. However, we do not associate a particular form of implementation with the term 'architecture'.

The progress of current hardware technology is causing the cost of hardware to decrease. Therefore, software is becoming relatively more expensive. This suggests that subsystems which are usually implemented in software can now be implemented in hardware in a cost-effective manner. The problem is that these subsystems are large and unstructured so that implementing them in hardware becomes extremely difficult. Thus, the decrease in the cost of a hardware implementation is more than offset by the cost of converting from a software to a hardware implementation. Therefore, a systematic approach to this conversion is needed.
In this research, we consider the conversion in implementation of two subsystems which belong to the operating system. These two subsystems compose a part of the operating system which interfaces with the architecture. We treat the implementation of these subsystems as part of the implementation of the architecture. The key idea is that we do not presuppose that because these subsystems are part of the architecture, their implementation has to be by means of hardware. On the other hand, we recognize different forms of implementation within the architecture. The conventional notion of a 'hardware' implementation is only one of these forms. We refer to these subsystems as being incorporated within the architecture.

The approach that we use to incorporate the two subsystems is based on structuring the subsystems internally. This structuring is such that the components of the structure are small and simple enough that they can be easily designed and implemented. We model the architecture so that the implementation of these components can take any of the forms of implementation, identified within the architecture. Thus, the second influence of software on the design of our architecture arises because two 'software' subsystems are now implemented within the architecture.

The architecture that we have designed incorporates a virtual memory and virtual processor subsystem. The two subsystems are non-trivial in nature and compare in functionality with the subsystems in any large multi-user operating system supporting a virtual memory. The technique used for structuring these two subsystems is called 'type extension'. This technique has been used in the structuring of some practical operating systems [Sch 77, Neu 77, LaS 76, Wul 74]. More important, this technique is increasingly becoming popular as the basis for formal specification and
verification of software systems [Neu 77, Flo 77]. Therefore, the experience gained from these systems can be applied to the problem of incorporating software subsystems within the architecture.

The software design methodology which is extended to the level of the architecture is that of 'abstraction' [Wul 76, Lis 77]. Specifically, our architecture presents an interface which supports the use of this methodology in the design of software systems implemented on the architecture. A significant feature of this architecture interface is that it defines a secure and protected environment [SaS 75] for user interactions. This feature is based on the use of a 'capability' mechanism [DeV 66, Lin 76].

Since our architecture is influenced by two software considerations—supporting the design methodology of abstraction and incorporating the virtual memory and virtual processor subsystems—we call it a software directed architecture. This research shows that it is feasible to design such a software directed architecture. The significance of this design is that it bridges the gap between architecture design and software design so that the total system design can benefit from the evolution of hardware technology.

A major part of this research effort was directed to the structuring of the virtual memory subsystem and to a lesser extent, the virtual processor subsystem. Structuring of these subsystems is a subset of the more universal problem—structuring an operating system. In the next section (I.1) we will review the evolution of structured operating systems. The structuring technique that we used—type extension—is a formalism for the programming methodology of 'data abstraction'. In Section I.2, we will discuss 'abstraction' as supported by programming languages,
and its offshoot in system design, type extension. In Section 1.3, we will review different approaches to the problem of incorporating software subsystems within the architecture. Finally, in the last section, we will present a plan of the thesis.

Section 1.1  Structuring of Operating Systems

The purpose of structuring an operating system is to decompose the system into a set of simpler components. There is a three-fold motivation for this decomposition:

1) The complexity of the entire system can now be resolved in terms of the complexity of the components.
2) The decomposition facilitates the orderly verification of the system through the verification of the components.
3) Any violations of a 'clean cut' design can be detected by such a structuring. Essentially, such a design avoids any mutual dependency between components.

The basis for the first motivation is that operating systems are becoming increasingly complex. Hence, the complexity of these systems is exceeding the human capacity to understand them. The second motivation arises because of the same complexity issue. Since operating systems are increasing in size, there is a need for a formal technique to detect any errors in design or implementation. The use of automated tools for verification, aids in this process. Finally, 'clean cut' design requires that components of the system do not depend on the use of each other's facilities. The primary reason for this requirement is to avoid implementing special case mechanisms within these components. Moreover, an orderly verification is not possible if this mutual dependency occurs. The most obvious example of such a mutual dependency is
that between the virtual memory and virtual processor subsystems. We will later discuss our approach to the solution of this problem.

One of the earliest approaches to structuring an operating system was in the design of the T.H.E. multiprogramming system [Dij 68]. The system was decomposed into a linear ordering of layers. Each layer refined from the interface presented by its lower layer. The lowest layer was the hardware machine. The basis for this approach is that each layer only marginally changes from its lower layer and, hence, the component performing the change is small and easily understandable. The correctness of the operating system was verified on the basis of verifying each layer assuming that the lower layers are verified as correct. The layered approach to structuring an operating system has been used in the design of an operating system at Carnegie Mellon [Pri 73] and in the design of a family of operating systems [Par 76b].

The inherent defect in the layered technique of structuring is that the components of the structure have to be totally ordered. An alternative approach to structuring is to decompose the system into a set of partially ordered components. The technique used for this purpose is type extension. We will define type extension and review its implications in the next section. For now, we need to stress that type extension has been used as a structuring technique only at the higher levels of the operating system. The lower levels of the operating system are either unstructured or use the layered technique.

The HYDRA system [Wul 74] defined a 'kernel' of the operating system as a set of functions of 'absolute reliability and universal applicability'. The kernel implemented the mechanism for type extension, which was then used to structure the rest of the system. The basis for the
kernel approach is that the kernel is small enough so that it can be verified directly. This implies that the implementation of the type extension mechanism can be verified. However, this approach is deficient when used as the basis for structuring a large general purpose operating system. This is because the flexibility of the type extension mechanism, which is required, causes the kernel to include a sizeable part of the operating system.

The CAL system [LaS 76] and the SRI system [Neu 77] are examples of operating systems which conform to a layered structuring at the lower levels, but use type extension as the structuring technique for the higher levels. The basic reason why type extension cannot be used as the structuring mechanism at the lower levels is because its implementation requires the use of the very subsystems which need to be structured. Janson was the first to propose a new viewpoint of type extension which can be used to structure these lower level components [Jan 76]. We will discuss this issue in detail in Chapter III.

A third approach to structuring an operating system is the use of 'processes' as the components of the structure. Each process is identified as a separate stream of execution. The lowest levels of the operating system defines a 'nucleus' which implements the mechanisms for this form of structuring. This approach has been used in the design and implementation of a practical operating system [Han 73]. Variants of this approach have been used in designing virtual memory and virtual processor subsystems [Sax 75, Hub 76]. The T.H.E. system [Dij 68] actually combines this approach with that of a layered design.

Section 1.2 Linguistic Abstraction and Type Extension

The key problem in the design and implementation of large software
systems is the reduction of the complexity that must be dealt with at any one time. 'Abstraction' is one way to achieve this. Abstraction is the retention of only the essential properties of some entity that we are interested in with the corollary neglect of inessential details. Towards this end, procedural abstractions were introduced into programming languages. These abstractions hide the details of control flow and control structures. Data abstraction is the natural evolution of this concept on the basis of the same 'information hiding principle' [Par 72]. Data abstraction was introduced to hide the details of data flow and data structures. Linguistic abstraction is the support of both forms of abstraction by providing features in the programming language.

SIMULA 67 [DaH 72] is an important forerunner of programming languages which support data abstraction. Essentially, the support of data abstraction involves a construct by which a data structure can be associated with a set of procedures. These procedures are defined to access the data structure. A strict mechanism for data abstraction enforces the constraint that the data structure can only be accessed through calling these procedures. SIMULA did not enforce this constraint. Subsequent programming languages which provide sophisticated mechanisms for abstraction do enforce this constraint. Examples of such programming languages are CLU [Lis 77], ALPHARD [Wul 76] and MESA [Ges 77].

Based on the use of procedural abstractions, Morris [Mor 73] introduced the idea of protection in programming languages. Protection was enforced through the control of accessibility to procedure abstractions. A development of protection within a programming language has also been introduced using data abstractions [JoL 76]. The basis is the control of accessibility to data abstractions. The notion of 'access correct-
ness' was developed in this context.

Data abstraction laid the basis for the use of type extension in the context of system design. The underlying concept in such systems is the recognition of 'objects' and the definition of the system in terms of these objects. Such a system is called an object-based system. In an object-based system, all information is logically encapsulated within objects. The information within an object can be maintained by the system, as in the case of resources, or by the user, as in the case of data. Each object is characterized by a set of operations defined on it. The information within an object can only be accessed (i.e., read or modified) by invoking one of these operations.

A type is defined as a class of objects which have the same set of operations defined for all objects in the class. Type extension is the technique of defining new types of objects using existing types of objects. A set of types in the system are called primitive types. All other types in the system are extended, either directly or indirectly, from these primitive types. Defining an extended type in such a system consists of defining a type manager for the type. This type manager contains a sample representation for objects in that type and a set of procedures which implement the operations of the type.

The basis for the use of type extension as a technique for structuring is that a system can be decomposed into a set of type managers, where each type manager defines a type of object. Since each type is defined in terms of existing types, a 'component' relationship can be defined among the type managers. The salient feature of type extension is that it imposes a partial ordering on the set of type managers, on the basis of the 'component' relationship. This partial ordering improves
the clarity of the system design as well as enhances the orderly verification of the system.

The main issue in the use of type extension is the design of a mechanism to support the technique. When the user of a type wants to invoke a type operation on an object managed by the type manager, it passes an identifier of the object to the type manager. The purpose of the mechanism for type extension is to prevent the user from directly accessing the object's representation, while allowing the type manager to access the representation through the same identifier. A simple mechanism for this purpose, based on the use of small address spaces, is discussed in [Hab 76]. Since the mechanism for type extension primarily plays a role of protection, a 'capability' mechanism is most often used as the basis for the implementation [Red 74, Wul 74, Fer 74].

A 'capability' is a mechanism which unifies protection and addressing [DeV 66]. Very simply, a capability is a 'ticket' which identifies an object in the system. The possession of a capability is interpreted as the authority of the possessor to access the identified object. In an object-based system using capabilities, every object has a unique identification code (ID), which is not reusable. Coupled with a type identifier, this serves as an ID which is unique system wide. A capability also contains a set of 'access rights'. An access right provides a finer resolution of protection, since the invocation of any operation on an object requires the presence of the corresponding access right in the capability used to identify the object.

When a user of a type invokes an operation on an object through its capability, the type manager needs to obtain access to the representation of the object through the same capability. The crux of the type
extension mechanism is to implement this change in the protection context. The various schemes used are 'type amplification' [Jon 73, Wul 74], 'generalized sealing' [Red 74] and 'restricted sealing' [Fer 74].

Important to the concept of type extension is the notion that a type is, itself, an object [Jon 73]. A type is an object of the class defined by a 'root type'. The operation defined for a type object is the 'create' operation with which a new instance of an object, in the class defined by the type, is created. (The 'destroy' operation is an operation defined by the type and not on the type). According to the view of a type as an object, the set of all objects in the system compose a three-level tree structure. The leaves of the tree are the objects of different types. The intermediate nodes are the type objects of the system, each type belonging to the class defined by the 'root type'. The root of the tree is occupied by the root type object [Jon 73, Red 74].

Section I.3 Incorporating Software Subsystems

Initially, the emphasis in incorporating an operating system was in the design of capability architectures. These architectures supported the capability mechanism which was used by the operating system. The Plessey system [Eng 74] and the CAP system [Neu 77] are examples of such systems. The architecture for these systems supported the maintenance and handling of capabilities, but there was no incorporation of the operating system per se.

The VENUS system [Lis 72] was, perhaps, the only system incorporating components of the operating system into the architecture, that was actually implemented. This system was structured as a layered design. The lower levels of the operating system defined a simple segmentation subsystem, a limited multiprogramming subsystem, a multiplexed I/O chan-
nel and the maintenance of procedures. All these subsystems were implemented at the microprogram level. The rest of the operating system was implemented on the architecture defined by this microprogram.

A capability architecture, which has been proposed [RaS 76], suggests a similar approach. This design provides for a set of functions which belong to the operating system and are designed at the architecture interface. This architecture protects the generation of logical addresses and recognizes a set of non-hardware types at the architecture interface. The design assumes a firmware-based implementation.

Many of the operating systems which are structured by the layered technique suggest that the hardware/software interface can be set at any of the levels of design. The implication is that the architecture is defined at this interface. The defect in all these approaches is that there is no recognition of the inherent characteristic of the architecture interface. In other words, what is it that distinguishes between a software implementation, an implementation within the architecture, and a hardware implementation?

Another issue is that, in most of these systems, the subsystem which is to be incorporated is either simplified (as in the VENUS system), or large and unstructured (as in [RaS 76]). An important aspect of incorporating a non-trivial subsystem is the structuring of the subsystem to a fine level of resolution. In our approach, we emphasize the importance of this structuring.

Section I.4. Plan of the Thesis

The thesis is divided into two parts. In the first part, which consists of Chapter II, we take a top-down approach to designing the architecture. The chapter develops a model for linguistic abstraction and
proposes mechanisms to implement the features of the model. On this basis, the architectural features required to support these mechanisms are identified.

The next three chapters constitute the second part. They take a bottom-up approach to the design of the architecture and develop the architecture interface from the level of primitive resources. Chapter III discusses the design of the virtual memory subsystem. This chapter first characterizes the subsystem and then shows that the technique of type extension can be used to structure such a subsystem. The primitive types are defined at the level of real memory. Chapter IV discusses the design of the virtual processor subsystem. It presents the structure of this subsystem, with the primitive types being defined at the level of physical processors. The role of this subsystem in the internal implementation of the architecture is identified. In the design of the above two subsystems, a standard approach is used to resolve the mutual dependency between the two subsystems [Sax 75, Ree 76].

Chapter V discusses the most important part of the thesis, namely, the issues related to architectural incorporation. This chapter also presents the interface of the architecture, as seen by the software system. The interface is defined in the form of a set of operations defined for the types which are supported at the interface. Chapter VI summarizes the main contributions of this research and suggests further areas of exploration.

The second chapter of this thesis is not essential to the understanding of the subsequent chapters of the thesis. It serves more as a motivation for the design of an architecture with the proposed interface.

As a rule, all the chapters are organized with an introductory sec-
tion and a summary section. Most of the detailed sections of the chapters are organized with a preview and an overview subsection. The details of the design are presented in the form of the operations defined by each abstraction (type). The semantics of these operations are described in conjunction with a semi-formal specification.
CHAPTER II

ARCHITECTURAL FEATURES TO SUPPORT LINGUISTIC ABSTRACTION

Section II.1  Introduction

This chapter presents the design of mechanisms which support the concept of abstraction in a programming language. The features required of the underlying architecture are derived from the design of these mechanisms. The abstraction concepts are presented by means of a linguistic model. This model emphasizes features related to data and procedure abstractions* [Lis 77, Wul 76, JoL 75]. The highlight of these features is a generalized and very flexible means of defining parameterized data abstractions. Another important feature is a dynamic type creation facility provided by the definition of procedure abstractions.

The linguistic model assumes that the programming environment is object based. Any abstraction language which avails of the abstraction features would be based on the linguistic model and will have to enforce this object-based environment. The abstraction features of the linguistic model are supported by a set of mechanisms. These mechanisms are used by the compiler of the abstraction language. The uniqueness of these mechanisms is that they themselves conform to an object-based system, in that they are defined as the operations for a class of objects. The implication is that an architecture which defines an object-based system can be used to implement these mechanisms, as well as define the programming environment for the abstraction language.

Section 2 presents the linguistic model, which defines the abstrac-

* Control abstractions like CLU's 'iterator' can be supported in a manner similar to procedure abstractions.
tion features. Section 3 presents the design of the mechanisms supporting these features. Section 4 establishes the set of architectural features required for these mechanisms. The last section summarizes our approach to supporting linguistic abstraction.

Section II.2 The Generalized Linguistic Model (GLM)

The generalized linguistic model (GLM) serves as a means of formalizing the linguistic abstraction features that we are proposing. GLM concentrates on only those programming language concepts which are related to abstraction. Some of these concepts are based on those existing in an existing programming language—CLU [Lis 77]. However, GLM defines more generalized features for abstraction than CLU. GLM assumes that all data entities in the programming environment are regarded as objects. A variable, in such an environment, is semantically interpreted as containing a reference to an object, rather than containing the data itself. In essence, GLM assumes a programming environment similar to the CLU environment.

In an object-based environment, each object belongs to a type, which defines the set of operations for accessing the object. We had seen that a set of 'primitive' types exist in such an environment. Additional 'extended' types can be defined using these primitive types and already defined extended types. In a programming language environment, the set of primitive types are defined by the programming language. Data abstraction is the term for defining additional types using primitive or already defined types. Hence, a data abstraction definition is synonymous to a type definition. Henceforth, we will use the latter term. A procedure abstraction definition is the standard procedure definition found in most programming languages. Each type definition contains a
set of procedure definitions which implement the operations of the type.
A procedure definition can also be defined independent of any type.
Hence, there are two scopes for a procedure definition. We will now present the features which GLM defines for type and procedure definitions.

a) **Type definition**

A type definition defines a class of objects. It does this by defining a 'sample' representation for each object in the class, and a set of procedures which can manipulate the representation of an object. Most of these procedures are invokable* by the users of the type and are called the operations of the type. A few of the procedures are internal and can only be called internally by the operation procedures. The representation of an object consists of a set of objects of already existing types. These types defining the representation are called the component types for the newly defined type.

A parameterized type definition is the use of one type definition so as to define a set of types. The basic idea is that all these types are similar in structure but can have different component types. Thus a parameterized type definition allows component types to be decided as parameters of the definition. The type defined by such a type definition is called a parameterized type. A parameterized type identifies a class of types, while a non-parameterized type identifies a class of objects. An example of a parameterized type is ARRAY[ ...... ]. This type can identify an ARRAY[real], ARRAY[integer], ARRAY[TREE], etc.

Here all the ARRAY types are similar in structure, but the type of the

* The term 'invoke' is used for calling a procedure which implements an operation, to manipulate the representation of an object. Simple 'calling' a procedure does not have the added connotation of an object's representation being accessed.
elements of the array is different.

A parameterized type has a set of parameters. Each unique combination of these parameters generates a new type. Hence, a parameterized type is a 'type generator'. A parameterized type generates types by assigning values to its parameters. We will call this assignment of values as 'fixing' the parameters. To distinguish between the two forms of types, we will refer to a parameterized type as 'ptype', and use the term 'ftype' to refer to a nonparameterized type or a ptype which has all its parameters fixed. Thus, a ptype defines a class of ftypes and an ftype defines a class of objects. To distinguish the parameters of a ptype from other kinds of parameters, we will call these as type parameters.

There are three kinds of type parameters for a ptype. The first two kinds of type parameters can only take* an ftype value, the third kind can only take an object value. We differentiate between a type parameter which takes an ftype value which is used in deciding a component type, and a type parameter which corresponds to an ftype for which the ptype simply provides a container service. For example, assume the ptype ARRAY has only one type parameter, which identifies the ftype of the elements in the array. If the ptype ARRAY defines operations which cause operations to be invoked on elements in the array, then the type parameter identifies a component type and is of the first kind. On the other hand, if ARRAY only defines indexing operations which reference an element without invoking any operation on the corresponding object, then the ptype provides a container service. In this case, the type parameter

* Taking an ftype or object value is in the sense of the parameter being 'bound' to the ftype or object. We will discuss the nature of this binding later.
identifies a contained type and is of the second kind.

The third kind of type parameter can only take an object value. In this case, the type parameter does not identify a component type or a contained type, but is simply a parameter used in deciding some particular aspect of the derived ftype. For example, the ARRAY ptype can have a type parameter of this kind, which decides the range of its array indexes*. We will use the abbreviations TPF, TPA and TPO to refer to type parameters of the first, second and third kinds, respectively. Of these TPFs which take type values identifying component types, are the most relevant to our discussion.

A ptype definition is similar to a nonparameterized type definition. Both define a set of component types. The main difference is that each component type in a ptype definition identifies a class of ftypes, while each component type in an ftype definition identifies a single ftype. When the type parameters of a ptype, particularly the TPFs are fixed, then a particular ftype in each component type's class is identified.

The set of types which exist at any point in time, either because they are primitive or because they have been defined, are called the 'external types'. Each new type definition adds a type to this set. The set of external types consists of both ftypes and ptypes. In the latter case, each ptype actually defines a set of external types. An external ftype value can either be the name of an external ftype or the name of an external ptype combined with an ftype value for each TPF** of the ptype.

---

*Strictly speaking, it can be argued that this range is more an attribute of the array object rather than the array ftype. We provide this kind of type parameter so that attributes which do not take ftype values can be used to differentiate between ftypes.

**Values have to be assigned to any type parameter of the other two kinds also. Since this is not crucial to our discussion, we will always assume that this is done.
An external ptype value can only be the name of an external ptype.

When a component type takes an ftype value, the implication is that the component type has been bound to an external ftype, identified by this value. As we had seen, this ftype could have been defined as being nonparameterized or it could have been derived from a ptype. Similarly, when the TPF of a ptype takes an ftype value, the implication is that the TPF is bound to the corresponding ftype. An additional complication in this case, is that some of the component types may have been defined to be dependent on the ftype value which this TPF takes. So these will also have to be bound to the external ftype. We will now discuss how a component type of a ptype can be defined to be dependent on the ftype value that a TPF takes.

The straightforward way is for the component type to take the same value as the TPF. For example, if the ptype ARRAY has one component which identifies the highest element value, then the corresponding component type is to be the same as the ftype for the array elements. A more flexible way of making a component type dependent on a TPF is to associate a component type with an external ptype. The component type can be defined to correspond to the resulting ftype when the parameters of this external ptype are fixed by TPFs. Hence, the component type now indirectly depends on the TPF through some external ptype. As we will see, this indirection can be defined through more than one level.

As an example, suppose the ARRAY object is to be implemented as a linked list of elements. Let LINKED_LIST[ .... ] be an external ptype with a single TPF, say T1, which identifies the ftype of the elements being linked. The ptype definition for ARRAY (let its single TPF be called T2) can define a component type as LINKED_LIST[T2]. This defines
the TPF of the linked list - T1, to be fixed with the same value with which the TPF of array - T2, is fixed. More important, an ARRAY object is implemented as a linked list object no matter what the ftype of the array elements are.

In the above example, the name of the TPF for ARRAY is used as an ftype value in fixing the TPF of LINKED_LIST. This is because, internal to a ptype definition, the names of TPFs can be used as ftype values. There is an additional set of names which can also be used as internal ftype values. These names are declared by 'equivalence declarations'. Each such declaration equates a new internal ftype name with the ftype produced by fixing the parameters of some external ptype. These ftypes are called 'type equivalence ftypes' (TEF). In the above ARRAY example, a TEF can be declared as LINKED_LIST[T2]. This TEF can then be used in the component type declaration. The parameters of the external ptype, which are fixed in an equivalence declaration, can themselves take any valid external or internal ftype value. This implies that the names of ftypes which have been defined by previous equivalence declarations in the same ptype definition, can also be used as an ftype value in fixing these parameters. We call the use of previously defined TEF names in the parameter fixing of a subsequent equivalence declaration, as 'ftype chaining'.

There is one other ftype name which can be used internally as an ftype value, but only in a restricted fashion. This corresponds to a recursive usage of the type name which is being defined. In the case of a ptype definition, the name of the ptype being defined can be used internally as an ftype value to denote a recursive structure.
To summarize, we have identified three forms of internal ftype values within a ptype definition. The first form corresponds to the names of the TPFs of the ptype. The second form corresponds to the names of the TEFs defined by equivalence declarations within the ptype definition. The third form corresponds to a recursive usage of the name of the ptype which is being defined. In addition to these three internal forms, an external ftype value can also be used. All these four forms of ftype values can be used to fix the TPFs of an external ptype in an equivalence declaration. All but the recursive form can be used to identify a component type. A recursive form can, however, be used indirectly through a TEF, to identify a component type.

A ptype definition consists of a 'type parameter construct' and a ptype body. The type parameter construct identifies the set of type parameters which are defined for the ptype. An internal name is associated with each type parameter. The kind of the type parameter is also declared. The ptype body contains a set of six constructs. The 'operations construct' identifies the set of external operation names defined by the ptype. When a ptype has a TPF parameter, its definition can require a set of operations which any external ftype, to which this TPF is bound, must have. More important, since a ftype defines a set of operations, the set of operation names which are used internally with respect to any ftype, bound with the TPF, should be specified. This is provided by a 'requires construct'. This construct is similar to CLU's 'requires specification'. This construct associates for each TPF, a set of internal operation names with their required operation specifications.

*Operation specifications can be simply through listing the types of input and return parameters, or can be more elaborate using formal specification techniques [LiZ 75, Gut 77].
The third construct in a ptype body is the 'type equivalence construct', which consists of a set of equivalence declarations. As we had seen, each such declaration equates an internal TEF name with the ftype produced from an external ptype. Both internal and external type values can be used for the parameter fixing of this ptype. The fourth construct is the 'component construct'. These declare the set of component types for the ptype. Each component declaration identifies the name of a component object with its corresponding component type. This object name is used by the type's procedures in manipulating the representation.

We also provide for a 'common construct' which identifies a set of common types. Each such type corresponds to a common object. A common object is shared across an ftype by all the objects belonging to the ftype. These objects can be used for bookkeeping and monitoring on a type wide basis.

The last construct in a ptype definition is the 'procedures construct'. This defines a set of procedures for the type. Most of these correspond to external operation names, but a few are internal to the type. Each procedure declaration equates an external or internal operation name with a type procedure definition. We will discuss procedure definitions in the next section.

We will use a formal means of specifying the characteristics of the generalized linguistic model (GLM). The specification of a ptype definition is shown in Fig. 2.1. We specify a ptype definition as a series of expansion rules using a format* similar to the Backus-Normal form.

* - Curly brackets '{...}' indicate that the contained syntax can be repeated null or more times.
- The angle brackets '<...>' identify a construct or declaration.
- The '|' delimiter indicates a choice of constructs.
- Reserved words are underlined.
- All other special characters are part of the type definition syntax.
<ptype_concept> ::= <ptype_n> = <ptype_definition>
<ptype_definition> ::= ptype[<type_parameter_construct>]
   <ptype_body>
<type_parameter_construct> ::= {<type_parameter_declarations>}
<type_parameter_declaration> ::= <tpf_n> : ftype | <tpa_n> : anytype
   <tpo_n> : <et_n"ftype>
<ptype_body> ::= begin <ptype_n>
   <operations_construct>
   <type_requires_construct>
   <type_equivalence_construct>
   <component_construct>
   <common_construct>
   <procedures_construct>
   end <ptype_n>
<operations_construct> ::= <ext_oprn_n> {, <ext_oprn_n>}
<type_requires_construct> ::= <tpf_n> has <requires_specification>
<requires_specification> ::= --
<type_equivalence_construct> ::= {<type_equivalence_declaration>}
<type_equivalence_declaration> ::= <tef_n> :- <et_n"ftype>
   [<type_equivalence_fixing>]
<type_equivalence_fixing> ::= <et_n"ftype> | <tpf_n> | <rt_n>
   <tef_n>
<component_construct> ::= <cpt_obj_n> : <cpt_obj_ftype> ,
   {<cpt_obj_n> : <cpt_obj_ftype> ,}
<cpt_obj_ftype> ::= <et_n"ftype> | <tpf_n> | <tef_n>
<common_construct> ::= {<cmm_obj_n> : <cmm_obj_ftype> ,}
<cmm_obj_ftype> ::= <et_n"ftype> | <tpf_n> |
   <tef_n (without rt_n)>

Fig. 2.1 The Specification of a 'ptype' definition
<procedure_construct> ::= <ext_oprn_n> =
   <type_procedure_definition>
   {{<ext_oprn_n> =
      <type_procedure_definition>}
   {{<int_oprn_n> =
      <type_procedure_definition>}

Fig. 2.1 (contd.) The Specification of a 'ptype' definition
These rules describe the syntax of a ptype definition, but in doing so illustrate the use of internal ftype values and other concepts that we have been discussing. In the specification, internal names are identified with a '_n' suffix. The syntax 'et_n"ptype' is used to denote an external ptype value, and the 'et_n"ftype' syntax denotes an external ftype value. The 'rt_n' syntax identifies a recursive type usage. The 'type parameter declaration' rule specifies how the three kinds of type parameters are identified. The 'type equivalence declaration' rule specifies the form of an equivalence declaration. The 'type equivalence fixing' rule specifies the allowed forms of internal ftype values for the parameter fixing. The 'cpt obj ftype' rule specifies the ftype values which can be used to identify a component type. Note that for the analogous case of a common type, a TEF ftype name, which implies a recursive usage, is not allowed as an ftype value.

b) Procedure definition

A procedure has two scopes in which it can be defined. One scope is for a procedure definition to be contained by a type definition. The other scope is for a procedure to be defined independently. A procedure can be regarded as being equivalent to a parameterized type. When a procedure is called, its set of parameters are fixed with values. This corresponds to deriving an ftype. The actual activation (execution) of the procedure is the object created from this ftype. We will describe the GLM features for procedure definition from this point of view.

The parameters in a procedure definition are called procedure parameters. A procedure parameter can take an ftype value or an object value. A procedure parameter which only takes an ftype value is called PPF and is the procedure analog of a TPF. A procedure parameter which only
takes an object value is called a PPO.

There are five kinds of PPO parameters. In each PPO declaration in the procedure parameter construct, a PPO is declared with an ftype name. This name identifies the ftype of the object which is passed as a parameter at the time the procedure is called. Four of the PPO kinds correspond to different forms of the ftype name. First, an external ftype name can be used. This is the only kind of parameter objects recognized in conventional languages. In the second form, a TPF name can be used indicating that once this TPF is fixed, the type procedure should expect objects of the corresponding ftype. Since this ftype is not known at the time of the ptype definition, the internal operation names defined in the 'type requires construct' are used in invoking operations on this kind of parameter objects. The third form is the use of a TPA name. Since the type only provides a container service for objects of this kind, no operations are invoked on such objects. The fourth kind of PPOs correspond to the use of a TEF name. Since this ftype is derived from an external ptype, the operations defined for parameter objects of this kind are known at the time of the ptype definition.

A procedure parameter object (PPO) can also be declared as 'generic'. This indicates that the ftype of the actual parameter object passed to the procedure can vary with each call of the procedure. A generic PPO is distinct from a PPF which can take different ftype values. A PPF can only take ftype values, while a PPO object declared as generic can only take object values. However, there is a provision by which a PPF can be declared as the ftype for a corresponding generic object. Thus, for each call of the procedure, this PPF takes the value of the ftype to which the generic object belongs, while the PPO takes the object value.
It must be emphasized that a procedure parameter taking an ftype value implies that an ftype is passed to the procedure and a parameter taking an object value implies that an object is passed. In the former case, an ftype authorizes creation of objects of that type, while in the latter case, an object does not authorize any additional creation. The use of an ftype (passed either as a type parameter or procedure parameter) to create objects of that type has been inbuilt into the semantics of the linguistic model.

A procedure definition contains a return construct, which identifies the ftypes of the objects returned by a procedure. The return specification consists of the ftype names for these objects. The implication is that the return construct identifies all those parameter objects which could have had state changing operations invoked on them within the procedure.

The procedure body contains four constructs. The 'requires construct' is similar to that of a ptype definition and contains a set of internal operation name to operation specification—associations for each PPF of the procedure. Furthermore, since operations are invoked on generic objects, whose ftypes are not fixed at the time of the procedure definition, the procedure requires construct also declares the set of associations for these generic types.

The second construct is the 'procedure equivalence construct'. This construct is similar to the type equivalence construct and declares a set of procedure equivalence ftypes (PEF). This is the procedure analog of a TEF. The parameters of the external ptype in such a declaration can be fixed with any of the type wide internal ftype values. Within a procedure, there are two additional forms of internal ftype values—a proce-
Fig. 2.2 The Specification of a 'type procedure definition'
dure parameter ftype (PPF) and a procedure equivalence ftype (PEF). However, the internal ftype value corresponding to a recursive usage cannot be used in these procedure declarations.

The third construct is the 'local object construct' and is the analog of a type's component construct. The set of local objects are newly created for each activation of the procedure. The types of these local objects are called local types and declared in a similar fashion to component types. A local type can be declared by any of the internal ftype values, valid within a procedure definition—external ftype, TFF (type parameter ftype), TEF (type equivalence ftype), PPF (procedure parameter ftype), and PEF (procedure equivalence ftype).

The fourth construct in a procedure body is the code construct. We will not discuss this issue, since our main interest is in the type creation features of ptypes and procedures. We should point out that in a GLM based language, the code consists of control constructs and operation invocations. The formal specification of a procedure definition in the context of a containing ptype definition, is shown in Fig. 2.2.

A procedure can also be defined independently. Here, the internal ftype names which were defined on a type wide basis in the previous case, are no longer valid. Therefore, there are only three kinds of procedure parameters—PPF, PPO for an external ftype, and a PPO for a generic object. The parameter fixing in a procedure equivalence declaration is also limited to procedure wide ftype names, as is the declaration of local types. The specification for an independent procedure definition is shown in Fig. 2.3.

c) Overview

In defining the features of the linguistic model (GLM), we have
<independent_procedure_definition>::= proc (<iproc_parameter_construct>)

       returns (<iproc_return_construct>)

       <iproc_body>

<iproc_parameter_construct>::= {<iproc_parameter_declaration>}

<iproc_parameter_declaration>::= <ppf_n>: ftype | <ppo_n>: generic

       <ppo_n>: <et_n"ftype> |

       <ppf_n>: gtype (<ppg_n>)

<iproc_return_construct>::= {<iproc_return_declaration>}

<iproc_return_declaration>::= <et_n"ftype>|<ppf_n> |

       gtype (<ppg_n>)

<iproc_body>::= same as <tproc_body>

<iproc_requires_construct>::= same as <tproc_requires_construct>

<iproc_equivalence_construct>::= {<iproc_equivalencedeclaration>}

<iproc_equivalence_declaration>::= <pef_n> := <et_n"ptype>

       [<iproc_equivalence_fixing>]

<iproc_equivalence_fixing>::= <et_n"ftype>|<ppf_n>|<pef_n>

<ilocal_object_construct>::= {<loc_obj_n>:<iloc_obj_ftype>,}

<iloc_obj_ftype>::= <et_n"ftype>|<ppf_n>|<pef_n>

<iproc_code_construct>::= - - - - - - - - - - - - - - - - - - -

Fig. 2.3 The Specification of an 'independent procedure definition'
stressed the generality and flexibility provided in the definition of parameterized types and procedures for these types. We will summarize these features. First, the component type of a ptype can directly or indirectly depend on the ftype value that a TPF (type parameter ftype) takes. Second, by allowing a recursive usage of the ptype name, a component type can depend indirectly on the ftype for which it is a component. This allows recursive type definitions. Third, the ftype chaining facility allows the component type to depend on the type parameter through several levels of indirection. In fact, there is no limit on the number of these levels.

As an example, we have illustrated a DIRECTORY ptype definition in Fig. 2.4. The DIRECTORY object associates a KEY with a set of N entries. The entries for one key are linked together in a linked list, but an additional link is also provided for each of these entries. The key and linked associations are maintained in a tree structure. We observe that by the ftype chaining facility, a very flexible directory type can be specified.

A final feature of the linguistic model is that the same features defined for a type definition are also provided with respect to the local objects in a procedure definition. In a procedure, the features are enhanced by the fact that a procedure parameter ftype (PPF) can be fixed with ftype values at execution time. This feature provides a dynamic type creation facility.
\text{DIRECTORY} = \text{ptype} [\text{KEY:ftype, ENTRY1:ftype, \ldots, ENTRYn:ftype}]

\text{begin DIRECTORY}

---

\text{equivalence}

\text{PRI} : \text{PAIR [ENTRY1, ALINK]}

---

\text{PRn} : \text{PAIR [ENTRYn, ALINK]}

\text{LIST : LINKED_LIST [PRI]}

\text{ENTRY : PAIR [KEY, LIST]}

\text{STRUC : TREE [ENTRY]}

---

\text{component}

\text{DIR : STRUC,}

---

---

end DIRECTORY

\textit{Fig. 2.4 An example of 'ftype chaining'}
Section II.3 The Design of the Abstraction Mechanisms

In this section, we will design the mechanisms with which the abstraction features that we have discussed, can be implemented. These mechanisms are designed in the context of an object based system. The key aspect of the design is the recognition that the set of ptypes and the set of ftypes, each constitute a distinct class of objects in the system. In Chapter I, we had introduced the idea that a type is an object of a 'root type'. This lead to a three-level tree structure of all the objects in the system. In the previous section, we distinguished between ptypes and ftypes, and introduced the idea that a ptype defines a class of ftypes. By recognizing ptypes and ftypes as objects in the system, we can organize the set of all objects in the system into a four-level tree structure. The 'root type' is the only object at the root (level 1). The set of ptypes constitutes the next level, where each ptype is an object of the 'root type'. The set of ftypes constitutes the third level. Each ftype is an object of the ptype that it is derived from. All ftypes which are defined as nonparameterized types are objects of an implicit 'master ptype'. Finally, the leaves of the tree structure are composed of elementary objects belonging to ftypes.

In the discussion of the linguistic model, we had identified objects, ftypes and ptypes through names. This was feasible as long as the discussion was confined to a programming language environment. The mechanisms supporting the abstraction features are defined and implemented in an execution environment defined at the architecture interface. Hence, there needs to be a different mechanism for identifying objects in an object-based execution environment. The standard mechanism used in such an environment is a 'capability'. An object in the execution en-
environment is identified by a name which is unique system wide. A capability identifies an object by containing this name. A capability also contains the type identity of the object. The usefulness of a capability is that it is tamperproof, because its contents cannot be altered, and it identifies an object on a system wide basis.

The key idea in an object based system is that each object has a representation which is accessible to the operations defined to access the object. We had reasoned that for elementary objects (those at the leaves of the four-level tree structure), the representation consists of a set of component objects, where each component object belongs to an ftype identified by a component type. The architecture or programming language, which enforces the object-based environment, ensures that the activation* of an operation defined for the object, has access to the set of component objects. If the architecture itself enforces an object-based environment, there is no necessity for the compiler of an abstraction language to implement the addressing and linking mechanisms required to enforce this environment. The added advantage is that the mechanisms supporting the abstraction features can, themselves, be implemented in the architecture's object-based environment. The crux of the design of the abstraction mechanisms is the design of the representation for ftypes and stypes and the definition of a CREATE operation for each of these classes.

a) **Representation segments**

On the basis of the previous discussion, we can assume that the

---

* The activation of an operation is the **execution state** of the procedure implementing the operation.
architecture ensures that the activation of an operation has access to
the representation of the object that it is invoked on. This is true
irrespective of whether the object is elementary (level 4), ftype (level
3), or ptype (level 2). Since this representation has to exist some-
where, we will assume that this representation is contained in a repre-
sentation segment* (rep segment). Thus, each object in the system is
associated with a rep segment that is accessible to any operation de-
finied for the object.

There are two main issues in the design of the abstraction mechan-
isms:

1) We will identify the information which needs to be contained
in the rep segment for a ptype, ftype and an elementary
object. In the first two cases, the respective forms of the
CREATE operation accesses the rep segment in order to create
an ftype or an elementary object.

2) We will specify the actions to be carried out by the two
CREATE operations. One creates a ftype from a ptype, the
other creates an elementary object from an ftype. We em-
phasize the similarity between the ptype - ftype and
ftype - object relationships.

We will discuss the first issue in this subsection and the second issue
in the next subsection.

An elementary object (elem object) is created from an ftype. The
ftype in turn, is created from a ptype. The main idea in designing the

* Segments are variable size memory containers. They are discussed in
detail in the following chapters of this thesis. For now, we treat
them as contiguously addressable units of storage, of variable size.
rep segment of a ptype or a ftype is that it should contain the information with which the rep segment of the created ftype or elem object can be derived. The rep segment of an elem object has to contain the set of capabilities to the component objects. It also has to contain a set of capabilities identifying the procedures defined by the object's type.

Since 'common' objects are shared by all objects created from the same ftype, the object's rep segment should also contain this set of capabilities. There is a fourth set of capabilities called 'omap' capabilities, which are also contained in an elem object's rep segment. We will establish the need for these later. The rep segment for an elem object is shown in Fig. 2.5. The first element in the rep segment is a capability identifying the ftype of the object.

An ftype's rep segment should contain the information to obtain the above four sets of capabilities. An ftype rep segment contains three sets of capabilities, which it distributes to all objects created from it. These sets are the 'common', 'omap' and procedure capabilities. The ftype rep segment contains a fourth set of capabilities which consists of a set of ftype capabilities. These ftype capabilities correspond to the component types and are used to create the set of component objects.

The rep segment for an ftype is shown in Fig. 2.6. The first capability in the rep segment identifies the ptype from which the ftype was created.

The rep segment for a ptype needs to contain the information to create the ftypes corresponding to component and common types. This rep segment also has to contain the information with which the procedure objects corresponding to the type's procedures can be created. The code for the procedures can be formed at the time of compiling the ptype definition. However, since local types (the types of the local objects) in
Fig. 2.5 An elementary (elem) object's representation segment
Fig. 2.6 An FTYPE's representation segment
the procedure can depend on the type parameters, the formation of the ftypes for the local objects has to be delayed to the time of ftype creation from the ptype. At the time of the ftype creation, the information in the ptype's rep segment is used to form the ftype's rep segment. This is analogous to the formation of an elem object's rep segment from the ftype's rep segment, at the time of the object creation. The ptype rep segment is, itself, formed at the time when a ptype definition is compiled.

Fig. 2.7 illustrates the information maintained in a ptype's rep segment. First, it contains the set of ptype capabilities which identify the external ftypes, which were used in the type equivalence declarations. Since each equivalence declaration equates an internal ftype name (TEF name) with the ftype created from an external ptype, each ptype in this set of capabilities corresponds to a TEF (type equivalence ftype). Next, the ptype rep segment contains a set of capabilities which identify the component types. A component type can be declared with any of the three internal ftype values or an external ftype value. Only in the last case does a corresponding ftype exist at the time of the ptype compilation. For the three internal ftype values, the compiler uses 'indirect' capabilities, one for each form of internal ftype. The 'tpf_type' capability contains the index of the TPF which is used as the internal ftype value, the 'tef_type' capability contains the index of the TEF which was used. Since the first set of capabilities contains a ptype capability for each TEF, the latter index indexes into this set. The 'r_type' capability need not contain any index, as it represents a recursive usage. The use of indirect capabilities relies on the fact that at the time of the ftype creation from the ptype, the set of external ftypes
'ptype' caps for creation of ftypes for TEFs.

Component types:
'ftype', 'tpf_type',
'tef_type',
r_type' caps.

Common types:
'ftype', 'tpf_type',
'tef_type' caps.

Type Proced. 1:
Procedure defn.
information

----------------

Type Proced. N:
Procedure defn
information

Information for creation of ftypes for TEFs.

Fig. 2.7 A PTYPE's representation segment
to which the TPFs are to be bound, is accessible. Hence, the ptypes associated with TEFs can be used to create the respective ftypes and these, together with the ftypes bound to the TPFs, can be used to replace the indirect capabilities in the newly formed ftype's rep segment. The newly formed ftype capability can be used to replace the 'r_type' capabilities.

The information maintained for the common types is similar to that of the component set. The ptype rep segment contains a set of indirect and ftype capabilities to identify the common types. The ptype rep segment contains 'procedure definition information' for each of the type procedures. Finally, the ptype rep segment contains the information necessary to create the ftypes from the ptypes associated with the TEFs. Each type equivalence declaration, equates a TEF with an external ptype, whose parameters (TPFs in particular) are fixed with external and internal ftype values. The capabilities to these external ptypes are contained in the first capability set of the rep segment. The information contained in the last part of the rep segment, specifies through indirect indexes (similar to those used for component and common types), the capabilities which are to be used to identify the actual ftypes which fix the TPFs of these external ptypes. The key idea is that the ftypes corresponding to a ptype's TEFs are created from the external ptypes, only at the time when an ftype is created from the containing ptype.

The implication of 'ftype chaining' is that at this time, an ftype which has just been created for a TEF, is subsequently used as the actual ftype to fix a TPF in the ftype creation of another TEF. The information for the ftype creation for TEFs also contains the capabilities identifying the necessary external ftypes, which were accessible at the time of the ptype compilation. These correspond to the use of external
ftype values to fix TPFs in the equivalence declarations.

The 'procedure definition information' part contains the information with which the procedure objects for the type procedures can be created. This creation has to be delayed to the time of ftype creation because the local types of the procedure can be specified in terms of type wide internal ftype values—TPF and TEF names. As we had seen, the ftypes corresponding to these values are accessible only at the time of ftype creation from the ptype. Fig. 2.8 illustrates the procedure definition information maintained for each procedure. The code for the procedure is maintained in a code segment. The first capability identifies this code segment. The next set of capabilities consists of ptype capabilities. These identify the external ptypes which were used in the procedure equivalence declarations. Each PEF (procedure equivalence ftype) has a corresponding ptype in this set.

The local types defined by a procedure are identified by a mixture of ftype and indirect capabilities. Ftype capabilities are used if an external ftype value was used to declare a local type. Within a GLM procedure definition, there are four different forms of internal ftype values. Two of these are internal to a procedure definition, the other two are internal on a type wide basis. The use of ftype values internal to a procedure definition is indicated by 'ppf_type' and 'pef_type' capabilities which contain the index of the respective PPF or PEF. For the internal ftype values which can be used type wide, the local type is indicated by a 'twf_type' capability. The definition contains a set of indirect capabilities which indicate the type wide internal ftypes ('twf' set) which had been used in that particular procedure definition. Each of the 'twf_type' capabilities used for the local types contains
Capability to CODE segment $\rightarrow$ c_seg

ptype

'ptype' caps. for creation of ftypes for PEFs.

Local types:
'ftype', 'ppf_type', 'pef_type', 'twf_type' caps.

Type-wide ftypes:
'tpf_type', 'tef_type' caps.

Information for creation of ftypes for PEFs.

Fig. 2.8 Procedure Definition Information
an index into this set. Hence, there are two levels of indirect capabilities when a type wide internal ftype is used in a procedure definition. The information which specifies the actual parameters to be used in creating ftypes for the PEs, constitutes the last part of the procedure definition information. The use of a type wide internal ftype value in a procedure equivalence declaration, is specified in this part as a 'twf_index'.

At the time of creating an ftype from the ptype, the actual ftypes for TPFs and TEFs are available. At this time, the 'procedure definition information' is used to form the rep segment of a procedure object. The indirect capabilities in the 'type wide ftype' (twf) set are replaced with actual ftype capabilities. Furthermore, any 'twt_type' capabilities in the local types are replaced with the appropriate ftype capabilities from the 'twf' set. Since procedure parameters are not fixed until the procedure is called at execution time, the creation for ftypes for PEs is delayed until this time. The replacement of the 'ppf_type' and 'pef_type' capabilities in the local type set also occurs only at this execution time. The rep segment for a procedure is shown in Fig. 2.9. The structure of the rep segment parallels the 'procedure definition information' maintained in the ptype rep segment. However, the 'twf' set now only contains ftype capabilities. The first capability in the procedure's rep segment identifies the procedure object itself. It can be used in recursive calls of the procedure.

b) The CREATE_OBJECT and CREATE_FTYPE operations

The CREATE_OBJECT operation is defined on an ftype and creates an elementary object from this ftype. The implementation of this operation is straightforward. The operation creates a rep segment for the elem ob-
'procedure' cap. to itself → proc
  capability to CODE segment → c_seg
   "ptype" caps. for creation of ftypes for PEFs
    Local types: 'ftype', 'ppf_type', 'pef_type' caps
     ftype
  Type-wide ftypes: 'ftype' caps.
   Information for creation of ftypes for PEFs.

Fig. 2.9 A PROCEDURE's representation segment
ject and associates this segment with a newly created object capability. three of the capability sets in the ftype's rep segment—the omap, the common, and the type procedure—are simply copied into the elem object's rep segment. The CREATE_OBJECT operation uses the component ftype capabilities to create the set of component objects and inserts the capabilities to these in the newly created elem object's rep segment. Alternatively, the design can allow the CREATE_OBJECT operation to receive a set of parameter values, with which it can decide on a subset of the component objects to be created.

The CREATE_FTYPE operation is defined on a ptype and creates an ftype from the ptype. The operation requires the actual values of the type parameters to be passed to it. These values are passed as capabilities identifying the actual parameters. Therefore the CREATE_FTYPE operation has access to the external ftypes to which the TPFs of the ptype are to be bound. The first step in the CREATE_FTYPE operation is to create a rep segment for the ftype and associate a new ftype capability with it. The second step is to create the ftypes for the TEFs (type equivalence ftypes) of the ptype. It uses the appropriate ptype capabilities maintained in the ptype's rep segment and invokes the CREATE_FTYPE operation on these ptypes. It uses the TEF ftype creation information to select the capabilities identifying the actual parameters. After the second step, the CREATE_FTYPE operation has access to the ftypes corresponding to all the TPFs and the TEFs of the ptype.

The third step is to insert the ftype capabilities for the component types. The set of ftype and indirect capabilities in the component set of the ptype's rep segment are used for this purpose. The ftype capabilities in the component set are simply copied into the ftype rep
segment. If an indirect capability represents a component type, then for a 'tpf_type' or 'tef_type' the ftype capability associated with the indexed TPF or TEF is inserted into the ftype rep segment. For an 'r_type', the newly formed ftype capability is inserted. The fourth step is to derive the set of ftype capabilities for the component objects using the same sequence of actions. However, these ftype capabilities are then used to create a set of common objects and it is the capabilities to these that are copied into the ftype's rep segment.

The fifth and final step of the CREATE_FTYPE operation is to form the rep segment for each of the type procedures. The procedure definition information is used for this purpose. First, the CREATE_FTYPE operation creates a rep segment for the procedure and associates a new procedure capability with this segment. This procedure capability is inserted as the first element in the segment. The ptype capabilities for the PEFs (procedure equivalence ftypes) are directly copied from the procedure definition part to the procedure's rep segment. Next, local types which are represented by ftype, 'ppf_type' and 'pef_type' capabilities are also directly copied into the procedure rep segment.

For the 'type wide ftype' (t wf) set, each 'tpf_type' or 'tef_type' capability in the set is used to identify the ftype capability associated with the indexed TPF or TEF. This ftype capability is then inserted into the 't wf' set of the procedure rep segment. After this, for each 't wf_type' capability which represents a local type in the 'procedure definition information', the indexed ftype capability is inserted into the local type set of the procedure's rep segment. The final stage of the fifth step is to insert the newly created procedure capability into the ftype's rep segment. This sequence of actions is repeated for each
of the type's procedures.

A last issue which needs to be considered is the assignment of the responsibility for the dynamic creation of the ftypes necessary for the local types of a procedure. One possibility is to implement this within the CALL operation defined on a procedure. An alternative implementation is for the compiler to automatically insert the code for this purpose. From the point of view of avoiding the check for this dynamic type creation, even if it is not required by the procedure, we contend that the latter implementation is more suitable.

c) The use of 'omaps'

When a component, common or local type is declared by a TPF (type parameter ftype), then objects of this ftype can be created and operations invoked on them. The compiler for an abstraction language generates code for an operation invocation by specifying an address to the capability identifying the object and using an operation index to identify the operation which is to be invoked. The complication arises if an object's ftype is decided by a TPF. This is because the operation indexes of the actual ftype to which the TPF will be bound, is not known at the time of compiling the ptype definition. In fact, the operation indexes can change for different ftypes to which the TPF can be bound. This problem is resolved by the use of internal operation indexes and omap capabilities. The omap capability maps from an internal operation index to the actual operation index. These omap capabilities are maintained in an elem object's rep segment and used in invoking operations on objects which correspond to a TPF usage.

The 'requires construct' of a ptype definition associates for each TPF, a set of internal operation indexes (names) with their correspond-
ing operation specifications. This 'requires specification' of a ptype is associated with the ptype in a system wide library*. Whenever an ftype is created from this ptype through the CREATE_FTYPE operation, an omap capability is passed with each actual ftype capability corresponding to a TPF. The omap** object maps from the internal operation indexes to the operation indexes of the actual ftype. If this ftype creation occurs when a compiler is compiling a ptype definition, then the compiler consults the system library to decide the mapping for the omap.

The above argument holds if the TPFs of the ptype are being fixed with external ftypes. But what if the TPFs are being fixed with internal ftype values at the time when the compiler compiles the containing ptype definition. This is so in the equivalence declarations within a ptype definition. In this case, the compiler still creates an omap, and adds it to the TEF ftype creation information maintained in the containing ptype's rep segment. The compiler decides the mapping of the omap as follows. If a TEF name is used as the internal ftype value, the compiler can decide the actual mapping at compilation itself because the mapping is between two external types. It is the use of a TPF name in the containing ptype definition, which causes a complication. The compiler resolves the problem by creating a temporary omap between the indexes used internal to the external ptype of the equivalence declaration, and the internal indexes of the TPF in the containing ptype. This omap is

---

* The CLU library [Lis 77] is an example of a system wide library of ptypes.

** The mapping information required can quite often be contained within the capability itself. In this case, there is no need for an omap object.
also maintained with the TEF ftype creation information. However, in this case, when the TPF of the containing ptype is fixed with an actual ftype, an omap is passed corresponding to the TPF. This omap is combined with the omap in the TEF ftype creation information part, and it is the composite omap which is passed to the CREATE_FTYPE operation which is invoked to create an ftype for the TEF.

The same argument holds for PPFs (procedure parameter ftypes). In this case, the omap capability is passed as a 'pseudo parameter' to the procedure. Since the ftype of procedure parameter objects (PPO) declared as generic is not known at compile time, these parameters also require an associated omap to be passed as a pseudo parameter.
Section II.4  The Required Architectural Features

There are two main features which are required of the architecture. First, the architecture should support the addressing environment of an object-based system. Second, the architecture should recognize ptypes and ftypes as primitive classes of objects in the system.

There are three issues with respect to supporting the addressing environment of an object-based system. The first issue is related to protection. The architecture should ensure that once an object's representation has been associated with a set of procedures, any direct access to the representation should be prevented. This applies for all the objects recognized by the architecture.

The second issue is that at each point of execution, the architecture should restrict the accessibility of objects to a specified set. In an object-based system, each point in execution is identified as the activation of an operation on an object. Thus, accessibility is defined as a set of three address spaces. These are the representation space of the object that is being accessed, the representation space of the currently executing procedure (which implements the operation), and a parameter space which contains the parameters of the operation invocation.

The third issue is the dynamic change of the above accessible sets with each operation invocation. A conventional architecture only provides for the dynamic change of the procedure representation and parameter spaces, at each procedure call. An architecture supporting an object-based environment should also provide for the dynamic change of the object representation space, with each operation invocation.

The second feature of the architecture is that it should recognize ptypes and ftypes as primitive objects of the system. This implies that
the architecture should provide the mechanism for defining a new ptype object as well as a new ftype object. This is in addition to the mechanism for defining elementary objects. In Chapter I, we had discussed the idea of a type extension mechanism. This allows the definition of a class of objects. The architecture has to not only support such a mechanism, but also support another mechanism which allows a class of types to be defined.

To complete the architecture requirements, we will discuss several implementation issues. Every object, whether it is elementary, ftype or ptype, has a rep segment. Since objects are defined in terms of other objects, the total representation structure is a tree structured set of rep segments. The implementation issue concerns the fact that these rep segments are generally small in size. There is some overhead with respect to the separate existence of a segment. This overhead becomes significant if there are many small segments. Therefore the architecture implementation should provide for efficiency in the maintenance of small segments. Furthermore, this implementation should be such that the individual overhead for each segment is reduced by distributing it over many segments.

The second implementation issue arises because of the indirection implicit in the use of a capability. For each operation invocation, the capabilities to the object, the procedure implementing the invoked operation, and the code segment containing the code, each have to be used to access the corresponding representation. The point is that the indirection through capabilities can impose a significant overhead on each operation invocation. Hence, the implementation of the architecture should provide for an efficient form of operation invocation.
Section II.5  Summary

In this chapter, we have followed a top-down approach to specifying an architecture. We started from the programming language point of view and specified a linguistic model which provides a set of constructs for abstraction. We then designed abstraction mechanisms that supported these constructs. The design of these mechanisms conformed to an object-based system. Finally, we specified a set of features required of the architecture, given that it supports an object-based system.

By defining the linguistic model, we established a flexible and generalized set of features for providing abstraction within a programming language. We then showed that, by taking a simple philosophy of approach, we could design the mechanisms to support these features. This philosophy is that by recognizing ptypes and ftypes as objects, we can effectively use the information contained in the representation of these objects. Finally, we proposed that the enforcement of the object-based environment be relegated to the architecture rather than be enforced at the programming language level.

The compiler for a GML based abstraction language can independently compile parameterized type (ptype) definitions, by using the discussed mechanisms. In a pure compiler-based approach to this problem, there are two alternative solutions. The first solution is to use a parameter dependent segment through which all references to the parameters are directed. The code not affected by the parameters is shared in one module. This is the approach used in CLU [Lis 77]. The implementation uses descriptors, which are very similar to capabilities, to address the objects. The parameter dependent segment serves as the indirection table through which the parameter objects are addressed, and is initialized be-
fore execution. The point is that the compiler essentially simulates an addressing mechanism which is capability based. We feel that the system exhibits a 'cleaner' design if the architecture is made responsible for the addressing mechanism.

The mechanisms that we have designed have two additional features which we feel that a CLU-like implementation cannot support. The first feature is that the proposed mechanisms support parameterization to any level. The equivalence declarations allow ftype chaining, such that a type whose parameterization is dependent on some other parameter fixings, may itself be used to fix the parameter of some ptype. This can be repeated any number of times. The second feature is that by explicitly recognizing parameterized types and fixed types as objects, we provide for dynamic creation of types whose parameters may be fixed dynamically.

The second solution for independent ptype compilation is to have a separate instance of the compiled code for each unique set of parameter values [GrG 77]. This solution is practical only in limited applications where there are not too many parameters. Furthermore, this requires that the compiled code be altered for each set of parameters. This is a basic violation of the principle of security. Our main contention is that the compiler should only be responsible for control structuring, name management, and type checking. It should relegate the responsibility of the addressing mechanisms required for object management, to the supporting architecture. We feel that this leads to a more understandable system organization.

We must emphasize that we are not proposing that all compiler-based support of abstraction be eliminated. Recognizing all data types at the architecture level and enforcing their addressing through separate
Descriptors will be grossly inefficient. Many conventional data types can be implemented by the compiler using primitive data elements in a segment. The representation of objects of these types can be collected in a segment and the compiler can generate code to manipulate the data in the segment. In this sense, the compiler treats a segment as a portion of linear memory space. The architecture does not recognize any structure inside the segment.

We envisage the use of the GIM abstraction mechanisms at a level where there is a lot of interaction between users of a computer system. Proprietary programs, shared data bases, security conscious user interactions are some applications which benefit from the use of the type definition facility. In general, any situation where users need to provide services for each other without risking their proprietary or integrity, falls into the class of problems which can be solved by utilizing the abstraction mechanisms.
CHAPTER III

THE VIRTUAL MEMORY DESIGN

Section III.1 Introduction

This chapter presents the design of the virtual memory subsystem. The emphasis in the design is the structuring of the virtual memory subsystem, so that the components of the structure are small and easily understandable. The technique of 'type extension' is used to structure the subsystem. We first present a new viewpoint of this structuring technique which makes it applicable within the virtual memory subsystem. We then adapt this to a refined interpretation which allows us to consider the problem of incorporating the subsystem within the architecture.

The macroscopic structure of the subsystem is that of a linearly ordered set of 'mainstream' abstractions. The real memory abstraction is characterized by four attributes. The virtual memory abstraction is then defined on the basis of the removal of all 'restrictions' on these four attributes. The rationale is that each mainstream abstraction removes a few of these restrictions, until the final virtual memory characterization is obtained. Each of the mechanisms for extending between two successive abstractions is decomposed into a set of primitive abstractions. The salient advantage of this refined viewpoint of type extension is that it allows the design considerations of the subsystem to be separated from the implementation considerations. This chapter focuses on the design considerations of the virtual memory subsystem.

The next section presents the new viewpoint of type extension as applicable to structuring a virtual memory subsystem and develops the
refined interpretation. Section 3 characterizes the virtual memory sub-
system. The three following sections, each discuss a non-trivial main-
stream abstraction. Section 4 discusses the paging abstraction, Section
5 discusses the address space abstraction and Section 6 discusses the
final virtual memory abstractions. Since the final two abstractions are
visible at the architecture interface, part of this discussion is rele-
gated to Chapter V. The last section of this chapter summarizes the
virtual memory design.

Section III.2 The Use of Type Extension in Internal Structuring

a) The basis for type extension

In Chapter I, we had identified type extension as being one of the
techniques used to structure an operating system. A basic characteris-
tic of these systems is that the lower levels of the operating system
are usually unstructured or use a different technique for structuring.
Type extension is only used to structure the higher levels of the opera-
ting system. The basic reason for this is that the implementation of the
type extension mechanism requires some of the features supported by these
lower levels. Specifically, the type extension mechanism requires
unlimited and variable size memory containers for its implementation.
Hence, type extension had always been used to structure the subsystems
which were external to the virtual memory subsystem and relied on the use
of this subsystem.

Janson [Jan 76] was the first to propose the use of type extension
as a structuring technique within a virtual memory subsystem. This was
as part of a reorganization effort directed at structuring the MULTICS
operating system [Sch 77]. Other related research regarding the internal
structuring of virtual memory can be found in [Num 76, Hub 76, Mas 77].
The key aspect of the internal structuring is the departure from the traditional viewpoint of type extension. We will discuss this new viewpoint later in this section.

b) The 'dependency' relationship

The motivation for the use of type extension to structure and modularize a system is twofold. First, a large unstructured system is divided into smaller components and, hence, the total system is made more understandable. However, subdividing into smaller components will not, by itself, improve the system organization, if there is extensive interaction between the components. Type extension modularizes a system such that there is minimal interaction between the components. This is because the resulting modules each manage (define) a class of objects and hence, most of the interaction is internal to the module. It should be realized that type extension is not a rigorous and deterministic methodology to modularize a system, but is a philosophy of approach to the design. Hence, modularizing a system using type extension is, to some extent, subjective.

The advantage of type extension is that it imposes a partial ordering based on the interaction relationships between the modular components. The advantage of a 'partial ordering' is that it allows an orderly verification of the system design. The specification of the design is also made easier. The question is what should the interaction relationship be? Parnas [Par 76a] suggested the 'uses' relationship. Since the basis of type extension is the definition of new classes of objects (that is, type manager modules) using existing classes of objects, the 'uses' relationship embodies this idea. A module A uses another module B, if A is defined in terms of B and perhaps some other modules. In this case B pro-
vides a service for A.

A stronger interaction relationship is the 'dependency' relationship. It is stronger in the sense that it includes the 'uses' relationship. Essentially, a module A 'depends' on module B if verifying the correctness of A requires verifying the correctness of B [Jan 76]. Another way that this may be stated is that the relationship A 'depends' on B, implies that the validity of the specification of module A requires that some assumptions about the specification of module B hold. As we will see in the next sub-section, the 'uses' relationship identifies only one form of the 'dependency' relationship between two modules. The 'uses' form of the dependency relationship is the most apparent.

c) A modified viewpoint of type extension

Type extension serves as both a structuring as well as a modeling technique in system design. It can serve as a modeling technique because it allows the presentation of a macroscopic specification of the system in terms of its modules, without including the microscopic details internal to the modules. When used within a virtual memory subsystem, one has to depart from the traditional viewpoint of type extension for the reasons mentioned earlier. Janson [Jan 76] first proposed this new viewpoint of type extension and we will now review it.

The basic idea in type extension is that the system which is being structured is to be divided into modules, where each module defines a type manager. A type manager manages a set of objects by defining a set of operations. Objects in the set may only be manipulated by these operations. An object is either primitive or defined in terms of some other objects. The latter kind of objects are called extended objects. The four main aspects with respect to which the new viewpoint of type ex-
tension differs from the traditional viewpoint are—the lifetime of objects, the representation maps for objects, the protection mechanism for objects, and the implementation of operations for objects.

In the traditional viewpoint of type extension, there is an essentially infinite supply of objects. Any number of objects may be created and can exist at a time. This was possible because of the fact that the underlying virtual memory provides unlimited space for the existence of objects. In the new viewpoint, there are two kinds of objects. The first kind is objects which are of unlimited supply, called the create/destroy (C/D) kind of objects. The other kind is objects of limited supply, called the allocate/free (A/F) kind. The latter kind of objects reflects the limited resources which are available to the implementation of the virtual memory subsystem. Such objects cannot be created or destroyed, but only allocated for some time and then freed.

The second difference between the two viewpoints concerns the representation map of an object. The representation map for an object identifies the component objects which constitute the extended object. In the traditional viewpoint, the map is supported by the virtual memory and hence can be of variable and unlimited size. This implies that the component set of an object can also be of variable size. In the new viewpoint, the application of type extension is in an environment where the variable size map containers are not available. In fact, the basic purpose of the subsystem which is being structured, is to perform a mapping function between primitive A/F kind of objects and the user visible C/D kind of objects. Hence, the representation maps of the objects supported by a type manager, constitute most of the object representation. The point is that each type manager module cannot assume a universal map con-
tainer, but has to invent its own form of map abstraction.

The third difference arises because of the different protection requirements between the two viewpoints. There are two forms of protection required in a system structured by type extension. The first is vertical protection between the users of the objects belonging to some type, and the type manager managing those objects. The second is horizontal protection between the users of objects themselves. In the traditional viewpoint, capabilities provide for both forms of protection, much in the same way as discussed in Chapter II. However, in the new viewpoint of type extension, the overhead of a capability form of protection at execution time is significant, when used in internal structuring. The alternative is to use an implicit form of protection using formal verification techniques. This is possible in internal structuring, since the users of objects are themselves modules in the internal structure. Capability based protection is only required at the user interface.

Finally, the fourth difference concerns the implementation of the operations defined by a type manager. In the traditional viewpoint, operations are implemented as callable procedures. The entity performing the execution is still the original process (representing the external user) which invokes the operation on the object. However, there is nothing intrinsic in type extension which requires the operations of a type manager to be implemented as callable procedures. The new viewpoint of type extension recognizes this and allows the operations of a type manager to also be implemented as concurrently executing processes. Communication with operations implemented as procedures is through the regular parameter passing mechanisms, while communication with operations im-
plemented as processes is through messages and/or synchronization mechanisms.

Earlier we had stated that the dependency relationship is stronger than the 'uses' relationship, since it includes the latter. Janson identified five forms of the dependency relationship in a system structured by the new viewpoint of type extension. These are:

1) **Component.** Every module depends on the type managers which manage its component objects. This is the form which corresponds to the 'uses' relationship.

2) **Map.** A module depends on the type manager which manages the map container used by the module.

3) **Program.** A module depends on the type manager which provides the storage space for the programs implementing the module's operations.

4) **Address Space.** A module depends on the type manager which enforces the addressing environment for the programs of the module.

5) **Interpreter.** A module depends on the type manager which interprets the programs of the module. This type manager also provides the synchronization facilities, necessary for the module execution.

In the traditional viewpoint, the latter four forms of dependency are not explicitly recognized. The virtual memory subsystem implicitly provides the containers for maps and programs, and enforces the addressing environment. The 'process' type manager implicitly provides the interpreter.

We have reviewed the new viewpoint of type extension, proposed by Janson. This new viewpoint is applicable to the internal structuring of
the lower levels of an operating system, specifically the virtual memory subsystem. In our case, the emphasis is on the architectural incorporation of the virtual memory subsystem. Therefore, we will adopt a refined interpretation of type extension.

Type extension imposes a partial ordering on the modules composing the system which is being structured. We have seen that if the system being structured is a low level component of the operating system, we then have to adopt a different viewpoint of type extension. The key issue to this different viewpoint is the recognition that type extension is a conceptual basis for the structuring and does not have to have a mechanism to enforce the structuring. There is still a deficiency with respect to this new viewpoint, in that there is an intermingling of the design and implementation considerations of the subsystem.

The design considerations of a subsystem arise out of the 'component' relationships between the abstractions composing the subsystem. In the case of the virtual memory subsystem, the primary role of the subsystem is to provide mapping functions. Therefore, the 'map' relationships between abstractions also influence the design considerations. For convenience, this relationship is considered to be part of the component relationship. Implementation of each of the component abstractions causes the three other forms of the dependency relationship. These relationships give rise to the implementation considerations. We call the design considerations as the abstraction issues and the implementation considerations as the meta-implementation issues in the design of the architecture.

We are interested in the use of type extension as a means of structuring certain low level operating system components, with a view to in-
corporating these components into the architecture. In order to do so, we want to model the subsystem so that we can first concentrate on the abstraction issues of the design and then consider the meta-implementation issues independently. We will, therefore, adopt the following viewpoint of type extension.

- Rather than structuring the subsystem as a partially ordered system of type managers (abstractions), we will recognize a linearly ordered system of 'mainstream' abstractions.
- We will associate such a set of mainstream abstractions with each subsystem, that is to be incorporated.
- We will decompose the extension mechanism between any two successive abstractions in every mainstream, into a partially ordered set of primitive abstractions. (Note that this makes the total structure partially ordered).
- We will identify and order the interactions between abstractions in different mainstreams.

We identify a mainstream abstraction with the virtual memory subsystem and another with the virtual processor subsystem. We will also consider a simple version of a file system mainstream. The virtual processor mainstream includes the message communication mechanisms.

The separation of the abstraction and meta-implementation issues of a subsystem, enables us to remove the linear ordering of implementation methodology, characteristic of conventional architecture design. In such designs, the linear ordering manifests itself as implementation through logic circuits at the lowest level (defines the micro machine); followed by implementation through microprogramming (defines the hardware processor); followed by implementation through low level programming (defines
the operating system primitives); and finally by implementation through high level programming (defines the user program). We are not just redefining the term 'architecture' to include the low level operating system components. Rather, we are defining the term 'architectural implementation' in the sense that the architecture designer may make independent implementation decisions with respect to each component of the subsystems which are incorporated. In the rest of this chapter, we will discuss the abstraction issues of the virtual memory subsystem. Type extension is used as the structuring and modeling technique.

Section III.3 Characterization of Virtual Memory

We have informally defined a virtual memory subsystem as providing a mapping function between an unlimited number of virtual memory objects at the user level and a limited number of real memory objects at the level of primitive resources. We will now characterize the real memory and virtual memory abstractions.

A memory abstraction can be characterized in terms of the 'address space' and 'memory container' objects that it supports. A memory container object consists of a set of memory cells. A memory abstraction defines a set of 'physical' addresses which uniquely identify the memory containers on a one-to-one basis. An address space object contains a set of memory container objects. This set may be a subset of the total number of memory container objects supported by the abstraction, or may be the whole set itself. The purpose of an address space is to allow the memory containers contained by it to be addressed by 'logical' addresses, which are local to the address space. The address space object performs the translation between the logical and physical addresses.

Real memory contains a fixed and limited number of real memory cells,
For simplicity, we will treat the real memory abstraction as consisting of a set of memory containers, each of which contains a small, fixed number of real memory cells. These memory cells are called 'frames'. The real memory abstraction supports a single implicit address space for which the set of logical and physical addresses are equivalent. This address space contains all the frames of the real memory abstraction and there is only a limited number of these frames.

Characterization of a memory abstraction is based on the following characteristics:

1) The number of address space objects.
2) The addressability of an address space, in terms of the number of memory containers that an address space can contain.
3) The total number of memory container objects available.
4) The size of the memory containers, in terms of the number of memory cells that a memory container object consists of.

On this basis, the real memory abstraction can be characterized as:

1) A single (implicit) address space.
2) A small, fixed number of memory containers (frames) contained by this address space.
3) A small, fixed total number of memory containers. (Since there is only one address space, the numbers in 2 and 3 are equal.)
4) Each memory container contains a small, fixed number of memory cells.

We can now characterize* the virtual memory as the abstraction pro-

* We owe part of the credit for this characterization to Hunt's model [Hun 76] of a virtual memory subsystem.
duced by removing each restriction in the above characterization:

1) An unlimited number of address spaces.

2) Each address space can contain an unlimited number of memory container objects.

3) An unlimited total number of memory container objects.

4) Each memory container object can contain an unlimited number of memory cells.

The use of the term 'unlimited' is in the sense of the number of objects varying from a single object to an arbitrarily large number of objects.

The purpose of the virtual memory subsystem is to extend from the real memory abstraction to the virtual memory abstraction. The subsystem is structured as a set of linearly ordered mainstream abstractions. Each successive mainstream abstraction removes some of the restrictions from the real memory characterization, until finally the virtual memory characterization is obtained.
Section III.4  The Paging Abstraction (VM2)

The best way to describe a type extension (object) based system is to formally specify each of the modular components of the system. Formal specification is still an ongoing area of research and specification techniques are neither widely available nor easily usable as, say, programming languages are. Most current specification techniques require semi-automated tools. Since the main thrust of this thesis is not in formal specification, we have used a semi-formal means of specification. We will describe each type manager in terms of the operations supported by that abstraction. Each operation requires a set of input parameters and returns a set of output parameters. We specify the type of each such parameter for every operation. Since many of the abstractions perform mapping functions, we specify their internal structure. This is more from the point of view of a suggestion, so as to help understand the specification through the operations of the abstraction. The conventions followed, with respect to the semi-formal specifications, are discussed in Appendix 1.

For two of the important virtual memory abstractions, we also specify an 'abstract implementation'. An abstract implementation is an 'abstract' program which uses standard control constructs for control sequencing and the primitives of which are the operations defined by the component abstractions. In our viewpoint of type extension, the abstract implementation of a mainstream abstraction consists of the operations defined by the next lower level mainstream abstraction and the set of operations defined by each primitive abstraction composing the extension mechanism. The abstract implementation provides valuable insight into the internal design of an abstraction. The concept is best illustrated
by applying it to the specification of the virtual memory structure.

a) The functionality and role of the Paging Abstraction

The functionality of an abstraction is defined as the operations provided by that abstraction. The purpose of the paging abstraction is to provide a large number of memory container objects. It multiplexes the small number of frame objects between a large number of page objects. Page objects are still limited (i.e., fixed) in number. This differs from the conventional notion of a page object as being essentially unlimited in number. We will remove the limitation on the number of objects, through a higher level mainstream abstraction. This viewpoint of a page object is quite a natural one to adopt. In fact, it has been independently shown [Mas 77], as part of the Multics restructuring effort, that such a viewpoint should be adopted within a virtual memory subsystem. We will later establish a practical basis for this viewpoint.

A page is a logical object and, when associated with a physical object like a frame, its contents can be physically contained by the frame object. However, when a frame is not associated with a page, its contents need to be stored in some other physical object. This physical object which serves as a backup is called the 'home' object (borrowing the Multics terminology). A home object is equivalent to a secondary storage block. We have introduced three types of objects—frame, page, and home. Each type has its respective type manager abstraction. Frames are managed by the real memory abstraction, pages by the paging abstraction that we are designing, and homes by the secondary storage abstraction, which is external to the virtual memory mainstream.

Each type of object is identified by its respective 'name_type'. The set of 'frame names' identify the set of frame objects (by a one-to-
one correspondence). The set of 'page names' identify page objects, and
the set of 'home names' identify home objects. It is important to real-
ize that a 'name_type' is a type distinct from the type of the object
class it identifies. Consequently, name_types may be used in abstrac-
tions without the implication that the corresponding object is also ac-
cessed. This idea is essential to understanding the dependency relation-
ships within the virtual memory structuring. Name_types are managed by
map abstractions, in that the map abstraction maintains a set of attri-
butes associated with each name_type object. (We will use the term
'name' to refer to a name_type object.) Some of these attributes may be
other names and, in fact, can be of the same name_type itself (as when
maintaining a linked list). All name_types have a special 'null' name
defined for them. A 'null' name does not identify any object. Since a
virtual memory component essentially provides a mapping function between
two sets of objects, we will observe that a majority of the primitive
abstractions composing extension mechanisms are map abstractions managing
name_types.

The functionality of the mainstream paging abstraction is defined in
Fig. 3.1a. (The reader is strongly encouraged to familiarize himself
with the specification conventions discussed in Appendix 1). Each opera-
tion is specified in terms of the types of its input and output parameters.
The use of a name_type in an operation has two possible interpretations.
One, this may signify that an object of the type identified by the
name_type is expected as a parameter. Alternatively, a name belonging
to the name_type is itself expected as a parameter*. To differentiate

* The confusion arises because of the type based system philosophy that
objects are passed by reference in parameter passing. Names are the
only objects not passed by reference.
READ_VM2_PG (p_n", offs, elem)
WRITE_VM2_PG (p_n", offs, elem)
PZERO_VM2_PG (p_n")
UPDATE_VM2_PG (p_n", zf)
PSET_VM2_PH (p_n", h_n, zf, p_n+)
SEARCH_VM2_PG (h_n, p_n+, p_n+)
GET_VM2_HN (p_n", h_n)
QZERO_VM2_ZF (p_n", zf)

**Fig. 3.1a Operations of the Paging Abstraction (VM2)**

READ_VM0_FR (f_n", offs, elem)
WRITE_VM0_FR (f_n", offs, elem)
PZERO_VM0_FR (f_n")

**Fig. 3.1b Operations of the Real Memory Abstraction (VM0)**
the former case, we will use the ' " ' notation to denote that an object identified by the name is the actual parameter. In all other cases, a name type in an operation specification signifies that the identified object is not manipulated and only a name belonging to the name_type is used.

The operations defined by the paging abstraction consist of a READ operation and a WRITE operation to access the elements logically contained by a page; a PZERO operation to set all the contents of a page to zero; a PUPDATE operation which guarantees that the contents of the associated home is updated with the page's logical contents; a PSET operation which changes the page to home association; and three interrogative operations SEARCH, GET_HN, and QZERO. Before discussing all these operations, we need to clarify one issue with respect to the lower level abstraction from which the paging abstraction extends.

b) The partitioned real memory abstraction (VM1)

There is an intervening mainstream abstraction between the paging and real memory abstractions. This serves to partition the set of frames. The paging abstraction uses only one of these partitions. The reason for the other partitions is that the implementation for many of the other abstractions may require portions of real memory*. One example is the lowest level abstraction in the virtual processor mainstream, which may require real memory to store process states. The exact number and size of the partitions is decided when the implementation for all the abstractions are decided. The partitioned real memory abstraction has usually a trivial implementation and, hence, no discussion is required as to its

* This abstraction serves the same purpose as 'core segments' in the restructured Multics system.
extension mechanism.

The paging abstraction treats one of the real memory partitions as a real memory abstraction. The partitioned real memory abstraction requires an additional 'partition name' parameter. For the sake of simplicity, we will treat this parameter as an implicit parameter in the abstract implementations of the paging operations. Therefore, the partitioned realy memory abstraction will be transparent between the real memory and paging abstractions. The operations of the lower level abstractions from which the paging abstraction extends, are specified in Fig. 3.1b.

c) The operations of the Paging Abstraction

The paging abstraction extends from a small limited number of frame objects to a large but still limited number of page objects. The set of home objects are essentially unlimited in number and only a subset of them are associated with pages. The paging abstraction makes no assumptions with respect to the number of home objects. The paging abstraction guarantees that no two pages are associated with the same home. Once a home is associated with a page, the logical contents of the page are initialized with the physical contents of the home. Thereafter, the paging abstraction maintains the logical contents of the page, either in a frame or in a home. The READ and WRITE operations guarantee access to the logical contents of the page. There is a more subtle performance implication in that after the first READ or WRITE access, subsequent accesses in 'close proximity' are guaranteed a much improved access performance (equivalent to directly accessing a frame). Hence, the paging abstraction provides for a much larger set of page objects, but with an average access performance close to that of frame objects.
The paging abstraction provides a PZERO operation, which sets the total contents of a page to zero. A zero page does not require a physical existence in a frame or home. A page may maintain its contents physically, either in a frame or in a home; or just logically as for a zero page. Thus, a home is not necessarily consistent with the logical contents of its page. The PUPDATE operation ensures that the physical contents of the home is made consistent with the page's logical contents. The PSET operation is provided to change the page to home association. This operation requires a flag parameter, which identifies whether the page is to be set to zero. The paging abstraction guarantees that no two pages are simultaneously associated with the same home. The PSET operation returns with a non-null page name, if a page has already been associated with the new home. If a null value is returned, it implies that the new home has been uniquely associated with the specified page. Finally, three interrogative operations are provided. The SEARCH operation returns a non-null page name, if a page is associated with the specified home. The QZERO operation returns a 'zero flag' which indicates whether a page is logically zero. The GET_HN operation returns with the name of the home associated with a page.

The abstract implementation for each of the paging abstraction operations is specified in Appendix 2a.3. Two map abstractions compose part of the extension mechanism for the paging abstraction. These are primitive abstractions and their specification is given in Appendix 2a.1. The efficiency of the internal implementation of these abstractions is a meta-implementation issue, which will be discussed in a later chapter. The paging abstraction also uses abstractions in other mainstreams. The VP2 (base processor) abstraction in the virtual processor mainstream pro-
vides the synchronization operations required in the abstract implemen-
tation of the paging operations. The SS (secondary storage) abstraction
manages home objects, and is part of the file system mainstream.

We encourage the reader to study the semi-formal specification of
the paging abstraction in Appendix 2a. The abstract implementation pro-
vides an insight into the interaction internal to the paging abstraction.
d) The structure of the extension mechanism

The structure of the extension mechanism for the paging abstraction
is illustrated in Fig. 3.2. The extension mechanism consists of two map
(data) abstractions and five functional abstractions. Two external
mainstream abstractions, which are used, are the VP2 and SS abstractions.
The paging abstraction (VM2) extends from the real memory (VM∅) abstrac-
tion (but implicit through the VM1 abstraction). The arcs between the
nodes representing abstractions, implies that the abstraction at the
tail of the arc 'uses' the abstraction at the head of the arc by invoking
an operation defined by the latter abstraction. The dependency relation-
ship is transitive and, hence, the paging abstraction depends on all the
ten abstractions. In the abstract implementation of the paging opera-
tions, we observe that the operations provided by these ten abstractions
are the only ones invoked.

Functional abstractions provide only one operation, corresponding
to the function it implements. The B_PL (block page lock) abstraction
ensures that a process secures exclusive access to a page. The U_PL
(unlock page lock) abstraction releases this exclusive access. The F_C
(frame claimer) abstraction is used by the paging abstraction when it
requires a frame allocation on a page fault. Frames are deallocated
(i.e., freed) by the F_F (frame freer) functional abstraction. It uses
Fig. 3.2 The Structure of the VM2 abstraction
the $F_R$ (frame replacement) functional abstraction to decide the frame which is going to be deallocated.

All the functional abstractions in Fig. 3.2, except the frame freer ($F_F$), are directly invoked by one or more abstractions. At this stage, there is no decision made as to the nature of their implementation. However, the decision to implement the frame freer as a separate process is made at this stage. The frame freer is activated whenever the number of free frames goes below a threshold level, and it remains active and keeps freeing frames until the threshold level is reached again. Since this freeing of frames occurs independently, it seems reasonable that the frame freer is implemented as a separate process which executes concurrently with respect to the users (processes) of the paging abstraction. In fact, this was suggested as a means of improving performance in the restructured Multics system [Hub 76]. Note that we have not yet made a decision with respect to the efficiency of implementation of the frame freer. There is no direct invocation of the frame freer. Since the frame freer is a separate process, it is indirectly invoked by activating it through the synchronization operations defined by the $VP2$ abstraction.

The two map abstractions, $FM$ (frame map) and $PM$ (page map), manage 'frame names' and 'page names', respectively. The $PM$ abstraction maintains for each page name, the following set of attributes—the associated home name, a frame name if a frame is allocated to the page, a core flag which indicates whether or not a frame is allocated to the page, a zero flag which indicates whether the page is logically zero, and a page lock which is used to ensure exclusive access to a page. The operations provided by the $PM$ abstraction allow access to these attributes. There are
two operations which provide a 'test and set' function with respect to
the page lock. There is also an operation which can be used to search
for the page associated with a given home.

The FM abstraction serves two purposes. First, it maintains a set
of attributes for each frame name. Second, it also maintains the asso-
ciative map which contains a set of frame name to page name associations.
The FM abstraction provides operations to access the attributes of a
frame name, and another set of operations defined for associative access
on the page name. The FM abstraction maintains a list of free frames
and provides operations to add a frame or remove a frame with respect to
the free list. The abstract implementation relies on a 'modified LRU'
page replacement algorithm (provided by the F_R functional abstraction)
which requires that the name of the last freed frame be maintained. The
FM abstraction is entrusted with this responsibility. The details of
the FM and FM abstractions are discussed in Appendix 2a.1.

The two external mainstream abstractions used by the paging abstrac-
tion are the SS and VP2 abstractions. The SS (secondary storage) ab-
straction manages home objects. It provides operations (Fig. 3.3a) to
transfer contents between a home and a frame. The SS abstraction itself
depends on the real memory abstraction, in order to access the contents
of a frame.

The VP2 (base processor) abstraction provides a set of synchroniza-
tion operations (Fig. 3.3b). In addition, it provides a more subtle
service in that the processes which are considered to be the users of
the paging abstraction, are themselves objects managed by the VP2
abstraction. The user processes and the process used by the frame freer,
all execute concurrently, according to the abstraction provided by VP2.
FETCH_SS_HM (h_n", f_n)
STORE_SS_HM (h_n", f_n)
HZERO_SS_HM (h_n")

**Fig. 3.3a** The Secondary Storage (SS) Operations

BLOCK_vp2_iq (PQ, p_n)
UNBLOCK_vp2_iq (PQ, p_n)
SETS_vp2_iq (PQ, p_n)

ENTER_vp2_msem (FFL)
EXIT_vp2_msem (FFL)
EXCHANGE_vp2_msem (FFL)
QEMPTY_vp2_msem (FFL, ef)

WAIT_vp2_cq (FFQ)
SIGEXIT_vp2_cq (FFL, FFQ)

**Fig. 3.3b** The VP2 Operations used by VM2
We will discuss the VP2 abstraction in more detail in the next chapter.

One of the main purposes of the abstract implementation is to specify the interaction between the component abstractions of a mainstream abstraction. A careful scrutiny of the abstract implementations in the appendix provide an understanding of the mechanistic details of the paging abstraction.

e) The synchronization considerations

One of the important considerations in the design of the paging abstraction is the provision for concurrent users (processes) of the abstraction. The paging abstraction cannot make any assumptions about the relative efficiencies of these users. In conventional virtual memory systems, a global page lock satisfies the synchronization requirements. This is very inefficient as only one user can make use of the paging abstraction at one time. Synchronization mechanisms based on monitors have also been suggested [Hoa 73, Sax 76], but these require the support of a large number of monitors, each of which requires considerable overhead (in space and time). The synchronization mechanisms that we have proposed, conform to the model of an object based system, while satisfying the requirements of maximum concurrency.

The synchronization requirements are satisfied by a set of nine operations supported by the VP2 (base processor) abstraction (Fig. 3.3b). The synchronization requirements arise from the following three considerations:

1) Once a page is being manipulated by an operation defined on it, the paging abstraction requires that any simultaneous manipulation on the same page be blocked until the current manipulation is completed.
2) The frame freer needs to be synchronized with respect to the users of the paging abstraction.

3) Invoking an operation of the SS (secondary storage) abstraction causes a delay, while the secondary storage transfer takes place.

The requirement arising out of the first consideration is satisfied by the 'page lock' maintained for each entry in the page map (PM), and three 'indexed Q' operations of VP2. An indexed Q is a queue of processes associated with a 'name_type'* . All the three indexed Q operations require a name of that name_type as a parameter. The BLOCK operation causes the invoking user (process) to be blocked in the queue. The page name, passed as a parameter, is associated with the user identity in the queue. When a user invokes the UNBLOCK operation, passing a page name as a parameter, the VP2 abstraction causes the first user, which is associated with the same page name, to be unblocked. The invoking user is not affected. The BLOCK-UNBLOCK relationship is based on a binary semaphore for each page name, which prevents the loss of an UNBLOCK signal. The SETS_IQ operation is used to set the binary semaphore associated with a page name to 1.

The two functional abstractions, BLOCK_PAGELOCK (B_PL) and UNLOCK_PAGELOCK (U_PL), use the three VP2 operations and the 'test and set' operations of the Page Map abstraction to effect the page synchronization (Appendix 2a.3). As shown in the abstract implementation, the BLOCK_PAGELOCK functional abstraction uses a TSTC operation (defined by

---

* The 'PQ' parameter identifies the indexed queue used as corresponding to the 'page name_type'.
FM) which returns both the core flag, and the original state of the page lock. The invoking user busy waits only if the core flag is set, indicating that some other user will soon release the page. Otherwise, the invoking user invokes the BLOCK indexed Q operation and blocks itself. The UNLOCK_PAGELOCK functional abstraction releases the page lock and invokes the UNBLOCK operation to unblock the first process blocked on the same page. After a page has been successfully locked, the SETS_IQ operation is used to set the associated binary semaphore.

The second synchronization requirement is satisfied by the four monitor operations and the two conditional Q operations of VP2. The frame freer (F_F) functional abstraction is synchronized with respect to the users of the paging abstraction, since it is activated by the users. The frame freer initially invokes the WAIT operation (defined by VP2) and waits in a conditional Q (called FFQ), which contains only one process. The conditional Q is associated with a monitor (called FFL). Each user using the frame claimer (F_C) functional abstraction causes the entry into the monitor (FFL). A monitor consists of a queue of processes waiting to enter the monitor, and one process which is in the monitor. A process enters the monitor by invoking the ENTER operation and exits from the monitor by invoking the EXIT operation. Only one process, among all the processes which have invoked an ENTER operation, is active in the monitor. The rest of the processes are blocked in the monitor queue. The EXIT operation causes the first process in the monitor queue to become active. Within the monitor, the frame claimer functional abstraction activates the frame freer by invoking the SIGEXIT operation, if it finds that the number of free frames has gone below the threshold level. The SIGEXIT operation causes the invoking process to exit from the moni-
tor, but instead of activating the first process in the monitor queue automatically, it first checks if the frame freer is blocked in the associated conditional Q (FFQ). If this is so, the frame freer is activated, otherwise (as we will show) the frame freer is anyway the first process in the monitor queue and is therefore activated. Thus, the SIGEXIT operation always causes the frame freer to be reactivated within the monitor.

The frame freer is initially activated from the FFQ conditional Q. Once activated, it controls the scheduling of the FFL monitor (refer to Appendix 2a.3). As long as there are free frames available and processes are blocked in the monitor queue, the frame freer releases the monitor to the first process, which then allocates a frame. This release occurs by invoking the EXCHANGE operation, which causes the invoking process to exchange places with the process at the head of the monitor queue. Thus it follows that the frame freer is either at the head of the monitor queue or in the conditional Q. In the former case, it has already been activated from the conditional Q and taken over control of the monitor scheduling. As shown in the abstract implementation, anytime there are no processes blocked in the monitor queue or if there are no free frames, the frame freer goes about freeing a frame. After this is complete, it comes back to check for blocked processes. Finally, when the frame freer has freed frames up to the threshold level, it invokes the WAIT operation and waits in the conditional Q. The frame freer uses the interrogative BPR operation, QEMPTY, to check if any processes are blocked in the monitor queue.

The final synchronization consideration arises out of invoking secondary storage (SS) operations. In practical systems, the delay due
to a secondary storage transfer is a significant overhead and usually causes the invoking process to be blocked. We will postpone discussion of the delay caused by invoking SS operations to the next chapter. At this stage, we will recognize that invoking a SS operation can cause blocking of the invoking user (process).

Finally, one subtle aspect of synchronization, common to all the abstract implementations, is that each operation on a primitive abstraction is considered to be atomic. This implies that a primitive abstraction may execute only one operation at a time. Thus, there is a form of blocking between users simultaneously invoking operations defined by the same primitive abstraction. We will discuss this issue in Chapter 4.

f) An overview

In this section, we have presented the design of the paging abstraction. The design conforms to the model of a typed object based system. Most of the details of the design are explained through the abstract implementation. The paging abstraction provides a set of eight operations, defined for the manipulation of page objects. Page objects are much larger in number than frame objects, but closely match the access performance of frame objects, subject to some constraints.

The paging abstraction guarantees operationality. This means that when one of its operations is invoked by a user, it guarantees that the function defined by the operation is carried out. For example, a READ operation on a page guarantees that the contents of the page are read. Similarly, a PZERO operation guarantees that all the page contents are set to zero. The paging abstraction also guarantees temporal spatiality. This implies that once an operation invocation has been executed, any
subsequent operation invocation sees only the changed state caused by the first operation. For example, a READ operation following a PZERO on the same page, always returns a zero element. Note that the synchronization within the paging abstraction, makes no guarantees about simultaneous operation invocations on the same page. The internal states remain consistent, but say a READ operation simultaneous with respect to a PZERO operation on the same page, may or may not return a zero element. However, this is a synchronization problem which is external to the paging abstraction.

By our modeling of the external VP2 abstraction, we have indicated how the dependency loop between the virtual memory and virtual processor subsystems may be broken. Our approach to the problem has followed the line of existing solutions [Sax 76, Ree 76]. The dependency loop is resolved by interleaving the mainstream abstractions in the virtual memory and virtual processor subsystems. This structure becomes apparent when we consider the structure of the total architecture.

Our viewpoint of requiring the paging abstraction to support a large but fixed number of page objects has a practical basis. If the number of page objects is to be unlimited, then the page map (PM) abstraction has to maintain a set of attributes for an unlimited number of pages. However, the resources (mainly in terms of real memory) available to the page map implementation is small in number. Therefore, it makes sense for the paging abstraction to first remove this restriction in number, and let the restriction of a fixed number of objects be removed by a higher level abstraction. An additional advantage of this approach is that the backup objects used by the paging abstraction may actually be objects managed by an intermediate paging device. Thus, al-
though the number of home objects is large, there can be a one-to-one correspondence between paging device objects and page objects. The use of a paging device will considerably reduce the delay caused by invoking a secondary storage (SS) operation.

The inherent concern in the design of the abstract implementation is that no assumptions can be made about any external synchronization between the users of the paging abstraction. This is because the paging abstraction allows simultaneous invocations of operations by its users. More important, it allows for the procedure activations corresponding to the invocations to execute concurrently. Therefore, the implementation has to enforce its internal synchronization to guarantee correctness of operation. Primitive abstractions, on the other hand, enforce the restriction that only one invocation of an operation can be executed at any time. This fact is to be used in making any assertions about the synchronization properties of an abstraction.

The abstract implementation assumes that checks are made to ensure that actual parameters for operation invocations conform to the type of the formal parameter. As we had pointed out, these checks are part of the static verification process of the total design. The type checking for invocation of operations defined by primitive abstractions is part of the verification of the abstract implementation of the mainstream abstraction.
Section III.5  The Address Space Abstraction (VM.4)

a) The functionality and role of the Address Space Abstraction

The paging abstraction served to increase the small number of memory container objects. Still, the abstraction that it provides places a limit on the number of these objects. The address space abstraction removes this limit so that an unlimited number of objects can exist. The address space abstraction also removes the restriction of a single address space by providing an arbitrarily large number of address spaces. Finally, the address space abstraction introduces the concept of sharing between objects. Sharing of objects is through a common home and can be local to an address space or across address spaces. Recollect that the paging abstraction did not allow sharing, as two pages could not share a home.

A significant point of difference between the paging and address space abstractions is the handling of faults. In the paging abstraction, a page fault corresponds to the access of a page which does not have a frame associated with it. This is handled internal to the paging abstraction and any access operation guarantees operationality. The address space abstraction is similar to the paging abstraction in that it multiplexes the large but limited number of 'page' objects between an unlimited number of 'block' objects. The difference is that when a block is accessed and does not have a page allocated to it, the access operation returns as a fault. The address space abstraction makes visible to its users, the difference between an 'active' block (which has a page allocated to it) and an 'inactive' block (which does not have a page allocated to it). The users of the abstraction are entrusted with the responsibility for converting between the two states. The rationale be-
hind this is that at the lower levels of abstraction, a universal policy can satisfy all users, but at the higher levels, users should have the provision to enforce individual policies. Hence, the address space abstraction just provides the mechanisms with which the policies can be enforced.

The operations of the address space abstraction also distinguish between 'active' and 'inactive' address spaces. An address space has to be active before any of its blocks can be activated and accessed. An address space is now an object which contains a variable size set of blocks. Addresses to identify the block objects, managed by the address space abstraction, are two tuples of the form, 'address space name, block name'. The address space abstraction provides a unique page limiting feature. With this feature, a limit can be set on the total number of pages which can be allocated to the active blocks in an address space.

Sharing between blocks is through a common home. A home serves as a backup physical object for the address space abstraction also, but the abstraction leaves the actual access of homes to the paging abstraction. Users of the address space abstraction specify sharing among blocks through the use of a common home name. Sharing can be local to an address space or across address spaces. The address space abstraction has to ensure that two active blocks, which share a home, are allocated the same page. This is because the paging abstraction does not provide for sharing between pages, for obvious reasons of consistency. If the two active sharing blocks are in the same address space, then this is counted as only one block being active, with respect to the page limit. In the abstract implementation, a set of 'page indexes' is provided for each active address space. The cardinality of this set
is equal to the page limit set for the address space. Every active block has to be associated with a page index in the address space. Two active sharing blocks in the same address space share both the page index and the page. Two active sharing blocks in different address spaces are allocated distinct page indexes in their respective address spaces, but share a page.

We have given a general description of the address space abstraction. As in any abstraction definition, the exact nature of the interface is best understood by specifying the set of operations (functionality) defined by the abstraction. The details of the internal structure are illustrated by an abstract implementation for the abstraction's functionality set. This is given in Appendix 2b.4.

b) **Partitioned Paged Memory Abstraction (VM3)**

The partitioned paged memory abstraction is an intervening abstraction between the paging and address space abstractions. This serves a purpose similar to the partitioned real memory abstraction (VM1). Abstractions in other mainstreams, which are not dependent on the paging abstraction, can use a partition of pages. A case in point, is the 'home name manager' (HNR) abstraction, which maintains information about all the homes in the system. Since there are a large number of homes, this data base may be maintained in backup storage, but accessed through the set of pages in a page partition. Note that the paging manager cannot itself use the HNR abstraction. Another use of a page partition is to store intermediate process states. In our case, there was no need for an intermediate process manager, which is above the VP2 abstraction, but below the address space abstraction, in the system's partial ordering structure. Finally, any of the primitive abstractions, composing the
extension mechanism for mainstream abstractions above the paging abstraction, can use a page partition for its implementation. Once again, as far as the address space abstraction is concerned, it treats the partition of pages available to it as the paging abstraction and extends from it. Therefore the partitioned paged memory abstraction is transparent between the paging and address space abstractions.

c) The operations of the Address Space Abstraction

The set of operations defined by the address space abstraction is shown in Fig. 3.4. This abstraction supports 'address space' and 'block' objects. The operations of Fig. 3.4 manipulate either an address space or a block object (the ' " ' notation signifies that an object of that type is manipulated). However, since an address space is comprised of a set of block objects, manipulation of a block object always causes an implicit manipulation of the containing address space object. The crucial difference, between the operations of this abstraction and those of the paging abstraction, is that any of the operations of Fig. 3.4 can return as a fault, with the fault flag set. We have introduced additional notation to include fault returns in an operation specification. The interpretation of the various fault flags is specified in Fig. 3.5. Apart from using name_types to specify parameters, the operation specifications include a different kind of parameter—len_max, plim_max, len_curr, and plim_curr. These are parameters associated with each address space object. All these parameters are integers, but are defined for a certain range, that is, they are bound by a maximum value*. The maximum value for the len_max and plim_max parameters are decided on a system wide

* The 'null' value for these parameters is zero.
basis. The len_curr and plim_curr bounds are decided by the len_max and plim_max values, respectively. Violation of the bounds causes the setting of the corresponding fault flags (Fig. 3.5).

The operations of the address space abstraction have been grouped into eight sets. The first set contains the standard memory access operations, all of which correspond to the equivalent paging abstraction operations. The second set of operations manipulates 'inactive' address spaces and the third set manipulates 'active' address spaces. The fourth set of operations are used to change the parameters associated with active address spaces. The fifth set of operations manipulates 'inactive' blocks, the sixth set manipulates 'active' blocks, and the seventh set manipulates an active block and an inactive block in the same operation. Finally, the last set of operations is used in making decisions regarding activation and deactivation of blocks and address spaces. We will now discuss each of the eight sets of operations.

The first set of operations implements the standard access functions of memory. Addresses are in the form of a two tuple, 'address space name, block name'. All these operations return as a fault, if the address space or block is not active, or if the block does not exist for the address space. If a block is inactive, the fault return indicates whether a home is associated with the block. An important point with respect to the address space abstraction is that since it does not implement the multiplexing policy, it does not hide the state in which the block object is maintained. An active block is maintained as a page object, and an inactive block as a home object. The BZERO operation requires the block to be active and directs the paging abstraction to zero all the elements of the associated page.
Set_1

READ_VM4_ASB (as_n, b_n", offs, elem): fault(tf', df, hf', bf')

WRITE_VM4_ASB (as_n, b_n", offs, elem): fault(tf', df, hf', bf')

BZERO_VM4_ASB (as_n, b_n")': fault(tf', df, hf', bf')

Set_2

CREATE_VM4_AS (len_max, len_curr, plim_max, as_n): fault(f4)

DESTROY_VM4_AS (as_n")': fault(ef', tf)

SETMAX_VM4_LM (as_n", len_max): fault(ef', f4, pbf)

SETPMAX_VM4_PLM (as_n", plim_max): fault(ef', f1)

Set_3

ACTIVATE_VM4_AS (as_n", len_curr', plim_curr):

    fault(ef', tf, f1, f4, f3, abf)

DEACTIVATE_VM4_AS (as_n")': fault(ef', sf)

DEACTALL_VM4_AS (as_n")': fault(tf')

Set_4

SETLEN_VM4_AS (as_n", len_curr): fault(ef', f4, f3, abf)

SETPPLIM_VM4_AS (as_n", plim_curr): fault(ef', tf', f1, f2)

Set_5

ASSIGND_VM4_ASB (as_n, b_n", h_n): fault(tf', bf, df, hef')

ASSIGNW_VM4_ASB (as_n, b_n", h_n): fault(tf', bf, df)

RELEASE_VM4_ASB (as_n,b_n")': fault(tf', bf, df)

DESTROY_VM4_ASB (as_n,b_n")': fault(tf')

RECREATE_VM4_ASB(as_n,b_n")': fault(tf', df)
Set 6

ACTIVATE_VM4_BK (as_n,b_n"'): fault(tf',df,hf',if,mof)

DEACTIVATE_VM4_BK (as_n,b_n''): fault(tf',df)

Set 7

SWITCHACT_VM4_LOC (as_n,b_n_s,b_n_t): fault(tf',dfs,df_b,bf',
                           hf',mof)

Set 8

QREF_VM4_ARF (as_n",tf,asrf,len_curr')

QREF_VM4_BRF (as_n,b_n"',bf,brf): fault(tf')

SUCC_VM4_AAS (asme_n,as_n,asrf,asme_n)

SUCC_VM4_ABK (as_n",b_n,b_n,brf): fault(tf')

GETR_VM4_LAS (asme_n)

SETR_VM4_LAS (asme_n)

GETR_VM4_LB (as_n",b_n): fault(tf')

SETR_VM4_LB (as_n",b_n): fault(tf')

Fig. 3.4 The VM4 Operations
abf : cannot increase the length of an active address space

bf  : block active (or inactive if bf' is specified)

df  : block destroyed

ef' : address space does not exist

fl  : plim_curr > plim_max

f2. : plim_curr < # (allocated page indexes)

f3  : len_curr > len_max

f4  : len_curr < length upto last block with a home associated with it

hef' : given home does not exist

hf' : block does not have a home associated with it

if  : page index cannot be allocated

mof : page cannot be allocated (need to deactivate some addr. space)

pbf : cannot increase the 'len_max' of the address space

sf  : some blocks are still active in the address space

tf' : address space is inactive

Fig. 3.5 Interpretation of the Fault flags
Since address space objects are of unlimited number (the create/destroy kind), they have a pair of operations, CREATE and DESTROY, which define their lifetime. An address space object has three parameters always associated with it. An additional parameter is associated when the address space is active. The len_max parameter decides the maximum length of the address space in terms of the number of blocks it contains. The len_curr parameter identifies the current length and is required to be less than the maximum length. All blocks which exist in the address space lie within the current length. The plim_max parameter sets the maximum page limit. When an address space is activated, a current page limit is set for it by fixing the plim_curr parameter. The page limit is used by the page limiting feature, supported by the abstraction. The page limit serves as a bound for the number of pages which can be allocated to the blocks, which are active in the address space. The plim_curr value has to be less than the plim_max value. The SETLMAX and SETPMAX operations can be used to change the len_max and plim_max parameters of an address space. The significance of the maximum length is that an inactive address space provides for space to maintain information for that many blocks, although all the blocks may not exist. Therefore, the SETLMAX operation may return with a 'pbf' fault, if it causes and increase in len_max, for which the inactive address space cannot maintain the additional information. An active address space maintains information only for blocks within the current length.

The third set of operations provide for activation and deactivation of address spaces. The len_curr parameter may be changed at the time of activation, a null input value indicating that a change is not required. A third parameter is required for the ACTIVATE operation, and it sets
the page limit for the active address space. Active address spaces are a limited resource. The limit is not enforced on the number of active address spaces, but indirectly through the sum of the current lengths of all active address spaces. The ACTIVATE operation returns with a 'abf' fault, if an address space of that current length cannot be activated. The ACTIVATE operation also returns as a fault ('ef' fault) if the address space name is invalid. An address space name is invalid if no address space of that name currently exists, although one may have existed and then been destroyed. This check for address space name validity is made by all the operations defined for manipulating an inactive address space. The operations for active address spaces, in any case, return as a fault if the address space is not active. The DEACTIVATE operation deactivates an address space and requires that all blocks of the address space be inactive. The DEACTALL operation deactivates all the active blocks in an address space. This operation can be used in conjunction with DEACTIVATE to deactivate an address space which has some active blocks.

The fourth set contains a couple of operations to change the len_curr and plim_curr parameters of an address space. Both these parameters are dynamically used by other operations for checking block validity and to activate and deactivate blocks. Hence, care has to be taken in modifying these parameters, if the address space is active. If len_curr is increased for an active address space, additional information has to be maintained and this may cause an 'abf' fault. The plim_curr parameter cannot be changed to a value lower than the total number of pages used by the active blocks in the address space. This condition causes an 'f2' fault return.
Since address space objects are unlimited in number and can contain up to a maximum number (decided by the system bound on len_max) of blocks, block objects are also unlimited and should have a pair of CREATE, DESTROY operations defining their lifetime. However, creation of block objects is implicit through creating a containing address space and setting the current length (len_curr) parameter. Destruction is also implicit by reducing the current length. Strictly speaking, once a logical block object is destroyed, it should never exist again. But in terms of practical implementations, as in using an address space object to implement a variable length segment, this proves to be impractical. We therefore allow blocks to be recreated implicitly by increasing the current length, although the length had been decreased before. Implicit creation and destruction of blocks by changing the len_curr parameter is restrictive in that the order in which the blocks can be created and destroyed is fixed. The address space abstraction provides two operations, DESTROY_BLK and RECREATE_BLK, with which any block in an address space can be destroyed or recreated. The DESTROY_BLK operation can be used to enforce the policy that a block object which is once destroyed, cannot be recreated implicitly. This is because a block, once destroyed by the DESTROY_BLK operation, can only be recreated explicitly (by the RECREATE_BLK operation). A block can neither be implicitly nor explicitly destroyed, if it has a home associated with it. Violation of this condition causes a 'f3' fault return in any operation which affects the len_curr parameter. The check for block validity in an address space checks the len_curr parameter first and then checks if the block has been destroyed explicitly. Any existing block still has to be within the current length. An attempt to recreate a block out-
side the current length, causes the RECREATE_BLK operation to return with a 'df' fault.

The block to home association can be changed by the ASSIGND and ASSIGNW operations. The ASSIGND operation associates an existing home with the block and requires a home name as an input parameter. The ASSIGNW operation allocates a new home and associates it with the block. The operation returns the name of the new home. These two operations may be used to effect sharing between blocks, by associating them with a common home. Both the operations require the block to be inactive, otherwise they return as a 'bf' fault. The ASSIGND operation may also return with a 'hef' fault if the home which is to be shared does not exist. The RELEASE operation disassociates the block and its home. The home is not necessarily freed, as some sharing blocks may still be associated with the home.

The operations of the sixth set provide for activation and deactivation of blocks. Since activating a block corresponds to associating a page with the block, there has also to be a home associated with the block. Otherwise the ACTIVATE operation returns with an 'hf' fault. Activating a block requires allocating a page index in the address space, as well as a page. If there is an active sharing block in the same address space, then neither of these allocations are required. If there is no active sharing block in the same address space, but there is at least one in some other address space, then only a page index needs to be allocated. Otherwise, both a page index and a page have to be allocated. The ACTIVATE operation returns with an 'if' fault if a page index cannot be allocated and an 'mof' fault if a page cannot be allocated. The DEACTIVATE_BLK operation deactivates a block, but the freeing
of the associated page index and page depends on whether active sharing blocks exist. We thus see that the page limiting feature is enforced through the allocation and freeing of page indexes within an address space.

The SWITCHACT operation in the seventh set provides for the transfer of block activation between two blocks in the same address space. The idea is to ensure that the target block does get activated without an 'if' or 'mof' fault. However, if the source block has an active sharing block in some other address space, then the page is not freed and, hence, a new page has to be allocated for the target block. This may cause an 'mof' fault if a page cannot be allocated. In any case, an 'if' fault cannot occur. The target block still has to have a home associated with it. It is important to realize that the SWITCHACT operation does not serve to pass the contents of a page from one block to another. The logical contents of the page changes since the page-to-home association is changed. The SWITCHACT operation is mainly to be used to implement block activation policies within an address space.

The final set of operations are used to decide the block or address space which is to be deactivated. Every active block in an address space has an associated block reference flag. Every active address space has an associated address space reference flag. The two QREF operations return the value of these flags. These operations also cause the corresponding reference flags to be reset to 0, if it was initially set to 1. The criteria for setting the reference flags to 1 is such that the state of the reference flag is an indication of the frequency of reference to the block or address space. One use of these reference flags is to implement the 'modified LRU' policy, which was used in the paging abstraction.
The NEXTA operation requires an 'asme_name' as an input parameter. The asme_name indicates an internal index associated with an active address space. The SUCC_AAS operation returns the identity of an active address space, its reference flag, and a new asme_name. The asme_name serves as a reference point in implementing the modified LRU policy for address space deactivation. In between deactivations, the asme_name is maintained by the address space abstraction itself. A couple of access operations, GETR_LAS and SETR_LAS, are defined for this purpose. Each active address space maintains the identity of the last block deactivated in it. The operations, GETR_LB and SETR_LB, are defined to access this identity. The SUCC-ABK operation requires a block name as an input parameter and returns the next active block in the address space, as well as this block's reference flag. The above three operations can be used to implement a modified LRU policy with respect to block deactivation within the address space.

Reference flags were not visible to the users of the paging abstraction because the multiplexing policy was handled internal to the abstraction. The address space abstraction only provides the mechanisms to enforce the multiplexing policy. Hence it provides operations to access reference flags. Since these operations do not affect any other states of the address space abstraction, their only use is in deciding which block or address space to deactivate. Actual deactivation and activation still has to be through invoking the operations in the other sets.

Finally, there can also be a set of interrogative operations defined for the abstraction. These operations do not change the state of an address space or block object, but only return with information
about the states. Since these operations are simple to understand and implement, we will not discuss them.

d) The structure of the Extension Mechanism

The structure of the extension mechanism for the address space abstraction (VM4) is illustrated in Fig. 3.6. The address space abstraction extends from the VM2 paging abstraction (implicitly through the VM3 abstraction). It uses two external mainstream abstractions, VP2, and HNR. The VP2 (base processor) abstraction is the same as that used by the paging abstraction. The HNR (home name manager) abstraction is part of the file system mainstream (just as the secondary storage, SS, abstraction was). We regard the home name manager as a map abstraction defined for the 'home name_type'. The extension mechanism of the address space abstraction consists of three map abstractions and three data abstractions which manage map objects. For the sake of clarity, we have not separately indicated any of the functional abstractions composing the extension mechanism. All the functional abstractions have been defined from the point of view of code collection, and do not require special interaction mechanisms, like the frame freer of the paging abstraction did. Hence, there is no need to include them separately as they are not crucial to the structure.

The three map abstractions—PSM (passive* space map), ASM (active space map), and PNM (page name map)—are defined for 'address space', 'active address space' and 'page' name_types, respectively. The three other abstractions composing the extension mechanism, PBR, ABR and PIR

* A 'passive' address space is an alternative term to refer to an 'inactive' address space.
Fig. 3.6 The Structure of the VM4 Abstraction
manage PEM (passive block map), ARM (active block map), and PIM (page index map) objects, respectively. Every address space has a PEM object associated with it. A PEM object is a map defined for the 'block name_type', and contains the information about the blocks composing an address space. Every active address space has an ARM object and a PIM object allocated to it. An 'active address name' (distinct from the address space name) identifies both the ARM and PIM objects. An ARM object is also a map defined for the 'block name_type' and contains the information about the blocks composing an active address space. A PIM object is a map defined for the 'page index_type*', and contains the information about the page indexes available for an active address space. A PIM object is used to enforce the page limiting on an address space.

The PSM and PBR abstractions are collectively called the passive space manager (PSM). The ARM, ABR and PIR abstractions are collectively called the active space manager (ASM).

A PEM object maintains the information for the blocks composing an address space, when the address space is inactive. In this state, all blocks have to be inactive. The domain of the block name_type for this map is decided by the len_max parameter. All entries for block names falling outside the len_curr value, correspond to destroyed (or created) blocks. Some of the entries for block names within the len_curr value may correspond to explicitly destroyed blocks. Once an address space is activated, its blocks may be activated and deactivated. An ARM object maintains this dynamic information about the blocks composing the address space. The domain of the block name_type is fixed by the len_curr param-

* An 'index_type' defines a set of integers. Unlike a name_type, it does not identify the objects of some type.
eter, in this case. Initially, when the address space is activated, the entries in the ABM are initialized from the PFM object. When the address space is deactivated, all the blocks have to be inactive. The entries in the PFM object are updated from the ABM object. The PIM object maintains information about the page indexes, which are to be associated with each active block in the address space.

The PSM (passive space map) maintains the set of parameter values for each address space, and the name of the PFM object used by an address space. The ASM (active space map) maintains the dynamically changeable parameter values for each active address space, and the name of the ABM and PIM objects allocated to an active address space.

e) The Home Name Manager (HNR)

The home name manager is one of the external mainstream abstractions used by the address space abstraction. The abstraction forms part of the file system mainstream. In this research, we have taken a very simplistic view of the file system mainstream and consider the HNR abstraction to be a simple map defined for the 'home name_type'. We define the HNR abstraction in terms of a limited functionality set, and do not offer any suggestions with respect to its internal organization.

The HNR abstraction provides a pair of ALLOCATE, FREE operations, which define the lifetime of a 'home name'. The address space abstraction views this as defining the lifetime of a home object. The HNR abstraction maintains a 'share count' for each home name. This count indicates the number of blocks sharing the particular home. For each additional block, which shares a home, the share count for the home name is incremented by the SHARE operation. The SHARE operation returns an 'hef' flag (home existence flag) which is set to zero if the home does not
exist. The FREE operation sets the home name free, only if the share count is \(1\). Otherwise, its only effect is to decrement the share count by \(1\).

The HNR abstraction maintains a zero flag with each home name. This can be accessed by a pair of GET and SET operations. The HNR abstraction does not interpret the zero flag and the file system does not guarantee that the zero flag is consistent with the contents of the home. This is because we have taken a very simplistic view of the two abstractions, SS (secondary storage) and HNR, which compose the file system mainstream. Therefore, the virtual memory abstraction regards the SS abstraction as a manager of an unstructured set of home objects, and the HNR abstraction as a map defined for the home name type.

We see now, one of the advantages of structuring through type extension based abstractions, namely we can take a simple view of abstractions which may be designed more elaborately at a later point in time. For instance, the file system mainstream can be designed to include the concepts of physical (disk) volumes, logical volumes, removable volumes, etc. Furthermore, it can impose restrictions such that the blocks composing an address space are to be maintained in one volume and also provide for storage of the PEM object in the same volume. All these features can be realized by adding more structure to the 'home name', its map abstraction (the HNR) and the manager of home objects (SS). As far as the virtual memory abstraction is concerned, its internal structure remains unaffected by most of these concerns. Designing an elaborate file system is beyond the scope of this research.

f) **The synchronization considerations**

Similar to the paging abstraction, the main synchronization consid-
eration of the address space abstraction comes from the fact that there can be concurrent users (processes) of the abstraction and no assumptions can be made with respect to the relative efficiencies of execution. The two other synchronization considerations of the paging abstraction—interacting with the frame freer and invoking a secondary storage (SS) operation—are not applicable in this case.

The synchronization requirement arising out of the concurrency consideration, is satisfied by a system of locks. The abstraction uses three locks, a 'ps_lock' maintained in a PSM (passive space map) entry for an address space, an 'as_lock' maintained in an ASM (active space map) entry for an active address space, and a 'pg_lock' maintained in a PNM (page name map) entry for a page. The locks are always set in the order, ps_lock, as_lock, pg_lock, and unlocked in the reverse order, thus preventing any deadlock conditions.

The key to the use of the ps_lock and as_lock are 'test and set' operations, which first make a check on parameter names passed to them, and only then test and set a lock. The TSTE operation requires that it be passed an address space name and first checks the PSM entries to ensure that the address space exists, and only then test and sets the ps_lock. In addition to the original state of the ps_lock, the operation returns an 'ef' (existence flag) flag which is set to zero if the address space does not exist.

The TSTG and TSTV operations are defined to test and set the as_lock. The TSTG operation requires that an address space name be passed to it. The operation searches the ASM entries for an 'active address space name' (aas name) which has been allocated to the address space. If successful, the corresponding as_lock is tested and set, and
the operation returns the associated 'aas name'. If no aas name has been allocated, it implies that the address space is not active and, hence, a null aas name is returned. The usefulness of this operation lies in the fact that if the as_lock has been successfully set and a non-null aas name is returned, then the aas name cannot be deallocated from the address space, that is, the address space cannot be deactivated. The TSTV operation provides a variant of this function. It requires that an 'aas name' be passed to it, in addition to the address space name. The operation checks if the aas name has been allocated to the address space, and, if so, the as_lock is tested and set. If the check fails, the operation carries out the same function as the TSTG operation. The advantage of the TSTV operation is that a search for the aas name can be avoided if the two input names match. This operation is used after an initial TSTG operation invocation, in busy waiting without the search overhead.

The functional abstraction, $LOCK_PSL, uses the TSTE operation in conjunction with a 'while' control construct, to busy wait on the ps_lock. Similarly, the $LOCK_ASL functional abstraction uses the TSTG operation to busy wait on the as_lock. The abstract implementation for the address space abstraction uses this simple locking strategy to guarantee its correctness of operation with respect to concurrent users. The pg_lock associated with a page, is locked by a simple test and set operation—TST_PGL. Since a page can be shared across address spaces, the first two locks are not sufficient to prevent access to a page. With respect to locking a page, it is sometimes necessary to ensure that the page-to-home association holds after the pg_lock is locked. However, since the page-to-home association is maintained by the paging abstraction, the PNM cannot
provide an atomic operation (like TSTG) to carry out this function. Hence, a functional abstraction, $LOCKVER_HOME$, is defined for this purpose.

Before discussing the blocking considerations in the design of the address space abstraction, we will consider a synchronization problem. When any of the memory access operations (set_1) of the address space abstraction (VM4) are invoked, the associative map (AM) maintained by the ABR abstraction is searched to find the name of the page allocated to an 'address space, block' pair. If the search is successful, the corresponding VM2 operation is invoked on the page. The problem arises because of the possibility that the page is freed from the block (that is, the block is deactivated*) and allocated to some other block, by the time the VM2 operation is executed**. This causes the VM4 operation to access the contents of some other block and not the block that it was invoked on. This violates the correctness of the VM4 operation. The key to the problem is that after a successful associative search, the block should not be deactivated until the corresponding VM2 operation is completed.

The solution to the problem lies in the use of a 'ucount' (use count) maintained in each associative map (AM) entry. A successful AM search by the AGET operation causes this ucount to be incremented. All the operations which require a block to be deactivated use the $DEACT_BLOCK functional abstraction to deactivate the block. This function invokes the ANUL operation to remove any AM entry for the block.

*Since a successful AM search does not lock any of the locks, there is nothing preventing the block from being deactivated.

**Note that the paging abstraction does not guarantee that simultaneous operation invocations by different users are executed in the strict order of invocation.
The ANUL operation returns an 'amf' flag set to 1, if the block deactivation has to be held up. Therefore $DEACT\_BLOCK busy waits on the ANUL operation until a flag value of zero is returned.

The criteria for the ANUL operation delaying a block deactivation is as follows. If the identified block (the block is identified with an 'aas name, block name' address) has an AM entry, then the entry is removed. However, if the 'ucount' for the entry is non-zero, then the association between the 'aas name, block name' pair and the ucount, is maintained in another associative map called ANM (associative null map). The ANUL operation returns with a flag value of 1. Even if no AM entry exists for the block, it is possible that an ANM entry exists for it. This is because an AM entry corresponding to the block may have been removed earlier because of an ASET operation adding a new AM entry. The replacement policy for AM entries is such that it prefers entries with a zero ucount, but if such an entry does not exist, then the entry with the lowest ucount is chosen. In this case, the block's association with its ucount, is transferred to an ANM entry. Here, also, the ANUL operation returns with a flag value of 1. If the ucount in a block's AM entry is zero, or if no AM and no ANM entry exists for the block, then the ANUL operation returns successfully with a zero flag value.

The implementation of a memory access (set_l) VM4 operation is such that if the associative search is not completely successful, the operation locks the as_lock, gets the name of the page allocated to the block, and then adds the new association into the associative map by invoking the ASET operation. The ASET operation allocates an AM entry (it might have to replace an existing AM entry), initializes the AM entry with the new association, and increments the 'ucount'. The VM4 operation then un-
locks the as_lock and only then invokes the VM2 operation. Thus, in all cases, the set_l VM4 operation ensures before invoking a VM2 memory access operation that the ucount, for the AM entry associated with the block that it is accessing, is non-zero. This causes the delay of any operation which requires the block to be deactivated. After the VM2 operation is executed, the VM4 operation decrements the ucount through the DECR operation. This operation decrements the ucount in the AM entry, if an AM entry still exists for the identified block. Otherwise, the block to ucount association is maintained in the ANM, in which case the ucount is decremented and, if it becomes zero, the ANM entry is removed. This causes any subsequent ANUL operation on the same block to be successful.

There are three forms of blocking which can occur within the execution of a VM4 operation. The first is the indirect blocking caused by invoking a VM2 operation. This is because any of the VM2 operations can be blocked in the paging abstraction. The second form of blocking occurs because of using the $DEACT_BLOCK function to deactivate a block. As we have just seen, this function may cause a busy wait on the ANUL operation. The final form of blocking occurs because of the busy wait on the three types of locks used by the address space abstraction.

Table 3.1 tabulates the blocking which can occur for each of the VM4 operations. The CREATE_AS and ACTIVATE_AS operations do not busy wait on their respective ps_lock and as_lock, but automatically set the locks when a PSM or ASM entry is allocated. Similarly, if the ACTIVATE_BK and SWITCHACT operations allocate a page, then the pg_lock is automatically set in the PNM entry, otherwise the operations busy wait on a pg_lock. The three set_l operations, READ, WRITE and BZERO, busy wait on the
### Table 3.1 Internal Blocking for the VM4 Operations

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>WAIT on ps_lock</th>
<th>WAIT on as_lock</th>
<th>WAIT on pg_lock</th>
<th>WAIT on ANUL</th>
<th>Through VM2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_VM4ASB</td>
<td></td>
<td>partial</td>
<td></td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>WRITE_VM4_AS</td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>BZERO_VM4_AS</td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>CREATE_VM4_AS</td>
<td>auto.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESTROY_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETLMAX_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETPMAX_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACTIVATE_VM4_AS</td>
<td>x</td>
<td>auto.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEACTIVATE_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEACTALL_VM4_AS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>SETLEN_VM4_AS</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETPLIM_VM4_AS</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSIGND_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSIGNW_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RELEASE_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESTROY_VM4_AS</td>
<td>x</td>
<td></td>
<td>(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECREATE_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACTIVATE_VM4_AS</td>
<td>x</td>
<td>auto or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEACTIVATE_VM4_AS</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWITCHACT_VM4_LOC</td>
<td></td>
<td></td>
<td>auto or</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
as_lock only if the associative search is not completely successful.
Even then, these operations unlock the as_lock before invoking a VM2
operation. Since all the blocking, except that caused by a VM2 opera-
tion, require busy waiting, it is necessary that there be no frequent
blocking in the VM4 operations. In most cases, excessive blocking is
caus ed only if an indirect blocking by a VM2 operation occurs, when some
of the locks have been set. A qualitative analysis* (Appendix 2b.5)
shows that, in most cases, the setting of locks is such that it does not
cause a significant blocking overhead. Moreover, the crucial memory
access operations (of set_l) of the address space abstraction are de-
signed such that they avoid most blocking, and, even if they do cause
blocking, it is rarely with respect to other set_l operation invocations.

g) Overview

The VM4 abstraction can be characterized as

1) an unlimited number of address spaces,

2) each containing a large but limited number of block objects
   (implies a large but limited addressability),

3) the total number of memory container (block) objects is
   unlimited, and

4) each block object is of a small and fixed size.

The primary purpose of the address space abstraction is to support an un-
limited number of address space objects (which are of the create/destroy
kind). The limit on the total number of memory container (block objects
is automatically removed, since each address space object contains a set
of blocks.

* A more thorough analysis could be based on a simulation study.
The main feature of the address space abstraction is that it provides logical memory spaces, which can be created and destroyed. Another feature of the abstraction is that it provides the mechanisms for sharing between logical spaces. The abstraction also supports the enforcement of local multiplexing policies with respect to block objects. The abstraction is designed so as to support the efficient accesses to blocks in the same address space, which are made in the same time interval. An important feature of the abstraction is that it accommodates the execution of simultaneous invocations of its operations, by concurrent users of the abstraction. The inbuilt synchronization guarantees the correctness of the operations with respect to this concurrency.
Section III.6  The Final Abstractions of the Virtual Memory Mainstream

a) The role of the final Virtual Memory abstraction

The memory container objects defined by the real memory, paging and address space objects—frames, pages, and blocks, respectively—are all small and of the same size. To obtain the final virtual memory abstraction, we need to remove this restriction on the size of the objects. The address space objects supported by the address space abstractions can contain a large but limited number of block objects, since there is a system wide bound on the maximum value that the 'len_max' parameter of an address space can take. This is another restriction which needs to be removed to obtain the virtual memory abstraction.

Since an address space object, managed by the VM4 abstraction, can contain a large number of blocks, it seems reasonable that the unlimited size virtual memory container objects can be implemented using these address space objects*. Arbitrarily large size virtual memory containers are called 'segments'. The extension of the segmentation abstraction (VM5) from the address space abstraction seems straightforward if each segment object is to be implemented in terms of an address space (VM4) object. However, the complications arise because of two considerations—the variable size nature of segment objects and the efficiency requirements, both in terms of memory resources and execution time.

If segments are always of sizes which are orders of magnitude larger than that of block objects, then these considerations do not affect the implementation. However, segments by their very nature, vary in size from

* The limit on the maximum number of blocks which can be contained in an address space is large enough that, for all practical purposes, a single address is sufficient to implement an unlimited size virtual memory container.
a fraction of a block size to large multiples of block size. Therefore, using separate address space objects to implement small size segments imposes an overhead in terms of resource (memory) requirements. This is amplified by the fact that, in most characteristic systems, a large number of segments are accessed during the course of the various computations. This number is much larger than the limit on the number of active address spaces (imposed by the VM4 abstraction), thereby causing an overhead in execution time with respect to activating and deactivating address spaces. The problem that we have emphasized is typical of segmented memory systems and is called the 'small segment' problem.

The obvious solution is to group small segments together in an address space. Large segments can be implemented using an address space object for each segment. We, thus, have a size dependent implementation for segment objects. Since accessing a block requires both the address space and the block to be active, small segments which are activated in the same period of time should be grouped together in an address space. This ensures that address space activation and deactivation is kept to a minimum.

There is another concern in the design of the segmentation abstractions, namely, the policies to be used for address space and block activation and deactivation. Recollect that the VM4 address space abstraction only provides the mechanisms to enforce these policies. The problems of policy decision are left to the users of the VM4 abstraction. An approach to this problem is to use policies for block activation and deactivation, which are local to an address space and use a global system wide policy for address space activation and deactivation. The page limiting feature of the VM4 abstraction helps this approach.
If a large segment is implemented in terms of an address space object, then the policy and the page limit for the address space can be decided on the basis of the expected access characteristics to the segment. If a set of small segments are grouped together in an address space, once again the policy and page limit for the address space can be decided on the basis of the access characteristics to the set. Since the segments grouped together are to be accessed in the same period of time to minimize address space activation, and since the segments should not be referenced together in a smaller period of time to avoid block activation and deactivation, there is a tradeoff involved. We take the point of view that the user of the segment abstraction is in a position to make this judgement, based on knowledge of the reference behaviour, or based on past heuristics.

Some bookkeeping has to be maintained when small segments are grouped into an address space. This is with respect to the allocation of the logical address space provided by each address space object. Each small segment has to be allocated a portion of this logical address space*. Since segments can be destroyed in any order, information has to be maintained, which keeps track of the portions of the logical address space which have already been used by destroyed segments, portions which are currently being used by existing segments, and portions which have not yet been allocated to any segment. We have designed a 'group' object,

* Segmentation (in a paged system) was initially introduced in virtual memory systems, to solve the problem of logical space allocation [Wat 70]. Therefore, it seems as though this purpose is defeated by reintroducing a manifestation of the allocation problem. This is not so, since now the allocation problem is restricted within an address space object, which can be created and destroyed. Furthermore, this allocation is necessary only in the case of small segments.
which maintains this bookkeeping information for a set of small segments which are grouped together.

Continuing with the evolution of the virtual memory abstraction, the final restriction on the maximum number of blocks that can be contained in an address space, may be removed by using segment objects to implement the arbitrarily large addressability address space objects. Since segment objects are of arbitrarily large size, they may contain the variable amount of mapping information required to implement the new kind of address space objects. The new kind of address space objects are called 'variable address space' (VM6) objects. The abstraction supported by the removal of this last restriction constitutes the final virtual memory abstraction.

b) The external (architecture) interface of the Virtual Memory subsystem

The virtual memory mainstream abstractions up to VM4 are characterized by the fact that these abstractions are not directly visible to the external users of the total system (architecture). The abstractions supporting segment and variable address space objects are visible at this external interface. The segmentation abstraction also makes group objects partially visible at the external interface, since their use is linked to the use of small segment objects. We define 'visibility' as the availability of a set of operations to manipulate the objects supported by an abstraction. Thus, the external users can invoke operations defined on segment and variable address space objects, and a subset of the operations defined on group objects.

External visibility of an abstraction brings up an additional consideration—protection. We had stated that there are two forms of protection in a system structured by type extension. The first form is
vertical protection between the type manager corresponding to an abstraction and the users of the (objects supported by the) abstraction. The second form is horizontal protection between the users of the abstraction themselves. For the non-visible abstractions, objects supported by the abstraction are identified by their name. Furthermore, the users of the abstraction do not use the names identifying the component objects of the abstraction. This implicit protection can be guaranteed by verification. At the external interface, objects cannot be identified directly by names for two reasons:

1) A name may be used to access objects supported by non-visible abstractions (violation of vertical protection).

2) A name may be used to access visible objects to which the user is not granted accessibility (violation of horizontal protection).

The solution to the problem is to seal each of the names identifying a visible object, in a tamperproof container. The external interface requires that such a container, containing a name to a visible object, be provided for any access to the visible object. Since containers containing names to non-visible objects do not exist, such objects cannot be accessed at the external interface. Furthermore, since the containers are tamperproof, the name that it contains cannot be changed. Therefore, each user can access only those objects whose names are contained in the containers, possessed by the user. These tamperproof containers are called 'capabilities'.

We have already discussed the structure of a capability as containing a type identifier, a set of rights and a unique ID. The type identifier identifies the type manager which manages the object. The unique
ID identifies the object and corresponds to the name which is sealed. The set of rights correspond to the set of operations which can be invoked on the object, by the possessor of the capability. At the external interface, any operation invocation requires a capability identifying the object on which the operation is invoked. The type identifier is used to identify the abstraction which supports that class of objects, and the unique ID serves as a name* for the object. The right associated with the invoked operation is required to be present in the capability.

The three types of objects—segment, variable address space, and group—are the visible objects supported by the virtual memory subsystem. As always, the abstraction provided by the virtual memory subsystem is defined in terms of the operations which can manipulate these objects. A significant point is that group objects are visible only to the extent of supporting the maintenance of segment objects. Hence, many of the operations defined on these objects are hidden from the external interface. We will relegate most of the discussion of the external visibility of virtual memory objects to Chapter V, which describes the total architecture.

c) The 'Group' abstraction

Group objects are managed by a group object manager (GOR). A group object is associated with a set of small segments which are to be grouped together and implemented with one address space object. We have recognized three kinds of small segments. The first kind corresponds to seg-

* In the semi-formal specifications, name-types for names identifying externally visible objects, will be distinguished by using capital letters for the name abbreviation.
ments whose sizes are a fraction of the block size. These are called 'very small' segments and usually many such segments use one block for their implementation. The second kind corresponds to segments whose sizes are less than the block size, but which require that the block, used for its implementation, not be used by any other segment. The need for such a requirement will be established later. These segments are called 'little' segments. The third kind of segment has a size which is greater than a block size, but which does not require a separate address space for its implementation. Such segments are called 'medium' segments. We will use the term 'small' segments to collectively refer to these three kinds of segments. Apart from small segments, 'large' segments are such that they are implemented with a separate address space for each segment. Every segment object has an 'implementation kind' (i_k) value which identifies the kind of segment it is.

The group object manager is specified in Appendix 2c.1. Every group object provides a set of 'group indexes'. The range of the 'group index_type' is a parameter of the group object and decides the maximum number of small segments which can be associated with the group object. Another parameter of a group object decides the maximum number of blocks which can be used by the small segments associated with the group. The maximum size of the address space (len_max), which is used for the implementation of the small segments, is decided by this same parameter.

* Note that the implementation kind is an implementation feature of a segment and is distinct from the segment 'type', to be introduced in Chapter V.
A group object consists of two maps, one defined on the 'group index_type' and the other on the 'block name_type'. Every small segment associated with the group, is allocated a unique group index. Depending on its implementation kind, it is also allocated a fraction of a block, a single block, or a set of blocks. These are used in the implementation of the small segment. The first map maintains a set of attributes for each group index. These attributes identify the implementation kind of the segment to which the group index is allocated, and the names of the blocks used in the segment's implementation. Group indexes, corresponding to 'very small' segments which use the same block, are linked together (in a SSL list). Group indexes are non-reusable in that they are destroyed if the segments associated with them are destroyed. Therefore, the map has also to keep track of destroyed group indexes, as well as unallocated ones.

The second map composing a group object is defined on the 'block name_type'. Although its range is decided by the total number of available blocks, entries corresponding to blocks which are used in implementing 'very small' segments are the only ones which have to be maintained (refer to Appendix 2c.1). The attributes for such entries contain the header to the corresponding SSL list and the fraction of the block which has been used.

Each group object maintains a set of map wide attributes. These correspond to the maximum number, the number allocated and the number of blocks still in use. The same three values are also maintained for group indexes.

Group objects are of the create/destroy kind and a pair of CREATE, DESTROY operations define their lifetime. Group indexes, although
bounded by a maximum number, are of the create/destroy kind. There are three forms of the CREATE operation defined for a group index. These correspond to the implementation kind of the segment which is to be associated with the group index. A single DESTROY operation is sufficient for all the three kinds of group indexes. A set of GET and SET operations are defined to access the attributes for the map defined on the group index_type. The map defined on the block name_type is used internal to the group object and operations are not defined to access these attributes.

When a 'very small' segment is destroyed, the block used for its implementation cannot be destroyed, if other 'very small' segments are still using the block. Therefore, the block contents used by the segment are still accessible. Although the segment abstraction prevents any further accesses to the segment through it, there is another form of direct access to the block contents, which may still be possible. (We will discuss this direct access to a block contents, in the next subsection). Therefore, the DESTROY operation in this case returns an 'sdf' flag (security deficiency flag) set to 1, to indicate this condition. This condition may also be checked by an interrogative operation, QCHECK, which returns an 'sdf' flag value of 1 if the group index corresponds to a 'very small' segment which is using a block also being used by other existing 'very small' segments. We now see the reason for the implementation as a 'little' segment, since the security deficiency condition does not arise, when the corresponding segment is destroyed.

All the small segments associated with a group are implemented using a single address space. The len_max parameter (maximum length) of the address space is used in creating the group object. The association,
between the name of a group object and the address space used in the small segment implementation, is maintained in a group space map (GSM) (specified in Appendix 2c.2). This map is defined on the 'group name_type' and maintains as attributes, the name of the associated address space, the 'plim_max' and 'len_max' parameters corresponding to this address space, the 'plim_curr' parameter which is to be used in activating the address space, a 'policy name' and a 'g_lock'. The policy name identifies a local policy which is to be used in the activation and deactivation of the blocks composing the address space. The GSM abstraction defines a set of operations to access the attributes maintained in this map.

It should be realized that a group object serves merely as a maintainer of the bookkeeping information required when a set of small segments are implemented with one address space. The operations defined by the GOR abstraction do not affect address spaces or blocks. Thus, when a group index is destroyed, the GOR abstraction does not cause the blocks to be destroyed, but just updates the map in the group object. Hence, it is the responsibility of the user of the abstraction to ensure that the group abstraction is updated when the segment is destroyed. The user of the group abstraction is the segmentation abstraction. Most of the operations defined on group objects are not available at the external interface, but hidden in the segment abstraction's implementation. The operations defined for group objects at the external interface mainly access the attributes maintained for the group object in the group space map. We will discuss these operations when the external interface of the virtual memory subsystem is discussed in Chapter V.

d) **Direct Addressability at the external (architecture) interface**

Since segments are the only form of memory containers available at
the external interface, all addresses with respect to the virtual memory abstraction are to be of the form 'segment name, segment offset'. This implies that an indirection through the segment abstraction is required for each memory reference. This can cause a significant overhead on each access. The solution to this problem is to use a form of direct access to the memory containers which compose a segment. In other words, this suggests that there be a direct access to the blocks or fraction of blocks which constitute a segment. Direct addresses of this kind are in the form of a five tuple—'address space name, (block name, offset), (block name, offset)'. The first 'block name, offset' pair identifies the 'base' (start address) of the directly accessible memory space, while the second pair identifies the 'limit' (end address) of the directly accessible memory space. Addresses of this form are called 'pointers'.

The implications of direct addressability will be discussed in Chapter V. Needless to say, the availability of pointers is strictly restricted by the addressing structure of the architecture. The significant aspect of pointers is that once a pointer is derived from a segment name, accessing the segment contents can be independent of the segmentation abstraction. This bypassing of the segmentation abstraction has its pros and cons. The main advantage is that the indirection overhead is avoided. The disadvantage is that a security deficiency condition coupled with direct addressability to a segment, can cause access to a segment (only a 'very small' segment), even after it is destroyed. However, with sufficient control over the existence of pointers, this problem can be alleviated.

The abstraction supporting 'variable address space' objects is closely linked to the addressing structure at the external architecture inter-
face. It will be shown in Chapter V that the addressing philosophy of
the architecture calls for only an implicit recognition of variable ad-
dress space objects.

Section III.7 Summary

In this chapter, we have successfully structured the virtual memory
subsystem, using the techniques of type extension. We chose an interpre-
tation of the structuring technique which allowed us to decompose the
subsystem to a fine level of resolution. The primitive components of the
structure are either map abstractions or managers of map objects. The
motivation for the structuring was to resolve the subsystem into primi-
tive components, so that the implementation decisions with respect to
these components can be made separate from the design considerations of
the subsystem. We will discuss the nature of these implementation deci-
sions in Chapter V.

In order to concentrate on the design considerations of the virtual
memory subsystem, we characterized it as a linear ordering of mainstream
abstractions. We then designed the extension mechanism between each
pair of successive mainstream abstractions. Each such extension mechan-
ism was decomposed into a set of primitive abstractions. The operations
defined by these primitive abstractions were used in defining the opera-
tions of the extended mainstream abstraction. Interactions with abstrac-
tions in other mainstreams were modeled on the basis of the operations
defined by these external mainstream abstractions. For two of the impor-
tant virtual memory mainstream abstractions, the paging and address space
abstractions, the details of the extension from the lower level main-
stream abstraction, were specified by means of an abstract implementation
for the operations of the extended abstraction.
We departed from the conventional view of a paging abstraction by placing a limit on the number of page objects. We provided a practical basis for this viewpoint. The address space abstraction automatically removed this limit on the number of memory container objects, by supporting the create/destroy kind of address space objects. These objects were used as logical memory containers in implementing the unlimited and variable size segment objects. A significant aspect of the design of the segmentation abstraction is that it effectively solves the 'small segment' problem, typical of most segmented systems. Finally, we indicated that a segment could be used to contain the mapping information to implement variable address space objects.

The semantics of the operations defined by an abstraction, provide the most effective description of an abstraction. Hence, all the abstractions that we designed were specified in terms of the operations supported by that abstraction. This viewpoint of specification is essential to understanding the design of the virtual memory subsystem. The fact that we were able to design, structure and specify a non-trivial virtual memory subsystem on this basis serves as a justification of this viewpoint and the effectiveness of the type extension structuring technique.

The concept of abstract implementation is a powerful tool which can be used to study the interaction of an abstraction with its component abstractions. It provides a basis for the verification of the design of a subsystem. Given the availability of formal specification techniques for an abstraction [LiZ 75, Gut 77], we contend that formal proof techniques can be used to prove correctness properties of the abstraction design.

The abstract implementation for a subsystem provides a valuable insight into the structure of a subsystem. It enables the designer to cope
with the complexity of synchronization problems caused by concurrent users. It also allows a qualitative analysis of the synchronization properties of a design. The most important benefit of an abstract implementation is that it enables a heuristic approach to the implementation of a subsystem. This aspect will be discussed in the context of the meta-implementation issues of the subsystem in Chapter V.
CHAPTER IV

The Virtual Processor Design

Section IV.1 Introduction

The virtual processor subsystem extends from a limited number of physical processing elements to an unlimited number of virtual processing elements. A processing element (called a processor) interprets a program, in that it sequences the operations invoked in the program and executes these operations. We distinguish between the term process and processor. A process is a sequence* of operation invocations [Han 73]. A processor executes the sequence of invocations. A process specifies a series of actions; a processor is the actor which carries out the series of actions. A process may use different processors to execute different subsets of operation invocations. A processor may execute operation invocations on behalf of different processes.

A virtual processor subsystem consists of levels of virtual processors. At each processor level, the defining abstraction multiplexes the smaller number of processors supported at the next lower level, between the larger number of virtual processors that it supports. This implements the virtual execution of processors at that level. A virtual processor virtually executes the actions of its process, by causing the lower level processor to execute the action. At different stages of virtual execution, the virtual processor may have different low level processors associated with it. The processors at each processor level, specify the actions which are to be executed at the next lower processor level. There-

* The term sequence implies that the process retains the effect of the operations which have already been executed.
fore, the processors at each virtual processor level are the processes for the next lower level of processors. At the lowest level, the physical processors physically execute the actions. At the architecture interface, the user defined processes execute on the unlimited number of virtual processors supported by the architecture.

The virtual processor and virtual memory subsystems use the objects and operations defined by each other. We had seen one aspect of this relationship, when the paging abstraction of the virtual memory, used some of the operations defined by the Base processor (VP2) abstraction. The virtual processor abstractions need to use the virtual memory abstractions for storing inactive processor states. Existing designs for providing unlimited memory space and unlimited number of processors have relied on the technique of 'sandwiching' [Par 76a], to resolve the mutual dependency between the virtual memory and virtual processor subsystems [Sax 76, Jon 76, Ree 76]. In this technique, the two subsystems are decomposed such that the dependency relationships are between the abstractions composing the subsystems. These dependency relationships are such that they impose a partial ordering on the component abstractions, thus preventing mutual dependency. We follow the same approach.

The Base processor (VP2) abstraction uses Real memory partitions (VM1) for storing inactive processor states. The next higher level processor abstraction, the System processor (VP3), uses the Address Space abstraction (VM4) to store inactive processor states. The two virtual memory abstractions—Paging (VM2) and Address Space (VM4)—use the synchronization operations of the Base processor abstraction. Higher level VM abstractions can use the synchronizing operations of the System processor abstraction. Finally, all abstractions depend on the appropriate
level of virtual processor abstraction for 'interpreting' their imple-
mentations.

In a conventional architecture, there is only one abstraction incor-
porated by the architecture. The architecture is the interpreter for
programs consisting of invocations of these operations. The virtual
processor and virtual memory subsystems are defined external to the archi-
tecture and use the architecture as the interpreter for their implementa-
tion programs. In our architecture design, we take the point of view
that these sybsystems are incorporated into the architecture. We also
relax the constraint of one universal interpreter, and identify individual
interpreters with the different abstractions forming the internal struc-
ture of the architecture. We will consider the implications of this
viewpoint in Chapter V. However, in the virtual processor design of
this chapter, we need to recognize that abstractions can be implemented
on dedicated internal processors, virtual or physical. The implication
of this in the virtual processor design is that synchronization has to
be provided for invoking operations on these abstractions. The inter-
action with abstractions implemented on dedicated processors, is support-
ed by message communication abstractions defined at each processor level.

In Chapter III, we had emphasized that the structuring through main-
stream abstractions allows us to concentrate on the 'component'* relation-
ships between abstractions. The three other forms of dependency arising
from the actual implementation of an abstraction were called the meta-
implementation issues. The abstract implementations are concerned only
with the component relationships. Any synchronization required in invok-

* We have included the 'map' form of dependency as part of the 'compon-
ent' relationship.
ing an operation is, therefore, a meta-implementation issue, since it is a part of the 'interpreter' relationship. Each level of message communication abstraction depends on the lower level message communication abstraction, not because of the component relationship but because of the interpreter relationship. Therefore, the message communication abstractions are not considered as constituting a separate mainstream.

In this chapter, we will emphasize the modeling of the interdependencies between the virtual memory and virtual processor subsystems. We will also identify the role of the virtual processor subsystem in the meta-implementation of abstractions. We will design the message communication abstractions and show their use in the inter-processor communications, internal to the architecture. Finally, we will define the operations at the highest virtual processor level, which are visible at the external architecture interface. This level supports mechanisms which can be used in implementing user visible synchronization mechanisms.

The next section presents an overview of the virtual processor mainstream. Sections 3 and 4 discuss the two lower levels of the virtual processor mainstream. Section 5 discusses the message communication abstraction defined at each processor level. Section 6 identifies the interpretive role of virtual processors in the meta-implementation of abstractions. Section 7 considers the special implementation requirement for the STOP processor operation. Section 8 specifies the final virtual processor level. Section 9 summarizes the important features of the virtual processor design.
Section IV.2  The Virtual Processor Mainstream

The virtual processor mainstream consists of three levels of abstractions—the Physical processor (VP1), the Base processor (VP2), and the System processor (VP3). The VP1 abstraction manages the physical processors in the system. Since our interpretation of 'architecture' encompasses the virtual processor subsystem, all these physical processors form part of the total architecture. The VP2 abstraction multiplexes these physical processors between a larger but still limited number of base processors. Finally, the VP3 abstraction supports an unlimited number of system processors, a subset of which executes on the base processors.

Each level of virtual processors constitutes the set of processes for the next lower level of processes. The number of lower level processors available to the higher level abstraction is smaller than the number of virtual processors managed at the higher level. Therefore, the managing virtual processor abstraction has to maintain the states of the virtual processors which are not executing on lower level processors. Physical processor states are maintained in the internal registers of the physical processor itself. The VP2 abstraction maintains inactive (non-executing) base processor states in frame partitions. The VP3 abstraction maintains inactive system processor states in the blocks comprising address spaces. Since address spaces are unlimited in number, the VP3 abstraction can maintain an unlimited number of inactive system processor states. The paging (VM2) and address space (VM4) abstractions use the synchronization facilities of VP2, while the higher level memory abstractions (VM5, VM6) can use the synchronization facilities of VP3. We thus observe that the 'component' relationships impose a partial ordering on the VP and VP abstractions.
As we have pointed out, the virtual processor subsystem plays a crucial role in the meta-implementation of the mainstream abstractions. Primitive abstractions are the most suitable for implementation on separate dedicated processors—physical, base, or system. This is because primitive abstractions are generally assumed to execute operation invocations one at a time, although there may be simultaneous invocations. A primitive abstraction implemented as a separate 'hardware box' is equivalent to an implementation on a dedicated physical processor. Abstractions may also be implemented on dedicated base and system processors.

The Frame Freer used by the Paging abstraction is an example of an abstraction implemented on a dedicated processor (a base processor in this case). This abstraction required special interaction mechanisms (supported by VP2). In most other cases, interactions with abstractions implemented on dedicated processors are based on a standard protocol. The important point here is that a form of blocking is implicit in the use of these interaction mechanisms. For example, in interacting with the Secondary Storage abstraction, the blocking required because of the delay in secondary storage transfer is implicit in the use of the interaction mechanism.
Section IV.3  The Physical Processor Abstraction (VP1)

The VP1 abstraction manages the set of physical processors composing the architecture. Some of these are dedicated processors with a restricted functionality oriented towards implementing a specific abstraction. Others are more general but still restricted to be used for executing dedicated higher level processors. A third set of processors are available for multiplexing by the VP2 abstraction. In the discussion of the virtual processor abstractions, we will combine the latter two sets. Hence, each level of processor abstraction manages a set of general purpose processors available to the next higher level processor abstraction, and a set of processors dedicated at that level. The design considerations for additional partitions can be included without much modification to the proposed design.

As in specifying any abstraction, we will describe the VP1 abstraction in terms of the operations that it defines. In describing the processor abstractions, an additional means of description is available in that the effect of an operation invocation on a processor's status can also be specified. The operations defined by VP1 are specified in Fig. 4.1. Physical processor (P_procr for short) objects are of the allocate/free kind and have an ALLOCATE FREE pair of operations defined on them. A QFREE operation returns the number of free P_procrs.

Once a P_procr is allocated, a P_procr state can be loaded onto the processor through the SET operation. A processor state specifies the starting state of a processor and a series of operation invocations which it has to execute. The status of a processor is a more macroscopic description of the current processor state. A P_procr has to be in the W_STP1 status, before its processor state can be loaded (through SET) or
ALLOCATE_VP1_PR (ppr_n+)
FREE_VP1_PR (ppr_n")
QFREE_VP1_PR ( #(free P procrs) )

RUN_VP1_PR (ppr_n")
STOP_VP1_PR (ppr_n")

GET_VP1_PS (ppr_n", ppr_state)
SET_VP1_PS (ppr_n", ppr_state)

SIGNAL_VP1_SEM (sse_m_n)
WAIT_VP1_SEM (sse_m_n)
BLOCK_VP1_SBL ()
UNBLOCK_VP1_SBL (ppr_n")

Fig. 4.1 The Operations of the VP1 Abstraction
unloaded (through GET). A P_procr starts executing its processor state, when a RUN operation is invoked on it, in the W_STPl status. An executing P_procr can be made to stop executing by invoking the STOP operation on the P_procr. An executing P_procr is in the E_RUN1 status.

The VP1 abstraction defines a set of four synchronization operations which are used by a message communication abstraction. A P_procr waits for an input message by invoking* the WAIT operation defined on a 'synchronization semaphore' (SSEM). It is then blocked in the W_SSM1(---) status. A P_procr sending a message, invokes the SIGNAL operation on this semaphore. The sender P_procr can, itself, wait for an acknowledgement from the receiver processor, by invoking the BLOCK operation. This operation causes the P_procr to be blocked, in the W_SBL1(---)** status, in a 'sender block list' (SBL). The receiver P_procr signals an acknowledgement through the UNBLOCK operation defined on the SBL. A unique SEM has to exist for each receiving P_procr, but there need be only one SBL which can be used by all sender P_procrs. We will discuss the nature of the synchronization operations when we consider the analogous operations defined by the VP2 abstraction. Fig. 4.2a lists the different kinds of status, which a P_procr can take. Fig. 4.2b illustrates the effect of the VP1 operations on a P_procr's status.

The VP1 abstraction is ideally suited for an implementation in a distributed manner because it manages a set of physical processors. There

* A processor 'invokes' an operation on behalf of the process whose actions it is carrying out. Therefore, it is the process which uses the corresponding operation.

** Both W_SSM1 and W_SBL1 identify a status class. Each status in the respective class is differentiated by the receiver P_procr with respect to which the synchronization occurs.
W_FRE1 : Status of a free P_procr.

W_STP1 : Status of a newly allocated P_procr → P_procr is not executing.

E_RUN1 : Normal execution status of a P_procr.

W_SSM1 : Wait status, while P_procr (receiver) waits for a message.

W_SBL1 : Wait status, while P_procr (sender) waits for an acknowledgement to a message.

Fig. 4.2a The Different Status of a P_procr

Fig. 4.2b Status Transitions in VP1
is no requirement for centralized information to be maintained by the processor abstraction. In addition, at the physical processor level, there is an efficiency constraint for communicating between processors. This negates the use of a centralized implementation through which all the synchronization is directed. The operations of the VP1 abstraction have been specified so as to show the analogy with the VP2 and VP3 operations. The VP1 operations correspond to the hardware* equivalent of the VP2 and VP3 operations.

Section IV.4 The Base Processor Abstraction (VP2)

The VP2 abstraction multiplexes the P_procrs available to it between a larger but still fixed number of B_procrs. A subset of these B_procrs are dedicated processors at that level. The rest of the B_procrs are general purpose and can be allocated and freed by a pair of ALLOCATE, FREE operations. A B_procr is initially in the W_STP2 status, when it is allocated. A B_procr state can be loaded and unloaded to and from the B_procr, in this status. The RUN operation causes the VP2 abstraction to schedule the B_procr for execution. Note that, as far as the users of the VP2 abstraction are concerned, the RUN operation causes the B_procr to start executing. This execution is only virtual as the VP2 abstraction internally multiplexes the P_procrs to accomplish the physical execution. The status when a B_procr is physically executing is called E_RUN2, the status when it is waiting for execution is called W_RUN2. Collectively, the union of the two is called RUN2. Therefore, a RUN operation causes a B_procr status to change from W_STP2 to RUN2, while the

---

* This is in the sense of the implementation being at the primitive level of physical (logic) circuits.
ALLOCATE_VP2_PR (bpr_n+)
FREE_VP2_PR (bpr_n")

RUN_VP2_PR (bpr_n")
STOP_VP2_PR (bpr_n")

GET_VP2_PS (bpr_n", bpr_state)
SET_VP2_PS (bpr_n", bpr_state)

QRUN_VP2_NP ( #(B_procrs in RUN2) )
QEXEC_VP2_NP ( #(B_procrs in E_RUN2) )
QEMAX_VP2_NP ( #(maximum B_procrs in E_RUN2) )

SIGNAL_VP2_SSEM (ssem_n)
WAIT_VP2_SSEM (ssem_n)
BLOCK_VP2_SBL ()
UNBLOCK_VP2_SBL (bpr_n")

BLOCK_VP2_IQ (iq_n, iq_param.)
UNBLOCK_VP2_IQ (iq_n, iq_param.)
SETS_VP2_IQ (iq_n, iq_param.)

ENTER_VP2_MSEM (mon_n)
EXIT_VP2_MSEM (mon_n)
QEMPTY_VP2_MSEM (mon_n, ef)
EXCH_VP2_MSEM (mon_n)

WAIT_VP2_CQ (cq_n)
SIGEXIT_VP2_CQ (cq_n, mon_n)

Fig. 4.3 The Operations of the VP2 Abstraction
STOP operation has the reverse effect. In actuality, the STOP operation
does not cause a direct transition from the RUN2 to W_STOP status. The
actual implementation of the STOP operation is discussed in Section
IV.7.

Fig. 4.3 specifies the operations defined by the VP2 abstraction.
The interrogative operations, QRUN, QEXEC, QEMAX, return the number of
B_procrs in the RUN2 and E_RUN2 status, and the maximum number of B_procrs
allowed in the E_RUN2 status, respectively. The last value indicates
the number of P_procr which have been allocated for the VP2 multiplex-
ing. The four synchronization operations—SIGNAL_SSEM, WAIT_SSEM,
BLOCK_SBL, UNBLOCK_SBL—are used for message synchronization. The remain-
ing VP2 operations are those invoked by the paging abstraction and have
already been discussed in the previous chapter. The different kinds of
B_procr status and the effect of the VP2 operations on them are illus-
trated in Figs. 4.4 and 4.5. Note that, when a B_procr executes a syn-
chronization operation which causes it to be blocked, its associated
P_procr is released and allocated to some other B_procr which has been
scheduled for execution. When the B_procr is unblocked from a wait
status, its status is changed to a RUN2 status. The B_procr starts physi-
cal execution only when a P_procr is allocated to it.

The message synchronization operations are defined for communication
between two B_procrs. Each receiver B_procr is blocked on a unique
'synchronization semaphore' (SSEM). This semaphore is defined as a
queue, capable of accommodating one process, and an integer semaphore
count. The value of the count is bounded by the total number of B_procrs
which exist. The semaphore count is initialized to a value of zero. A
WAIT operation blocks the invoking B_procr on the semaphore if the sema-
W_FRE2 : Status of a free B_procr.

W_STP2 : Stopped status of a B_procr. Also, status of a newly allocated B_procr.

W_RUN2 : Status of an unblocked B_procr, waiting for execution on a P_procr.

E_RUN2 : Status of an executing B_procr.

W_SSM2(...) : Wait status class, corresponding to a B_procr (receiver) waiting for a message.

W_SBL2(...) : Wait status class, corresponding to a B_procr (sender) waiting for an acknowledgement.

W_IQ : Wait status, while blocked in an indexed Q.

W_MSEM : Wait status, while blocked in a monitor queue, while waiting for exclusive access to a monitor.

W_CQ : Wait status, while blocked in a conditional Q.

Fig. 4.4 The Different Status of a B_procr
Fig. 4.5 Status Transitions in VP2
phore count is zero. The SIGNAL operation unblocks a B_procr if there is one blocked in the semaphore queue, otherwise it increments the semaphore count by 1. We observe that, as long as there are no pending input messages, the receiver B_procr is blocked because of the WAIT operation on its SSEM. The sender B_procr signals the message through the SIGNAL operation on the SSEM, which causes the receiver to be unblocked and receive the message.

After sending a message, a sender B_procr can block itself until an acknowledgement (reply) is received. The 'sender block list' supported by the VP2 abstraction is used for this purpose. The BLOCK_SBL operation causes the invoking B_procr to be blocked in the SBL. A receiver B_procr acknowledges a message by invoking the UNBLOCK_SBL operation and causing the corresponding sender B_procr to be unblocked. A significant advantage of the SBL mechanism is that a receiving B_procr is not constrained to acknowledge messages in the order that they are received. In fact, more than one message can remain unacknowledged by a receiver B_procr. The utility of this feature is that a receiver B_procr can carry out actions which can be executed concurrently, although the sender B_procrs are unaware of this concurrency. A common SBL can be used by all the sender B_procrs. However, each receiver B_procr has to have a unique SSEM (identified by an 'SSEM name'*) associated with it.

We now need to clarify, what are the receiver and sender B_procrs? A receiver B_procr is a dedicated B_procr, which is used for the implementation of some abstraction. The messages that it receives correspond to the invocations of operations defined by the abstraction. Although

* Since each receiver B_procr has a unique SSEM, the 'bpr name' can, itself, be used to identify the SSEM.
it is not necessary, most of the receiver B_procrs execute one operation invocation at a time. We have found this single operation execution to be characteristic of all primitive abstractions. However, mainstream abstractions can also be implemented on dedicated B_procrs, if they require this constraint to be enforced. The sender B_procrs communicate the messages on behalf of the process whose invocations it is executing. Thus, in executing the invocation of an operation, a B_procr may send a message to the dedicated processor on which the abstraction defining the operation is implemented.

We had justified the distributed implementation of the VP1 abstraction on the basis of its hardware equivalency. The VP2 abstraction is better implemented as a single entity because it manages virtual processors. It requires the constraint that there be no concurrent execution of the VP2 operations. B_procrs invoking VP2 synchronization operations may cause blocking or unblocking of other B_procrs, which in turn may cause the release or require the allocation of P_procrs. In view of this centralization requirement, the VP2 abstraction is best implemented on a dedicated processor. Since the abstraction itself manages B_procrs, it itself has to be implemented on a dedicated P_procr. Thus invocations of operations on the VP2 abstraction requires that the interaction mechanism use the synchronization operations defined by the VP1 abstraction.

The VP2 abstraction uses some primitive abstractions, which are similar to maps and maintain the information about the B_procr status and the scheduling information. Since the VP2 abstraction is implemented as a single entity, these abstractions can be implemented using memory space local to the dedicated P_procr. These local abstractions are discussed in Appendix 3a.
Synchronization operations are the only operations invokable on dedicated processors. These processors cannot be allocated or freed, neither can their execution be stopped, since they are permanently executing. The GET and SET operations to access the processor states are also not applicable. However, a lower level processor which is used to execute the higher level dedicated processor, has all the operations defined on it. The lower level processor is released when the dedicated processor is blocked. When the dedicated processor is unblocked, it is scheduled for execution and subsequently another lower level processor may be allocated to it.
Section IV.5 The Message Communication Abstractions

A message communication (MC) abstraction is defined for each level of processor abstraction. Each MC abstraction uses the synchronization operations defined by the corresponding processor abstraction. An MC abstraction uses memory containers to contain the messages. The memory abstraction that manages these memory containers is lower in the total partial ordering structure, to the processor abstractions providing the synchronization. The three MC abstractions and their corresponding processor abstractions are:

Physical Message Commun. (MC1) — VP1
Base Message Commun. (MC2) — VP2
System Message Commun. (MC3) — VP3

An MC abstraction defines four operations for processor communication through messages. The SEND_MSG operation writes a message onto a message container and signals the receiver processor. The RECEIVE_MSG operation causes the invoking processor to be blocked until a message is signalled. The message is then read from the message container. The receiver processor can reply to a message through the SEND_ACK operation which writes a reply message onto the same message container used to receive the message. The sender processor is then blocked to receive the acknowledgement. The sender processor, after sending a message, can wait for an acknowledgement through the RECEIVE_ACK operation, which returns the reply message read from the message container.

Each message container contains, in addition to the message, an 'acknowledgement flag' (af) and a 'sender name'. The af flag is set to 1 if an acknowledgement is expected by a sender processor. The sender name identifies the sender processor. For example, if the communication
is between two processors, the sender name is the 'bpr name' of the sender. A function is defined for each processor level, to return the identity of the invoking processor. Each receiver processor is associated with an unsynchronized queue of message containers. The 'SSEM name' can be used to identify this queue. A SEND_MSG operation causes a message container to be added to the tail of this queue*, a RECEIVE_MSG operation, after the unblocking, causes a message container to be removed from the head of the queue.

The message containers and the unsynchronized queues are managed by 'message managers'. A message manager exists for each level of MC abstraction. A message manager extends from the appropriate virtual memory abstraction. The implementation of the message container objects and unsynchronized queues is straightforward and summarized in Appendix 3b. The operations defined by the message manager corresponding to the Base Msg. Commun. (MC2) abstraction, are specified in Fig. 4.6. The operations defined by the other levels of message managers are analogous. The abstract implementation code corresponding to the MC2 operations are shown in Fig. 4.7.

We have intentionally stressed the analogy between the three levels of message communication abstractions. A primary motive for representing the physical message communication (MC1) abstraction on the same basis as the higher level MC abstractions, is to recognize that, even at the level of physical communication between P_procs, there are two considerations:

* The capacity of the queue can be such that it can accommodate a message from all the existing processors of that level.
**Parameter Interpretation** :-

- `sse_m_n` : identifies the SSEM and unsynchronized queue of message containers, uniquely associated with each receiver processor.

- `bpr_n` : identifies the sender B_procr.

- `msg` : the message contents: an 'opcode' and input parameters OR the return parameters.

- `af` : acknowledgment flag.

- `bmc_n` : identifies the 'base message container' used to contain the message.

**Operations** :-

- `WRITEM_MRI_MSG (sse_m_n, bpr_n, msg, af, bmc_n)`

- `READM_MRI_MSG (sse_m_n, bpr_n, msg, af, bmc_n)`

- `WRITEM_MRI_ACK (bmc_n, msg)`

- `READM_MRI_ACK (bmc_n, msg)`

---

*Fig. 4.6 The Operations defined by the 'Base Message Manager' (MRL)*
SEND_MC2_MSG (s: ssem_n, m: msg, ag: af, bmc_n⁺) :-
begin SEND_MC2_MSG
b = BPID(bpr_n)
1m = WRITEM_MR2_MSG(s,b,m,ag,bmc_n⁺)
SIGNAL_VP2_SSEM(s)
return(1m)
end SEND_MC2_MSG

RECEIVE_MC2_MSG (s: ssem_n, bpr_n, msg, af, bmc_n⁺) :-
begin RECEIVE_MC2_MSG
WAIT_VP2_SSEM(s)
return ( READM_MR2_MSG(s,bpr_n,msg,af,bmc_n⁺) )
end RECEIVE_MC2_MSG

SEND_MC2_ACK (b:bpr_n, m:msg, lm:bmc_n) :-
begin SEND_MC2_ACK
WRITEM_MR2_ACK(lm,m)
UNBLOCK_VP2_SBL(b)
end SEND_MC2_ACK

RECEIVE_MC2_ACK (lm: bmc_n, msg) :-
begin RECEIVE_MC2_ACK
b = BPID(bpr_n)
BLOCK_VP2_SBL()
return ( READM_MR2_ACK(lm,msg) )
end RECEIVE_MC2_ACK

Fig. 4.7 The Implementation of the MC2 Operations
1) A receiver \texttt{P\_procr} has to perform a form of arbitration in choosing between simultaneous invocations.

2) A process can be blocked at the \texttt{P\_procr} level when it invokes an operation implemented by a dedicated \texttt{P\_procr} ('hardware box').

In the actual hardware implementation, the VPl abstraction can cause the blocking of \texttt{P\_procrs} through 'cycle stealing'. Since the VPl abstraction is implemented physically in a distributed manner, the synchronization operations are implemented as the transmission of physical signals. Furthermore, message containers are implemented with physical registers, perhaps using a buss structure, and unsynchronized queues are implemented using physical buffers. The salient point is that in describing the MC abstractions so as to accentuate the analogy between the three levels, we have been able to model the \texttt{P\_procr} interactions, although these are implemented at the level of physical circuits.
Section IV.6 The Interpretive Role of the VP Abstractions

A virtual processor interprets a user process by executing the sequence of operation invocations that it specifies. These operations are defined at the external architecture interface. As we have seen, the implementation of these operations may consist of operation invocations on lower levels of abstractions, which in turn may cause further operation invocations. The interpretive role of a processor is to execute each externally invoked operation by invoking the operations which constitute the implementation of the external operation. A main benefit of structuring the architecture as a set of mainstream abstractions, is that it allows us to partition the processor's interpretive role.

Each abstraction, whether it is a mainstream abstraction, or a primitive abstraction (composing the extension mechanism of a mainstream abstraction), has an interpreter identified with it. An interpreter interprets the implementation of its associated abstraction. In the case of mainstream abstractions, this implementation is specified by the abstract implementation. The interpreter for a mainstream abstraction interprets the abstract implementation by sequencing and executing the required operation invocations on primitive abstractions or lower level mainstream abstractions.

The meta-implementation of an abstraction (mainstream or primitive) can take either of two forms. One, the abstraction can be implemented on the same processor as the invoker of the abstraction. Alternatively, the abstraction can be implemented on a dedicated processor, separate from the processor used by the invoker. This implies that, in executing an operation invocation, a processor has to transfer control of execution either to an interpreter located in the same processor, or to an
interpreter located in a separate dedicated processor. We will now consider the implications of the second kind of execution transfer on the design of the virtual processor subsystem.

If a processor has to execute an operation invocation on an abstraction implemented on a dedicated processor, it transfers execution by a synchronized message communication. It uses an appropriate message communication (MC) abstraction for this purpose. The appropriate MC abstraction is the one which corresponds to the processor level of the dedicated processor, implementing the invoked abstraction. For example, if the invoked abstraction is implemented on a dedicated P_procr, then the MCl (physical mg. commun.) abstraction is used to communicate between the P_procr being used by the invoker and the dedicated P_procr. The key idea here is that, as far as the higher level processors of the invoker are concerned, the synchronization at the lower level (VP1, in this case) is not visible.

A user process is defined at the external architecture interface and executes (virtually) on a System processor (S_procr). The S_procr itself executes (virtually) on a B_procr, which in turn executes (physically) on a P_procr. In invoking (an operation defined by) an abstraction implemented on a dedicated processor, the synchronization between the invoker and invoked processors is at the level of the dedicated processor. The implication of the synchronization is that the invoker processor's execution may be blocked at that level, although the blocking is not visible to the virtual execution at the higher levels. For example, if the synchronization is at the VP2 level, then the S_procr (VP3) is still executing virtually, although the B_procr (VP2) may be blocked and be without a P_procr (VP1) component. The crux of the meta-
implementation is that the interpretation of an operation invocation is such that the required message communication is implicitly handled at the appropriate processor level.

The message corresponding to an operation invocation consists of an 'opcode' identifying the operation and the set of input parameters. If return parameters are expected from the operation invocation, then the 'af' flag is set to 1. This indicates that an acknowledgement is expected. The sender processor is blocked while waiting for the acknowledgement. The receiver processor, on which the abstraction defining the operation is implemented, receives the message and performs the required action. It then returns an acknowledgement message containing the return parameters. This causes the sender processor to be unblocked.

The communication protocol allows the sender processor to continue executing without waiting for an acknowledgement, if no return parameters are expected*. The message communication abstraction guarantees that messages sent between the same two processors are received in the order that they were sent. There is no guarantee made with respect to different processors sending messages to the same receiver processor. However, recollect that the abstract implementations were designed such that no assumptions were made with respect to concurrent operation invocations on the same primitive abstraction.

We have seen that the set of interpreters defined within the architecture are of two kinds:

* The disadvantage is that the bound on the SSEM semaphore count and the capacity of the unsynchronized queue, will both have to be decided on a probabilistic basis rather than on a worst-case basis.
1) Those corresponding to abstractions which are not implemented on a dedicated processor. These interpreters are collected together for each general purpose processor.

2) Those corresponding to abstractions which are implemented on dedicated processors. Each dedicated processor contains one such interpreter.

Each interpreter, whether it is mainstream or primitive, has an interpreter of one of these two kinds associated with it. A general purpose processor has to contain all the interpreters of the first kind. However, a subset of these processors may be restricted to be used to execute higher level dedicated processors, and, hence, these need not contain all the interpreters. Processors at each level contain the interpreter for the message communication (MC) abstraction at that level. Since a virtual processor uses a lower level processor for its execution, a processor also contains the interpreters for the MC abstractions defined at the levels higher to it. A dedicated processor contains the interpreter for the single abstraction which is implemented on it, as well as the interpreter for the corresponding MC abstraction. In addition, if the abstraction implemented on the dedicated processor uses other mainstream and primitive abstractions, then the corresponding interpreters are included in the dedicated processor, if they are of the first kind.

The state of a processor is defined as the union of the states of the interpreters that it consists of. Since an interpreter interprets the abstract implementation of the operations defined by its corresponding abstraction, its state specifies the current execution point with respect to the abstract implementation of an operation. This exe-
cution point can correspond to a pending operation invocation defined on some other abstraction. In this case, the invoked interpreter is also in the state of interpreting an abstract implementation. Thus, at any point in the execution of an operation visible at the external architecture interface, a chain of invoked interpreters may be in the state of interpreting their corresponding abstract implementations. We call this state of an interpreter as a non-null state. If the interpreter is not in the state of interpreting its abstract implementation, it is then said to be in the null state.

A processor state is specified as the non-null states of its interpreters and the invocation sequence of these interpreters. If a processor is in the state of having invoked an operation implemented on a dedicated processor, then the last non-null interpreter state in the invocation sequence corresponds to the message communication interpreter used in the processor communication. A P_procr state can have any of its constituent interpreters in a non-null state. Although B_procrs and S_procrs are virtual processors extended from the physical P_procrs, there is a restriction with respect to the interpreters which can be non-null in their respective processor states. We will now examine the nature of this restriction.

The MCI (physical msg. commun.) abstraction manages the physical message communication at the P_procr level, and is used in communicating with the VP2 abstraction, which is implemented on a dedicated P_procr. Since the VP2 abstraction accesses inactive B_procr states, the state of the MCI interpreter cannot be non-null in the B_procr state. This is because the VP2 abstraction should not be able to affect its meta-implementation. Similarly, since the VP3 abstraction is implemented on
a dedicated B_procr, the MC2 interpreter state cannot be non-null in an
S_procr state.

There is another restriction with respect to the S_procr state.
The VP3 abstraction uses the address space abstraction of the virtual
memory. Therefore, the interpreters corresponding to the address space
and the paging abstractions (VM2, VM4) cannot be non-null for an inac-
tive S_procr. The total architecture itself presents an interface which
serves as an interpreter for user-defined processes. The operations
invokable at the architecture interface are called 'externally visible'
operations. User processes can only consist of invocations of these
operations. Thus, the state of a user process cannot contain non-null
states for any of the interpreters composing the entire architecture.
The analogy to a conventional machine architecture—the state of a user
program cannot include the state of the micro machine (interpreter)
which defines the hardware processor.

The VP2 and VP3 abstractions are implemented on a dedicated P_procr
and B_procr, respectively. The meta-implementation of these abstrac-
tions requires mechanisms to interact with these abstractions. These
mechanisms are defined by the MC1 and MC2 abstractions, respectively.
Therefore, the meta-implementation of the VP3 abstraction uses the VP2
abstraction, the meta-implementation of the VP2 abstraction uses the
VP1 abstraction. The question, then, is how does a processor interact
with the VP1 abstraction, and which abstraction is responsible for the
interaction mechanism? We will now discuss this issue.

The VP1 abstraction is implemented in a distributed manner, at the
level of physical logic circuits. Hence, its interaction mechanism is
defined at the level of the physical circuit abstractions. This reso-
olution can well be extended to the atomic level. The essential idea is that, because of the distributed implementation, each P_procr contains a portion of the VP1 abstraction. All these VP1 portions collectively implement the VP1 abstraction. A P_procr interacts only with the VP1 portion defined within it. However, this VP1 portion may then communicate with other VP1 portions (through physical signals) in order to carry out the action of the VP1 operation. This implementation is not visible to the invoker of the VP1 abstraction. A P_procr wait status is specially defined for a P_procr which has invoked a VP1 operation and is waiting for an acknowledgement. This status is called W_VP1.
Section IV.7  The STOP processor operation

The special implementation of the STOP operation is necessitated by the fact that the state of a stopped processor (in the W_STPx status) is accessible to the next higher level VP abstraction. It is this VP abstraction which invokes the STOP operation on the lower level VP abstraction. For example, after the VP3 abstraction invokes a STOP_VP2 operation, the state of the stopped B_procr is accessible to it. Therefore, this processor state should not contain non-null states for any of the interpreters which interpret the abstractions used by VP3. This implies that the STOP_VP2 action has to be delayed until the B_procr has completed execution of the current address space (VM4) and paging (VM2) operations. In addition, the B_procr should not be in the middle of a base message communication (MC2).

At the VP1 level, the STOP_VP1 action is to be delayed until the P_procr completes a physical message communication (MC1). At the VP3 level, since the stopped status of an S_procr allows the S_procr state to be accessible through external operations, none of the interpreter states internal to the architecture can be non-null. Therefore the STOP_VP3 action has to be delayed until the completion of the currently executing external (architecture interface) operation.

We impose a stronger requirement on the STOP_VP2 operation. This is necessary because of the use of 'busy waiting' with respect to the internal locks used in the abstract implementations. The rationale behind the use of busy waiting is as follows. Busy waiting is only used in the implementation of an operation (defined by some abstraction) if a qualitative analysis (as in Appendix 2b.5) shows that it is unlikely that the executing processor will be blocked in the implementation of
that operation, after some of the internal locks have been locked. Now, the STOP_VP2 operation is invoked by the VP3 abstraction, in order to implement the multiplexing policy for the virtual execution of S_procrs. The resultant blocking at the VP3 level is independent of the abstract implementation execution*. Therefore, in order for the busy waiting justification to remain valid, we impose the requirement that the STOP_VP2 action be delayed until the completion of the current external operation's execution. The action is similar to STOP_VP3 in this regard.

The same argument can also hold for delaying the STOP action at the VP1 level. However, there are two reasons against delaying the STOP action at this level. First, the blocking imposed because of the VP2 multiplexing is on a much smaller time scale as compared to that imposed by VP3. Therefore the justification for busy waiting is still valid with respect to the virtual execution at the VP2 level. The second reason is more crucial. Since there are only a small number of P_procrs, it is possible for all P_procrs to be busy waiting in the middle of executing an external operation. If the STOP action at the VP1 level is to be delayed until the completion of the external operation's execution, then the STOP action will not have an effect on any of the executing P_procrs, thus causing a system wide deadlock. At the VP2 level, there is a much larger number of B_procrs. Therefore it can be ensured that the maximum number of processors which can busy wait on any lock is less than this number.

Even if the STOP action at the VP1 level is not to be delayed, there is still the need of a VP1 operation with which the VP2 abstrac-

* Note that the requirement that the current address space and paging operations be completed before the blocking at the VP3 level, still holds.
tion can implement the STOP_VP2 action. This VP1 operation will have to
detect the completion of the current external operation's execution. Be-
cause of these dual requirements, the VP1 abstraction defines two opera-
tions which perform a STOP action at the VP1 level.

The STOP_VP1 operation is used in implementing the STOP_VP2 action.
It causes the VP1 abstraction to detect the completion of the external
operation's execution. This completion is signalled as an ACKSTOP_VP2
pseudo-invocation. The STOP_VP1 operations implements a delayed STOP
action.

The HALT_VP1 operation implements the undelayed STOP action. This
operation is only used by VP2 in multiplexing the P_procrs between
B_procrs. In implementing the STOP_VP2 action, VP2 invokes the STOP_VP1
operation each time it schedules the B_procr which is to be stopped.

The consequence of the delay in the completion of a STOP operation
is that its meta-implementation should be handled in a different manner.
The standard communication protocol is infeasible, because the invoking
processor, which itself implements a higher level VP abstraction, would
be blocked until the STOP action is completed. So what is required is
a form of interaction with which the STOP operation is invoked without
an acknowledgement, but the completion of the STOP action is signalled
back to the invoker.

Although a processor at some level may have had a STOP operation
invoked on it through its managing abstraction, it may still invoke the
synchronization operations of the manager VP. For example, a B_procr
with a STOP_VP2 operation invoked on it, can still block itself by in-
voking synchronization operations on the VP2 or VP1 abstractions. How-
ever, after being unblocked, the B_procr is stopped as soon as the exe-
cution of the current external operation is completed. It then enters the W_STP2 status.

The implementation details of the STOP operations are discussed in Appendix 3c. The essence of the implementation is the use of an operation called ACKSTOP (acknowledge STOP) which is 'pseudo-invoked' on the higher level VP abstraction which required the STOP action. The implementation relies on the fact that the STOP operation, defined at each of the three processor levels, is invoked only by the next higher processor abstraction. For instance, the STOP_VP1 operation is always invoked by the VP2 abstraction and the STOP_VP2 operation is always invoked by the VP3 abstraction. If a user defined scheduler is implemented on a dedicated S_procr, then it is the only invoker of the STOP_VP3 operation. Therefore, the VP abstraction which implements the invoked STOP operation, signals the completion of the STOP action by simulating an ACKSTOP operation invocation on the invoker. For example, the VP2 abstraction signals the completion of a STOP_VP2 action by simulating a pseudo-ACKSTOP_VP3 operation invocation on the VP3 abstraction.

The important idea in the ACKSTOP pseudo-invocation is that it does not constitute a dependency of the lower level VP abstraction on the higher level one. The justification for this claim is that it is the lower level VP abstraction which implements the synchronization required for any operation invocation on the higher level VP abstraction. So all that the lower level VP abstraction does is to use the appropriate message manager abstraction (which is anyway lower to it) and execute the synchronization actions for communicating a message. Since message communication at a processor level corresponds to an operation invocation at the next higher processor level, this sequence of actions simulates
an ACKSTOP invocation (on behalf of the stopped processor) on the VP abstraction which invoked the STOP action. As an example, the VP2 abstraction, in simulating the ACKSTOP_VP3 invocation, writes an 'ACKSTOP_VP3' message into a message container managed by the MR2 (base msg. manager) abstraction. It then executes the actions for a SIGNAL_VP3_SSEM operation such that the invoker of the STOP_VP2 operation (that is, VP3) receives the message as if from the stopped B_procr. However, at the VP3 level, this is considered as an ACKSTOP_VP3 invocation. Each VP abstraction, on receiving an ACKSTOP invocation, performs the actions which it would have performed if it had waited for the STOP operation to complete. The reader is referred to Appendix 3c to convince oneself that the ACKSTOP pseudo-invocation does not violate the dependency relationship's partial ordering.

There is one final issue which needs to be resolved. The definition of the STOP operation specifies that the execution of the operation leaves the processor in the W_STPx status. On the other hand, since the completion of the STOP action is indicated by an ACKSTOP invocation from the stopped processor, the stopped processor is in a wait status corresponding to a message communication—W_SBLx(VPy)*. The solution to this conflict in status is to equate the two. Hence, the wait status of a sender processor, when it communicates with the processor on which a VP abstraction is implemented, is equivalent to the W_STPx status at that level. Specifically, the W_SBL1(VP2) status is equivalent to the W_STP1 status and the W_SBL2(VP3) status is equivalent to the W_STP2 status.

Additionally, if a user defined scheduler is implemented as a dedicated

* Recollect that W_SBLx(----) corresponds to a status class. A status in this class is identified by the identity of the receiver processor.
S_procr, the W_STP3 status can be equated to the W_SBL3(—) status corresponding to communicating with this S_procr.

The scenario for processor multiplexing is as follows. The VP abstraction performing the multiplexing invokes the STOP (or HALT) operation on the next lower level abstraction. The completion of the action is indicated by an ACKSTOP pseudo-invocation. The invoking (stopped) processor's status now allows the processor state to be accessed. The VP abstraction stores this processor state in a memory container and loads another processor state, corresponding to a processor which has been scheduled for execution. The key idea is that the non-null message communication interpreter state is maintained with the lower level processor. For example, the VP2 abstraction after completing the STOP_VP2 action simulates a MC2 message from the stopped B_procr to the VP3 abstraction. The non-null interpreter state is maintained with the B_procr. The VP3 abstraction can schedule another S_procr to execute on this B_procr. The B_procr execution is started by invoking the RUN_VP3 operation. The effect is as if an acknowledgement for the ACKSTOP_VP3 invocation has just been received from the B_procr on which VP3 is implemented. Since initial allocation or creation of a P_procr, B_procr or S_procr causes the processor to be in the W_STPx status, the corresponding initial processor states are such that the respective message communication interpreters are in the non-null state of waiting for an acknowledgement to an ACKSTOP pseudo-invocation. Therefore, the RUN operation always has the effect of acknowledging the ACKSTOP invocation, and causes the processor to resume execution.
Section IV.8  The System Processor Abstraction (VP3)

The VP3 abstraction supports an unlimited number of System processors (S_procrs), which are of the create/destroy kind. The operations defined by the abstraction are specified in Fig. 4.8. The lifetime of an S_procr is defined by the CREATE and DESTROY operations. A newly created S_procr is in the W_STP3 status. The RUN operation causes the S_procr to enter the W_RUN3 status and be queued for execution (allocation of a B_procr). The STOP operation, as we have already discussed, initiates the STOP action on an executing S_procr. The completion of the STOP action is signalled by an ACKSTOP pseudo-invocation. An S_procr in the W_STP3 status, has its processor state accessible through a GET and SET pair of operations. There are three interrogative operations—QRUN, QEXEC and QEMAX—which return the number of S_procrs in the RUN3 and E_RUN3 status and the maximum number allowed in the E_RUN3 status. The last value indicates the number of B_procrs which are available for executing the S_procrs.

There are four synchronization operations, which are used by the System Message Communication (MC3) abstraction, to provide for synchronized interaction between S_procrs. These operations are SIGNAL_SSEM, WAIT_SSEM, BLOCK_SHL and UNBLOCK_SHL. Their functions are similar to the analogous VP2 operations.

The VP3 abstraction also supports a fixed number of 'exclusion semaphores' (ESM). These serve the same purpose as monitor semaphores, in that only one S_procr is allowed to 'execute within the semaphore'. Each ESM consists of a binary semaphore value, the identity of the S_procr executing within the semaphore, and a queue of S_procrs waiting for execution within the semaphore. The ENTER operation blocks the invoking
CREATE_VP3_PR (Spr_n)
DESTROY_VP3_PR (Spr_n")

RUN_VP3_PR (Spr_n")
STOP_VP3_PR (Spr_n")

GET_VP3_PS (Spr_n", spr_state)
SET_VP3_PS (Spr_n", spr_state)

QRUN_VP3_NP ( #(S_procs in RUN3) )
QEXEC_VP3_NP ( #(S_procs in E_RUN3) )
QEMAX_VP3_NP ( max.#(S_procs in E_RUN3) )

SIGNAL_VP3_SSEM (ssem_n)
WAIT_VP3_SSEM (ssem_n)
BLOCK_VP3_SBL ()
UNBLOCK_VP3_SBL (Spr_n")

ALLOCATE_VP3_EXS (esem_n+)
FREE_VP3_EXS (esem_n)
ENTER_VP3_EXS (esem_n)
EXIT_VP3_EXS (esem_n)

BLOCK_VP3_EBL ()
UNBLOCK_VP3_EBL (Spr_n")

Fig. 4.8 The Operations of the VP3 Abstraction
S_procr in the W_EXS status, if there is an S_procr already executing within the semaphore. The EXIT operation unblocks the first S_procr in the associated queue and allows it to execute. Both the ENTER and EXIT operations change the binary semaphore value, if there is no S_procr executing in the semaphore, or if the queue is empty. Exclusion semaphores are, themselves, limited in number and, hence, have the ALLOCATE and FREE operations defined for them.

Exclusion semaphores are meant to be used for the implementation of user visible synchronization mechanisms. The implementation of the user visible synchronization mechanisms also requires a BLOCK, UNBLOCK pair of operations to block and unblock S_procrs. Hence, VP3 provides an 'external block list' (EBL) on which these operations are defined. The external synchronization mechanisms use these two operations. The W_EBL status corresponds to the blocked status in the EBL. Fig. 4.9 illustrates the effect of the VP3 operations on an S_procr's status.

The VP3 abstraction is implemented on a dedicated B_procr. Therefore, the invocation of a VP3 operation is implemented as a message communication between B_procrs and is managed by the MC2 abstraction. The primitive abstractions it uses to maintain information about S_procrs are similar to those used by the VP2 abstraction. The VP3 abstraction stores inactive S_procr states in the blocks composing address spaces. The VP3 abstraction has an internal scheduling policy for deciding on the B_procr allocation to the S_procrs.

A subset of the S_procrs may be used as dedicated processors. The System Message Communication (MC3) abstraction provides for the interaction with these dedicated S_procrs. One use of a dedicated S_procr that we had suggested, is in the implementation of a user defined scheduler which sched-
Fig. 4.9 Status Transitions in VP3
ules the user processes at the architecture interface. Note that an accessible S_procr state does not have any of the internal interpreters in a non-null state.

Section IV.9 Summary

In this chapter, we have described the virtual processor subsystem as consisting of three levels of processor abstractions. We have described the three levels so as to show the similarity between them. We have shown that two other abstractions serve as limiting abstractions in this analogous relationship. The level of physical circuits forms the lower limit and the level of user processes forms the upper limit.

Each processor abstraction is defined such that there is a different grain of execution for the processors that it manages. At the level of the VP1 abstraction, the P_procrs are always physically executing except for the blocking imposed by the physical communication. At the next level of the VP2 abstraction, VP2 performs micro scheduling to implement the virtual execution of B_procrs. At the next higher level, VP3 performs medium term scheduling, on a larger time scale, so as to implement the virtual execution of S_procrs. At the architecture interface, each user process makes use of its own S_procr to execute its actions. However, only the S_procrs which are not in the W_STOP status, are virtually executing. Therefore, even if each user process has its own S_procr, the execution rate of S_procrs can be controlled through the use of the RUN_VP3, STOP_VP3 operations, by a user defined scheduler.

We have shown how the virtual processor subsystem plays a vital part in the interpretive role of the architecture. The large number of virtual processors encourages the implementation of abstractions on dedicated virtual processors, so as to best utilize the concurrency in exe-
cution. Special hardware implementation of abstractions are equivalent to implementation on dedicated physical processors. We showed that operation invocations on abstractions implemented separately on dedicated processors, physical or virtual, correspond to a message communication at the level of the dedicated processor. We were thus able to model these interactions on the basis of the processor synchronization operations. A significant feature of the virtual processor design is that the virtual processor abstractions are, themselves, implemented on dedicated processors and, hence, rely on the interpretive role of the lower level processor abstraction.

Each level of processors has a message communication (MC) abstraction defined for inter-processor communication at that level. The MC abstraction invokes the synchronization operation defined by its corresponding VP abstraction. Since the two higher level VP abstractions are implemented on dedicated processors, each MC abstraction depends on its lower level MC abstraction for invoking the VP synchronization operation. Therefore, the dependency relationships between the MC abstractions arises because of its meta-implementation and not because of extension from the lower level abstraction. This subtle distinction is the reason for not grouping the MC abstractions as a separate subsystem composed of a mainstream of abstractions.

The MC abstractions are used only in processor communication internal to the architecture and are not visible at the architecture interface. The synchronization mechanisms visible at the architecture interface are discussed in the next chapter. These mechanisms can be implemented using the 'exclusion semaphores' supported by VP3.

We identified a special requirement in the implementation of the
STOP operation, used in stopping a processor's execution. The special requirement was brought about because a processor's state in the stopped status is accessible to the next higher processor abstraction. We then imposed a stronger requirement which required that a STOP operation at the VP2 and VP3 levels, be effective only after the completion of the current external (architecture interface) operation execution. However, this is more a point of view. An alternative solution is to replace the busy waiting in abstract implementations with the use of synchronization operations supported by the appropriate processor abstractions. The tradeoff is in the cost of invoking the synchronization operations at execution time, and more important in the additional information which the processor abstractions would have to maintain. In any case, the weaker requirement which causes the STOP VP2 action to be delayed until the current address space and paging operations are completed, still holds.

The virtual processor design uses a unique scheme for upward signaling. This is through the ACKSTOP pseudo-operation invocation by which a lower level processor abstraction can signal the completion of a STOP action to the invoker of the operation. We feel that this is a better solution than an implementation based on 'event counts' as in [Ree 76]. The other use of 'event counts' to unblock a processor blocked at a higher level, from a lower level, is unnecessary in our case because of the modeling of the virtual memory - virtual processor inter-dependencies. We showed how these dependencies can be ordered, so as to solve the 'unlimited memory space, unlimited number of processors' problem.
CHAPTER V

THE ARCHITECTURE INTERFACE

Section V.1 Introduction

An architecture interface can be defined as a set of operations. These operations are invokable by the programs defined on the architecture. In a conventional architecture, these operations are called the instruction set of the 'hardware processor'. The 'micro program' defines the implementation of these instructions. A 'micro machine' serves as the interpreter for these internal implementation programs. The set of programs defined external to the architecture constitute the software system implemented on the architecture. The operating system is the set of programs which hide the limitations on the physical resources supported by the architecture and support a set of virtual resources. The lower layers of the operating system define a set of functions in terms of the processor instruction set. These functions are then used as primitives by the higher levels of the operating system and by the rest of the software system.

The defect in the above approach to designing a system is that there is a linear ordering imposed on the implementation methodology. This implies that all abstractions other than the one defined by the architecture's instruction set must be implemented as programs external to the architecture. Furthermore, all these programs have to use the architecture as their interpreter. Thus the abstractions defined for the operating system are constrained to an implementation through programs defined on the architecture. The point is that this restriction on the implementation of abstractions can be removed.

In the preceding two chapters, we presented the design of the vir-
tual memory and virtual processor subsystems. These subsystems form the lower layers of the operating system. The design was based on the use of a structuring technique—type extension—which decomposed the subsystems into primitive component abstractions. The significant feature of the design was that we chose a viewpoint of type extension which allowed us to concentrate on the 'component' relationships of the internal structure. These issues are called the abstraction issues of the design. The actual implementation of the component abstractions require the recognition of the additional relationships within the internal structure. These issues are called the meta-implementation issues of the design.

The main difference between system design on a conventional architecture and our approach to system design arises out of the difference in the treatment of meta-implementation issues. In the conventional approach, the meta-implementation of abstractions assumes that the architecture serves as the sole interpreter for the implementation programs of abstractions. In our approach, we assume the existence of an interpreter for each abstraction. The advantage of this approach is that the nature of the interpreter can be radically different for different abstractions. An extreme example—an abstraction can be implemented as a specialized 'hardware box'. It is not necessary for all the interpreters to be located in one processor. Abstractions which require similar interpreters can use the same physical interpreter, although each abstraction is still regarded as having its own interpreter. Our generalized approach to the meta-implementation of abstractions accommodates all forms of implementation. We are thus able to remove the restriction in implementation methodology. Furthermore, we now include, by the term 'architecture', all the subsystems whose implementations are
no longer restricted to the use of a single interpreter. This is the
crux of our 'architectural philosophy'.

With the above definition of architecture, we can state that the
virtual memory and virtual processor subsystems are to be incorporated
within the architecture. The architecture interface that we present is
defined as a set of operations. These operations are partitioned into
subsets, where each subset defines a 'type' of object supported by the
architecture. The abstraction which defines the architecture operations
is called as the ARCH abstraction. The ARCH abstraction is equivalent
to the abstraction defined by the microprogram in a conventional archi-
tecture. We have illustrated the analogy between our architecture
(O_arch) and a conventional architecture (C_arch) with a set of equa-
tions (Fig. 5.1).

The ARCH abstraction defined at the architecture interface defines
the interpreter for all 'software systems' external to the architecture.
The architecture interface presented to these systems is similar to the
virtual machine defined by the lower layers of an operating system. The
advantage is that the implementation internal to the architecture can be
optimized for maximum effectiveness of performance. A key feature of
the architecture interface is that it is oriented towards the support
of the linguistic abstraction features that were discussed in Chapter II.

We have neglected an important aspect of designing a practical archi-
tecture—the input/output subsystem. We have assumed that the architec-
ture's interaction with external systems is confined to the use of the
addressing structure. This is a rather restricted viewpoint in any prac-
tical environment.

In the next section, we will present the total internal structure
Defining Equations:

Any arch. = Control Program + Internal Interpreter + Memory

Internal Interpreter = Sequencer + Executor

A Conventional Architecture:

C_arch = Microprogram + Micro-machine + Real Memory

Micro-machine = Sequencing Logic + Executing Logic

Our Architecture:

O_arch = ARCH abstraction + Virtual Interpreter + Virtual Memory

Virtual Interpreter = VP3 Sequencer + set of interpreters (one for each abstraction)

Fig. 5.1 A Set of Architecture Equations
of the architecture. Section 3 discusses the meta-implementation issues internal to the architecture and introduces a simulation technique which helps the making of implementation decisions. Section 4 discusses the functionality of the ARCH abstraction. This functionality defines the operations which are visible at the architecture interface. Section 5 specifies the addressing structure of the architecture. We view the addressing mechanisms as constituting the interaction mechanism between the external programs and the architecture. Section 6 ties together the bottom-up approach of Chapters III, IV and V with the top-down approach of Chapter II by showing that the architecture satisfies the linguistic abstraction requirements. The last section summarizes the important features of our architecture design and highlights the uniqueness of our approach.
Section V.2  The Internal Structure of the Architecture

The virtual memory and virtual processor subsystems are the two main subsystems incorporated into the architecture. We had shown that these two subsystems are structured as a set of mainstream abstractions. The extension mechanism between two successive mainstream abstractions in a mainstream is decomposed into a set of primitive abstractions. In extending from a lower level mainstream abstraction, abstractions defined in other mainstreams are sometimes used. The virtual memory mainstream abstractions also use the abstractions defined in a simplified file system mainstream.

The ARCH abstraction defines the set of operations visible at the architecture interface. These operations are partitioned into subsets, where each subset contains the operations defined for a type of objects. These types visible at the architecture interface are called 'architecture types' (arch types). The ARCH abstraction extends from the address space (VM4) abstraction and supports segment objects (VM5) and an implicit form of variable address space objects (VM6) at the architecture interface. The ARCH abstraction defines 'process' objects at the architecture interface. Each external process is identified as a process object and executes on a system processor (S_procr defined by VP3) that is uniquely associated with it. There are other arch types defined by the ARCH abstraction. These are also extended from the VM4 and VP3 abstractions.

The ARCH abstraction is actually a set of abstractions, where each abstraction extends from the virtual memory or virtual processor mainstream or from both. Since the objects defined by these abstractions are available external to the architecture, the names identifying these
objects have to be sealed within tamperproof containers called capabilities. A map abstraction is required to maintain the attributes of objects belonging to each arch type. The map is defined on the corresponding name_type. Collectively, these maps are called capability maps.

We had shown that our interpretation of type extension allows us to structure the incorporated subsystems, by concentrating on the abstraction issues of the design. The 'component' relationship is the basis for this structuring. Fig. 5.2 illustrates the internal structure of the architecture, based on the component relationship. The ARCH abstraction extends from the address space (VM4), system processor (VP3) and group object manager (GOR) abstractions in defining the set of arch types. It uses the capability maps to contain information about the visible objects identified by capabilities.

The structure of Fig. 5.2 illustrates the interdependencies between the mainstream abstractions. None of the abstractions exhibit a component relationship with any of the three levels of message communication abstractions (MC1, MC2 and MC3). These MC abstractions are used only in the meta-implementation of abstractions. Note that each MC abstraction extends from the appropriate virtual processor (VP) and message manager (MR) abstraction but not from the lower level MC abstraction. The dependency of a VP abstraction on its corresponding MR abstraction arises because of its use in the simulation of the ACKSTOP_VPx pseudo-invocation.

An abstract implementation can be specified for each of the mainstream abstractions of Fig. 5.2. We had shown this for the virtual memory abstractions. An abstract implementation can also be specified for the operations defined by the ARCH abstraction. Thus, the entire intern-
Fig. 5.2 The Internal Structure of the Architecture
KEY:

VM00: Processor register memory
VM0: Real Memory
VM1: Partitioned real memory
VM2: Paging
VM3: Partitioned paged memory
VM4: Address space (block memory)

VP1: Physical processor
VP2: Base processor
VP3: System processor

HNR: Home name manager
SS: Secondary storage

MC1: Physical message communication
MC2: Base message communication
MC3: System message communication

MR1: Physical message manager
MR2: Base message manager
MR3: System message manager

Fig. 5.2 (contd.) The KEY for "Internal Structure of the Architecture"
al structure of the architecture may be defined on the basis of an abstract implementation.

The definition of an architecture by means of an abstract implementation is a powerful design tool. It formalizes the internal structure of the architecture. This serves as a basis for the verification of the correctness of the operations defined by the architecture. Since the architecture defines a secure capability based environment, the security of software systems defined on the architecture relies on the security properties of the architecture's operations. These can be guaranteed on the basis of a correct abstract implementation. The use of a software based methodology in the internal design of the architecture allows a host of semi-automated tools to be used in the internal structuring. Finally, the abstract implementation programs defining the internal structure can be used in the performance analysis of the architecture. More important, they can be used in conjunction with simulation studies as a means of tuning the performance of the architecture.
Section V.3 The Meta-Implementation

a) The meta-implementation issues

We have defined the meta-implementation issues of an abstraction as the issues arising out of the additional three forms of the dependency relationship. In the architecture design based on an abstract implementation, we have been concerned with only the 'component' (and 'map') relationships between abstractions. The three additional forms of the dependency relationship are caused by the dependency on the abstractions supporting the program, the address space and the interpreter used by the abstraction (refer to Section III.2e).

Since the abstractions of the virtual memory subsystem primarily provide a mapping function, the majority of the components of the abstractions are maps. Therefore, in our modeling of the internal structure, the map dependency is considered as part of the component dependency. This holds for the virtual processor abstraction also. For the ARCH abstraction, the set of capability maps provide the map functions and are considered to be components of the ARCH abstraction.

The program and address space dependencies are combined together, since our design of the memory abstractions includes the notion of address spaces. We will call this combined dependency as the program space dependency. The program space dependency is closely related to the choice of the interpreter dependency and constitutes a sub-issue after this choice has been made. Previously, we had distinguished our architecture philosophy on the basis that we identified an interpreter with each abstraction. The main meta-implementation issue is the choice of the interpreter dependency for each abstraction. There are two forms of interpreter implementation for an abstraction. For each form,
there are three sub-issues which have to be considered in the meta-
implementation.

The interpreter for an abstraction can be located either centrally
on a dedicated processor, or duplicated over all the processors which
can execute operation invocations on the abstraction. In the latter
case, each of the processors contains an identical copy of the inter-
ter. These are the two forms of interpreter implementation. The choice
of the form of interpreter implementation is not arbitrary. It is dic-
tated by the nature of the abstraction for which the interpreter is de-

defined. If the abstraction is such that it executes only one operation
at a time, although there may be simultaneous operation invocations,
then it is better suited towards an interpreter located on a dedicated
processor. On the other hand, if the abstraction is designed to allow
the concurrent execution of its operations, it is better suited for a
duplicated interpreter implementation. These are not hard and fast
rules. An abstraction implemented using duplicated interpreters can en-
force the restriction of no concurrent operation executions by using an
abstraction wide lock. Alternatively, an abstraction implemented on a
dedicated processor can receive simultaneous invocations and cause the
concurrent execution of some of them.

Another aspect in the choice of the interpreter implementation is
cost. If an interpreter requires specialized hardware for its implemen-
tation, it is more economical to localize it in one processor. The trade-
off is with the loss of concurrency of execution. A third aspect in the
choice of the interpreter implementation is the internal structure of
the abstraction. If the abstraction maintains an abstraction wide data
base, then duplication of the interpreter requires that all interpreters
be made aware of modifications to the data base. Therefore, the maintenance of consistency imposes an execution overhead.

For either form of interpreter implementation, there are three sub-issues concerning the meta-implementation. One is the efficiency of implementation for the interpreter. The second is the efficiency of the program space abstraction. For the centralized form of interpreter implementation, it is the efficiency of the interpreter implementation which is the significant issue. For the duplicated form of interpreter implementation, it is the efficiency of the interaction mechanism and the efficiency of the abstractions supporting the program space which are of significant interest. We will now justify these claims.

In the discussion of the virtual processor subsystem, we had considered the role of the subsystem in the meta-implementation of abstractions. We had shown that a processor can be dedicated at any of the three levels of processor abstraction. A message communication mechanism is defined at each processor level and constitutes the interaction mechanism for dedicated processors at that level. The important meta-implementation issue for a centralized interpreter implementation is the choice of level for the dedicated processor. The interaction mechanism is automatically decided by this choice. The memory abstractions supporting the program space can be any of the abstractions which are lower (in the partial ordering structure) than the abstraction being implemented. With the current day trend in hardware technology, a majority of the abstractions implemented on dedicated processors would be chosen to be implemented at the physical processor level. We had equated the notion of a 'hardware box' implementing an abstraction to a dedicated physical processor. Even after such a choice has been made, the effici-
ency of the interpreter's hardware implementation still remains to be
decided. The implementation choices are numerous—the use of a simple
processor, the use of standard logic circuits, the use of specialized
logic circuits, the use of special hardware technology, etc. The point
is that the nature of the physical communication mechanism remains the
same for all choices of interpreter hardware. Also, for all these
choices, the program space is constrained to be implemented either on a
real memory partition (VM1) or on the register memory in the processor
(VM00).

If an abstraction is implemented using duplicated interpreters,
then a copy of the interpreter exists in every processor which can exe-
cute an operation invocation on the abstraction. In the last chapter,
we had seen that a physical processor (P_procr) which is used in the
multiplexing of higher level general purpose processors, contains a set
of interpreters for all the abstractions which are not implemented on a
dedicated processor. If a P_procr is restricted to be used in the mul-
tiplexing of higher level dedicated processors, then it must contain
interpreters for all the abstractions, which are invokable by the ab-
straction implemented on the dedicated processor. For example, the VP3
abstraction is implemented on a dedicated base processor (supported by
VP2) and can invoke the address space abstraction. Therefore, any
P_procr which is used in this base processor's multiplexing has to con-
tain the interpreter for the address space abstraction. The important
issue in the case of the duplicated form of interpreter implementation
is not the efficiency of implementation as all interpreters are assumed
to have a common hardware implementation in the physical processor.
The important issues are the efficiency of interaction with the interpre-
ter and the efficiency of the program space abstraction.

We view the invocation of an operation on an abstraction as the transfer of execution control from the interpreter of the invoker abstraction to the interpreter of the invoked abstraction. If the latter interpreter is in a dedicated processor, we had seen that this transfer is implemented as a synchronized message communication. If the two interpreters are located in the same processor, then there is a variety of choices for the interaction (transfer) mechanism. One choice is for the two interpreters to be physically implemented as one interpreter and to implement the execution transfer as in a conventional procedure call mechanism. This requires a temporary storage space to maintain the previous state of the common physical interpreter and provide the parameter space for the parameters of the operation. A stack implementation using a lower level memory abstraction can satisfy these requirements.

An alternate means of interaction between two interpreters which share a physical interpreter is by means of macro-substitution. In this scheme, the code for the invoked operations is inserted into the implementation code of the invoker abstractions, at each point of invocation.

A third alternative is for the two interpreters to be physically distinct. The hardware for each interpreter maintains the state of the interpreter. In this case, the implementation issue is the choice of memory space for the parameters. This could be through a lower level memory abstraction or through the internal registers of the processor. Since, in most cases, the parameter space requirement is small, the latter seems to be a more promising solution. The choice of a memory abstraction to support the parameter space is an important implementation
issue, also in the case of the interpreter being implemented in a dedicated fashion.

In the previous discussion, if a memory abstraction is used in the interaction between two interpreters, the abstractions for which the interpreters are defined should be higher (in the partial ordering) than this memory abstraction. The use of a memory abstraction in the interaction mechanism between two interpreters implies that rather than a direct transfer of execution, the transfer occurs through the interpreters defined for the memory abstraction. This behaviour is also true of the message communication mechanism. Here the additional synchronization is implicit within the communication mechanism. We had seen that the synchronization for the message communication at one processor level, also necessitates synchronization at lower processor levels. A significant point is that the choice of the interaction mechanism is a meta-implementation issue of the abstraction which is invoked, and not of the invoker abstraction. A case in point—the level of the message communication to be used in invoking an abstraction implemented on a dedicated processor, is decided by the level of the dedicated processor.

The second important implementation issue, given the choice of a duplicated interpreter implementation, is the choice of the memory abstraction supporting the program space of the abstraction. The implementation programs are stored and accessed by invoking the operations of this memory abstraction. The range of choices for the program space abstraction parallels the choices for the parameter space. Any lower memory abstraction can be used to contain the implementation programs for the abstraction. The most efficient implementation is to use a pro-
gram memory space implemented in the processor itself (VMOO). A lesser scale of efficiency is obtained by using a real memory partition to contain the program space. Progressively less efficient choices are by using higher level virtual memory abstractions. Again, there is a tradeoff between cost and performance. Another aspect of the program space dependency is the memory space required for local variables used in the implementation of an abstraction. As we had seen in the abstract implementation programs, the fine level resolution of abstractions results in a very minimal memory space requirement for this purpose.

To summarize, we have identified the key meta-implementation issue of an abstraction as being the choice of the form of implementation for the interpreter of the abstraction. We showed that there are two alternate forms of interpreter implementation—a centralized interpreter located in a dedicated processor or duplicated interpreters, one for each processor. In the latter case, we had observed that it was possible for more than one abstraction to share a physical interpreter. We identified three sub-issues common to either form of implementation. These sub-issues concern the efficiency of the interpreter itself, the efficiency of the interaction mechanism to the interpreter, and the efficiency of the program space used by the abstraction. We then showed that, for the centralized interpreter case, the first issue is of predominant interest, while for the duplicated interpreter case, the last two issues are of significance.

When a conventional architecture is used as the basis for the design of a system, the range of choices for the software implementation is limited. The conventional architecture defines the universal interpreter for all abstractions. Although a virtual processor subsystem,
designed over the architecture, can support virtual copies of this interpreter, the choice of a specialized and dedicated interpreter is not possible. Furthermore, all interaction mechanisms between abstractions are based on the use of a standard procedure call mechanism. A virtual memory subsystem designed over the architecture, can support virtual memory abstractions, but the choice of a program space located on the processor itself is not possible. Implementation of system software is restricted even further to the use of a physical processor and real or intermediate stages of virtual memory.

The conventional notion of a *microprogrammed* implementation corresponds to the use of a duplicated form of interpreter. This interpreter is the micro-machine, and the interaction mechanism is through micro-procedure calls. The program space is supported using the processor's internal registers.

A *hardware* implementation corresponds to the use of a dedicated micro-machine interpreter. The physical message communication abstraction (MCI) supports the interaction mechanism, and the program space is local to the dedicated interpreter.

In essence, our separation of the design and meta-implementation issues removes the linear ordering of implementation methodology, characteristic of conventional systems. The significance of our model of the architecture's internal implementation is the recognition that a full range of choices is available for the implementation of the component abstractions. We have shown that conventionally recognized forms of implementation utilize only a subset of these choices. The implication is that, although the partial ordering of the 'dependency' relationship has to be obeyed, there is NO other ordering imposed on the implementation.
External to the architecture, the software resembles conventional systems in that the architecture serves as a duplicated form of interpreter for the external abstractions.

b) The dependencies of the mainstream abstractions

We have already discussed the inter-dependencies between the mainstream abstractions arising out of the component relationship. The resultant internal structure of the architecture was illustrated in Fig. 5.2. In the previous subsection, we identified the meta-implementation issues of an abstraction. We will now discuss these issues with respect to the abstractions composing the architecture.

In the design of the abstractions of the virtual memory mainstream, we had constantly emphasized the concurrent execution of the operations of the abstraction. Since most programs use the memory abstractions, concurrent streams of execution utilize this feature. Therefore, the interpreters of the virtual memory abstractions are best suited for the duplicated form of implementation. In the design of the virtual processor subsystem, we had shown that the virtual processor abstractions serve as a junction point for the concurrent streams of execution. Hence, their interpreters are best implemented in a centralized manner. At the physical processor level, the hardware separation constraints require a distributed implementation. However, this distributed implementation is at the level of physical circuits and is distinct from the duplicated interpreter form of implementation. Even in this case, we showed that the physical processor abstraction can be modeled on the same basis as the higher level processor abstractions. The two higher level virtual processor abstractions are implemented on dedicated processors thus utilizing the centralized form of interpreter implementation.
All of the primitive data abstractions composing the extension mechanisms for the virtual memory abstractions execute one operation at a time. Therefore, these abstractions are adapted towards a centralized form of interpreter implementation. Since these abstractions are frequently invoked, their interaction mechanism should cause minimal synchronization overhead. Hence, the best choice of the level of their dedicated processor is that of physical processors. A primary motivation for the fine resolution of the component abstractions in the virtual memory was to identify primitive abstractions which can directly correspond to specialized hardware implementations. The map abstractions and the managers of map objects confirm this point of view. Some of the map abstractions can be implemented to execute independent operations in parallel, although this feature is not directly visible to the users of the abstraction. For example, independent operations on the associative maps can be executed concurrently with operations defined on the regular maps. Map manager abstractions can use the lower level mainstream abstractions for their implementation. For example, the manager of passive block maps (used in VM4) can use the paging abstraction and through it the secondary storage abstraction to maintain most of the inactive map objects.

We had shown that the primitive abstractions composing the mechanisms extending between virtual processor abstractions, can be implemented local to the dedicated processors, used by the processor abstractions. We have also identified the role of the message communication abstractions in the interactions with abstractions implemented on dedicated processors. These communication abstractions satisfy the two objectives of an interaction mechanism—parameter passing and synchronization. The
interpreters for the message communication abstractions are required to be present in all processors, dedicated and general purpose, since each kind of processor requires to communicate with the other kind. However, for a dedicated processor, the communication abstractions defined for processor levels higher than that of the dedicated processor are not required.

To summarize, the nature of the virtual memory abstractions suggests that they be implemented with the duplicated form of interpreters. The nature of the virtual processor abstractions suggests that they be implemented with the centralized form of interpreters. For this subsystem, each virtual processor abstraction is implemented on a dedicated processor supported by the next lower level processor abstraction. We had also shown that the characterization of the primitive abstractions for the virtual memory subsystem suggests that their interpreters be of the centralized form. In addition, because of the crucial nature of their performance behaviour, these abstractions are oriented towards an implementation at the physical processor level. Towards this end, these abstractions (maps and manager of maps) have been identified with the view of showing their equivalency to specialized hardware. Message communication abstractions have their interpreters duplicated in all processors because of the role of these abstractions in operation invocation. Finally, since the ARCH abstraction supports concurrent streams of execution at the architecture interface, its interpreter is implemented in the duplicated form.

In essence, we have been able to identify the form of interpreter implementation for all the significant mainstream abstractions. For each form of implementation, there are three sub-issues that need to be
resolved. The relative significance of the sub-issues depends on the choice of the interpreter's implementation form. The question is how do we resolve these issues? The multiplicity of choices for each sub-issue compounds the problem. What is required is a new approach to performance analysis of architectures based on our philosophy of meta-implementation. A more heuristic approach is to utilize simulation techniques. We outline the nature of one such technique in the next subsection.

c) A simulation technique

The simulation technique assumes the existence of an abstract implementation for all the mainstream abstractions of the architecture. It also assumes that the choice of the form of interpreter implementation—centralized or duplicated—have been made for each abstraction. As we have seen, this choice can be made on the basis of the nature of the abstraction. Once this choice has been made, a simulation based on this technique provides the information with which the sub-issues of the meta-implementation can be resolved.

We had identified three sub-issues concerning the meta-implementation. These are the efficiency of the interpreter, the efficiency of the interaction mechanism and the efficiency of the program space. The simulation consists of:

1) Modeling the efficiencies of implementation as identified by the three sub-issues.

2) Executing the abstract implementation programs.

3) Measuring the effect of varying the efficiencies of implementation on the performance of the architecture.

We will now describe a simulator which allows the efficiencies to be
modeled and varied.

We had seen that for a centralized form of interpreter, the efficiency of the interpreter is the dominant issue. The simulator defines a set of concurrent processes. The scheduling of a concurrent process is based on a priority assigned to it. We can then model the three levels of processors by deciding on an appropriate ratio between priorities. Each abstraction using a dedicated processor is modeled as executing one of these simulation processes. If the dedicated processor is at one of the two higher virtual processor levels, then its simulation process is assigned the fixed priority chosen to model processors at that level. However, if the dedicated processor is at the physical processor level, then we want to study the effect of varying the efficiency of hardware implementation. In this case, the priority of the corresponding simulation process is a parameter of the simulation, and can take values in a given range.

A subset of simulation processes are chosen to model the general purpose processors. The priority assigned to these processes is chosen as an appropriate weighted average of the priorities chosen for the three levels of processors. The weights are, themselves, simulation parameters because they model the number of processors supported at each of the processor levels. The simulator defines a universal message communication mechanism between all sender processes and all receiver processes. The sender processes are the processes modeling general purpose processors, while the receivers are those modeling dedicated processors. The simulator provides for a delay parameter in the message communication mechanism. A delay parameter value is identified with each receiver process and is the same for all communication with
the receiver process. The delay parameter is used to model the efficiency of the message communication mechanism. A range of delays is identified with each level of message communication. Values in the range model the efficiency of the memory containers used to contain the messages and to pass parameters.

Each receiver process executes the implementation program of the abstraction which it models. For primitive abstractions at the physical processor level, these programs correspond to the hardware implementation. A few simulation processes model dedicated processors which use special interaction mechanisms. The Frame Freer is an example. In this case, the implementation program corresponds to an abstract implementation and invokes operations on other abstractions. Therefore, these processes are treated as sender processes. The simulator, in defining the simulation processes, models the physical processor abstraction (VP1) itself. The higher level virtual processor abstractions are modeled by their individual receiver processes. Since we are modeling the effect of synchronization and scheduling independent from the virtual processor abstractions, these abstractions need not implement these operations. Hence, each virtual processor abstraction can be implemented as a pure receiver process, since they need not invoke operations on any other abstractions.

The abstractions implemented with duplicated interpreters are modeled on each of the sender processes used for the general purpose processors. These processors execute the implementation programs of all these abstractions. If the abstraction is a mainstream one, then the implementation program is the abstract implementation program. In order to model the efficiency of interaction between interpreters located in
the same processor, the simulator defines a set of delays, which can be used to delay any process. Finally, in order to model the program space efficiency, the priority of the process can be changed within a range, whenever a particular abstraction's implementation program is being executed.

To summarize, the requirements of the simulator are:

1) Support a set of concurrent processes, partitioned into sender and receiver processes.

2) Schedule these concurrent processes, based on the priority of each process.

3) Support a universal message communication mechanism between all sender and receiver processes.

4) Provide for a delay parameter for the universal communication mechanism.

5) Provide for a set of execution delays, which can be used by all processes.

Given the above requirements, we showed that we can model the meta-implementation issues as follows:

1) Model the efficiency of the centralized interpreter form by using a process with an appropriate priority.

2) Model the efficiency of the duplicated interpreter form by using a set of sender processes, whose priority is decided by a weighted average of priorities.

3) Model the levels and efficiency of message communication abstraction by choosing appropriate delay parameters for the universal communication mechanism.

4) Model the efficiency of interaction between interpreters
on the same processor by using the execution delays.

5) Model the efficiency of program space by dynamically changing the simulation process's priorities, when executing within the corresponding abstraction.

The abstract implementation programs play a key role in the simulation, for they define the interactions between abstractions. The entire internal implementation of the architecture can be simulated, if an abstract implementation exists for all the internal abstractions. The driving input for such a simulation is a program model for invocations at the architecture interface. The relevant performance characteristics can be measured. The most important benefit of the simulation is that the effect of varying the meta-implementation efficiencies can be studied with respect to the architecture's performance. Based on this, the various efficiencies of implementation can be decided. This performance tuning is the main feature of our simulation technique.

To justify the viability of the simulation technique, we implemented a simplified version of the simulator and simulated the paging abstraction (VM2). The simulation was performed on a mini-UNIX [RiT 74] operating system running on a PDP 11. This simulation experiment is summarized in Appendix 4a.
Section V.4  The Operations at the Architecture Interface

In this section, we will discuss the operations which are visible at the architecture interface. These operations are classified into subsets where each subset defines the operations for an 'architecture type' (arch type). We call the abstraction which collectively defines all the architecture operations as the ARCH abstraction. In effect, the ARCH abstraction consists of the abstractions which define the arch types. There are ten such arch types. Apart from the operations defined for these arch types, a set of visible operations are defined on pointer addresses. These will be discussed in the next section.

The ARCH abstraction extends from the virtual memory VM4 and virtual processor VP3 abstractions. The components of the extension mechanism are the group object manager (GOM), and the set of capability maps defined for the objects of each arch type. The group objects play a crucial part in the architecture's support of arch objects. The names identifying arch objects are sealed within capabilities. Hence, any operation invocation at the architecture interface requires the presentation of a capability. This capability identifies the arch object on which the operation is invoked. This capability should possess the access right corresponding to the invoked operation. The capability also contains the identity of the arch type to which the arch object belongs. All the CREATE operations defined for the creation of arch objects require a 'type capability' which authorizes creation of objects of a particular arch type.

The operations of the ARCH abstraction define the 'instruction set' of the architecture. We need to emphasize that the ARCH abstraction is a set of abstractions. In a conventional architecture, the architecture
interface supports only one type of object—data. At our architecture
type, there are ten types of objects supported, one of which cor-
responds to a conventional data object. The nature of the arch types
is such that they support the design of an object based system. A key
aspect of this design is an efficient implementation for objects, as
would be required in a facility which supports linguistic abstraction.
a) The 'Segment' arch type

The segmentation abstraction (VM5) extends from the address space
abstraction (VM4). It uses the group object manager (GOR) as a compon-
ent abstraction. Group objects maintain bookkeeping information for seg-
ments which are grouped together. Segments which are grouped together
use the same address space for their implementation. A segment can also
be implemented using a separate address space. The significant features
of the address space abstraction are that it supports sharing of blocks
across address spaces as well as supports a page limiting mechanism with
respect to each address space. These two features are utilized in the
design of the segmentation abstraction. We have already discussed many
of the implementation aspects of segments in Chapter III.

A segment can be one of several 'segment types' (s_type). The
s_type of a segment restricts the kind of elements which can be contained
by the segment. An object based system recognizes two kinds of elements,
data and capability. We recognize two additional kinds of elements in
our system. One is the code element which contains the information about
an operation invocation at the architecture interface. The other kind is
the pointer element, which defines direct addressability to memory space.
We recognize this as a separate kind of element, because its existence
is strictly controlled by the architecture. The five s_types defined and
The restrictions which they enforce in terms of the elements which can be contained in the segment, is shown in Table 5.1.

The creation of a segment requires the corresponding type capability (Tp cap.). Since there are two basic forms of segment implementation—sharing an address space and using a separate address space—there are two forms of the CREATE operation, which are defined for segments. In the first case, the implementation kind of a segment, and a capability identifying a group object are required by the CREATE operation. Both forms require the maximum and initial lengths of the segment to be specified at the time of segment creation. If the segment is associated with a group object, the CREATE operation returns a group index. This cannot be used in any operation but serves only as an indicator of the number of group indexes that have already been allocated. Both forms of the CREATE operation return a capability identifying the segment.

There is a provision for sharing copies of a segment across many groups. A separately existing segment can also have a shared copy in a group of segments. We will consider the implications of shared copies later. Because of the sharing of segments, there are two forms of the DESTROY operations defined. One form can only be used to destroy either a separately existing segment or a segment which is not shared by any other groups. The other form destroys all shared copies of the segment. Destroying a segment consists of destroying the blocks used in the segment's implementation. However, the address space abstraction does not destroy the contents of a home if it is shared by more than one block. Therefore, destruction of a block used in a segment's implementation does not imply that the contents of the segment is no longer accessible through shared copies of the segment (which are implemented on different
<table>
<thead>
<tr>
<th>Seg. Type</th>
<th>Kind of elements allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>data</td>
</tr>
<tr>
<td>CODE</td>
<td>code, data</td>
</tr>
<tr>
<td>CAPABILITY</td>
<td>capability</td>
</tr>
<tr>
<td>RESOURCE</td>
<td>capability, data</td>
</tr>
<tr>
<td>MIXED</td>
<td>capability, data and pointer</td>
</tr>
</tbody>
</table>
sets of blocks). This brings about the necessity for two forms of the DESTROY operation. Another issue in segment destruction arises if the segment is a 'very small' segment and shares* the block used for its implementation with other very small segments. In this case, the block used by the segment cannot be destroyed when the segment is destroyed. This is flagged as a 'security deficiency condition', as discussed in Chapter III.

'Pointers' define direct addressability to a segment. The OPEN operation on a segment causes the derivation of a pointer to the logical space occupied by the segment. Since the segment may or may not have shared copies of it, there are two forms of the OPEN operation defined. If the segment has multiple shared copies of it, then the OPEN operation requires a group capability as an input. The pointer derived by the OPEN operation now points to the segment's representation in the address space associated with the group.

The information about the logical space occupied by a segment is maintained in the Segment Capability Map (SCM) (discussed in Appendix 4b). This is a map defined on the 'segment name_type' (S_n). If a segment has multiple copies existing for it, then the SCM entry contains a list of—group name, group index, and open count—tuples. The open count is incremented each time a pointer is derived with respect to the group. If a segment does not have multiple copies, then its SCM entry is differentiated by an 'mgf' flag which is zero. In this case, the SCM entry contains the address of the logical space allocated to the seg-

* The sharing of a block by two very small segments is different from the sharing of a segment across many groups. In the former case, the segments are distinct but use the same block for its implementation, such that the space used within the block remains separate. In the latter case, the same segment is implemented with different sets of blocks, such that the different sets of blocks share the same contents.
ment. Since the allocated logical space is decided by the maximum length of the segment, the SCM entry has to maintain the current length of the segment as a separate attribute. When a pointer is derived by the OPEN operation, the 'mgf' flag is first checked. If no multiple copies exist, then a pointer can be formed from the information in the SCM entry itself. Otherwise, the group object is accessed to determine the logical space allocated to the segment.

Apart from pointer formation, an important function of the OPEN operation is to cause the address space pointed to, to be activated. If other segments belonging to the group have existing pointers to them, then the address space may already be active. The 'open count' is maintained for each (segment, group) pair and is decremented by a CLOSE operation defined for segments. We will study the structure and use of pointers at a later stage.

Each separately existing segment has its own address space object which is used for its implementation. Therefore, such a segment has a maximum and current page limit value associated with it. The page limit value limits the number of blocks which can be simultaneously active in the address space. The segment also has a local policy, identified by a policy name, which identifies the block deactivation policy local to the segment. All these parameters of a separately existing segment can be modified by SET operations defined for these segments. The maximum length of a separately existing segment can also be changed by a SET operation. The current length of any segment can be changed by the INCLEN and DECLEN operations. The INCLEN operation increases the length of a segment, but only up to the maximum length value. The DECLEN operation decreases the length. Since a partially used block cannot be
destroyed, this operation returns an 'sdf' flag which flags a security
deficiency condition. The segment operations introduced till now are
specified in Fig. 5.3.

For segments which are grouped together, the operations to change
the page limit values are defined on the group objects. Similarly, the
changing of the maximum length value affects the group object, since a
new logical space has to be allocated to the segment. The operations to
add a shared segment copy to a group and to delete such a copy from a
group are, likewise, defined jointly on the segment and group object.
We will postpone discussion of operations which jointly affect segments
and groups to the next subsection.

b) The 'Group' arch type

The objects managed by the group arch type are slightly different
from the objects managed by the group object manager (GOR). The differ-
ence is that the group objects managed by GOR purely serve a bookkeeping
purpose, while the group objects defined at the architecture interface
imply the existence of an associated address space. Therefore, the
group arch type abstraction extends from the GOR and address space (Wm4)
abstractions. The capability map defined for the group name_type is the
Group Space Map (GSM) which was introduced in Chapter III.

The creation of a group object at the architecture interface implies
that an address space of the specified maximum length is also created.
The name of this address space is maintained in the GSM entry for the
group. When a group object is destroyed at the architecture interface,
the associated address space object is also destroyed. Hence, all blocks
in the address space should either be destroyed or associated with null
home objects.
CREATE_SEG_LSG (Tp_c(seg), s_type, len_max, len_curr, S_c)  
CREATE_SEG_SSG (Tp_c(seg), G_c", s_type, i_k, len_max, len_curr, 

S_c", g_ind")

DESTROY_SEG_SG (S-c", sdf)  
DESTROY_SEG_ALL (S_c", sdf)

OPEN_SEG_SG (S-c", S_p")  
OPEN_SEG_MSG (S_c", G_c, S_p")  
CLOSE_SEG_SG (S_c", S_p)

SET_SEG_PM (S_c", plim_max)  
SET_SEG_PC (S_c", plim_curr)  
SET_SEG_LM (S_c", len_max)  
SET_SEG_PY (S_c", policy_n)

INCLEN_SEG_LC (S_c", len_curr)  
DECLEN_SEG_LC (S_c", len_curr, sdf)

Fig. 5.3 The 'Segment' Operations
The maximum and current page limit values of a group object can be changed by a couple of SET operations. Another SET operation is defined for changing the policy to be used in the group's block deactivation. The count of the blocks and group indexes which have been allocated or used, or the maximum values of these two parameters can all be accessed by GET operations. The group arch type defines a COMPACT operation, with which the size of the address space associated with the group can be reduced to the number of currently existing blocks. The COMPACT operation also changes the values for the maximum number of blocks and group indexes allowed in the group to the current values. These current values are returned by the operation. The purpose of the COMPACT operation is to compact the group object after all the segments using the group have been associated with it.

We had introduced the basic purpose of a group object as being the mechanism to collect a set of segments which can have a common address space for their implementation. The utility of this feature is that segments which are going to be referenced during the same period of time can save the overhead of address space activation and deactivation. Moreover, the grouping of segments allows the overhead of small segment maintenance to be shared with other segments. Another advantage of groups is that it allows the application of a local policy for block deactivation.

Some of the defined architecture operations have a combined effect on a segment and a group object. The ADGRP operation adds a shared copy of a segment to a group. This is implemented as the allocation of a logical space in the group, so as to accommodate the segment's maximum length, and then causing the allocated blocks to share the contents
of the blocks composing the original segment. This operation does not cause the formation of a new segment capability, since the same segment name is used to identify all the copies of the segment. As we had seen, the group capability is required as an additional parameter when a particular shared copy is to be identified. The ADDGRP operation does not permit the sharing of a 'very small' segment since such a segment uses only a portion of a block for its implementation and the rest of the block may be used by other very small segments.

The segment which is being copied by the ADDGRP operation can be separately existing, or be associated with a group. In the former case, the shared copy of the segment is implemented as a 'medium' segment (multiple blocks in a group), although the original is a 'large' segment (separate address space). The DELGRP operation is defined to delete a segment's shared copy from a group.

When a segment is associated with a group, logical space to accommodate the maximum length of the segment is allocated in the group. Since adjacent logical spaces may have already been allocated to other segments, increasing the maximum length of the segment requires the allocation of a new logical space. The INCLMAX operation is defined for this purpose. This operation is restricted to be applicable only if multiple copies of the segment do not exist, as otherwise the logical spaces for all the shared copies will have to be reallocated. The implementation kind of a segment within a group can be changed by a CHANGEIK operation. Various interrogative operations may be jointly defined on a segment and a group. The CHECK operation detects the presence of a security deficiency condition for a very small segment, the GET_MGC operation returns the number of shared copies which exist for a segment.
CREATE_GRP_P (Tp_c(grp), blk_max, g_max, G_c+)
DESTROY_GRP_GP (G_c")

SET_GRP_PM (G_c", plim_max)
SET_GRP_PC (G_c", plim_curr)
SET_GRP_PY (G_c", policy_n)

a set of GET operations to get the group's attributes

COMPACT_GRP_GP (G_c", g_used, blk_used)

Fig. 5.4a The 'Group' Operations

ADDGRP_GRS_GSG (S_c", G_c", g_ind+)
DELGGRP_GRS_GSG (S_c", G_c")

INCLMAX_GRS_GSG (S_c", G_c", len_max, g_ind+)

CHANGEIK_GRS_GSG (S_c", G_c", g_ind+)

CHECK_GRS_SDF (S_c", G_c", sdf)
GET_GRS_MGC (S_c", mgf, mcount)

Fig. 5.4b Joint Operations on Groups and Segments
The operations defined jointly on segments and groups are specified in Fig. 5.4b. The set of interrogative operations is not exhaustive.

The utility of group objects in maintaining shared copies of a segment, is a useful feature. It allows the contents of a segment to be shared between different localities of reference. Our discussion has been limited to the use of group objects in sharing the implementation space of segments. We will show in the rest of this section that group objects can also be used in sharing the implementation space of other arch objects. The group object play a vital role in the design of the arch type abstractions supporting these objects.

c) The 'Procedure' arch type

A procedure object is composed of a set of code elements and a set of capability elements. The latter set contains the capabilities which identify the local objects defined for the procedure. The code elements identify the operation invocations which are to be executed by the procedure. A procedure object can have a set of entry points defined for it. Therefore, a procedure object contains an entry point map (EPM). Each of the three procedure components can be implemented as a segment. The disadvantage is that the segment indirection overhead is tripled for each operation invocation on the procedure object. Therefore, a more efficient form of implementation is required for a procedure object.

The obvious solution is to use group objects. Since any fraction or multiple of blocks can be allocated in a group object, the memory space for each of the component objects is allocated from the logical space defined by a group object. Each component object of the procedure is allocated a group index. The key idea here is that there are no segment names defined for the group indexes allocated to the component ob-
jects. Hence, the use of the group indexes is internal to the procedure object. We had seen that a group allocates three kinds of logical spaces as defined for a 'very small' segment (fraction of a block), a 'little' segment (a single block) and a 'medium' segment (an integer multiple of blocks). Depending on the size of the component object, the procedure abstraction allocates the appropriate kind of logical space. The starting and limiting addresses of an allocated logical space is maintained by the group object. However, to avoid the overhead of obtaining the information from the group object, the procedure abstraction can maintain this information about each of its component objects, in a Procedure Capability Map (DCM) entry. The DCM map abstraction is defined on the procedure name_type (Pd_n).

The address of each component object of a procedure consists of i) the group index used by the object, ii) the starting logical address of the object, and iii) the limiting logical address. This information is collectively called the component address. Fig. 5.5a shows the attributes maintained for each DCM entry. The code component contains the set of code elements, the 'lob' component contains the set of capabilities to the local objects, and the 'epm' object maps each entry point name (epoint_n) to an index into the set of code elements. If the procedure does not contain any local object capabilities or has only one entry point, then the 'lof' and 'epf' flags indicate the absence of these components. (Recollect that a '+' superscript implies that a null value is possible).

The CREATE operation defined for the procedure object requires the appropriate type capability, a segment capability identifying the code segment, a group capability identifying the group which is to be used in
DCM :

\[ Pd_n \rightarrow G-n, as_n, code_a, lof, lob_a^+, epf, epm_a^+ \]

LOB : set of Local Object Capabilities

EPM : map of the form –

\[ epoint_n \rightarrow s\_ind \]

**NOTE:** The '_a' suffix identifies an attribute as a 'component address of the form –

\[ (g\_ind, \text{starting address, limiting address}) \]

**Fig. 5.5a The Procedure Capability Map (DCM)**

CREATE_PDE_PE \((T_p\_c(pde), S\_c'', G\_c'', epoint\_max, \text{Loc Obj. 1, ..., Loc Obj. n, } Pd_c^+)\)

DESTROY_PDE_PE \((Pd_c'')\)

SET_PDE_EPT \((Pd_c'', epoint\_n, s\_ind)\)

CALL_PDE_PE \((Pd_c'', epoint\_n)\)

RETURN_PDE_PE \([Pd_c'']\)

**NOTE:** The \([...]\) indicates an implicit parameter

**Fig. 5.5b The 'Procedure' Operations**
the implementation, the maximum number of entry points, and the set of local object capabilities. Using these parameters, the procedure abstraction can create the three components of appropriate sizes. The blocks used by the code object share the contents of the code segment, thus avoiding any duplication of contents. The local object capabilities are written into the memory containers defined for the logical space allocated to the 'lob' component. The CREATE operation returns a procedure capability. The DESTROY operation destroys all the blocks used by the procedure's component objects. The entry points of the procedure can be set by a SET operation. The CALL and RETURN operations provide for the conventional transfer of execution to and from the procedure. As we will see later, each activation of the procedure requires that pointers be formed to the logical spaces used by the 'code' and 'lob' components of the procedure. The procedure operations are specified in Fig. 5.5b.

There are two advantages to a procedure implementation based on group objects. First, the small amount of memory space which may be required for the three procedure components, the local object set and the entry point map in particular, can be efficiently implemented. Note that the same group object can be used to contain segments and the representation of other arch objects also. The second advantage is that the execution overhead in accessing the component objects is reduced because of the use of a common address space. In fact, this can be reduced even further if other arch objects which are likely to be referenced in the same period, are also grouped together.

d) The 'Object Procedure set' (porocs) arch type

We had seen that the representation space of a procedure object can be collected with the representation spaces for other arch type objects.
The 'object procedure set' (oprocs) object is defined so that a set of procedures may be collected together by one group object. The set of procedures can then be used as the operation definitions for an extended object, in the object based system supported by the architecture. The need for a separate arch type for this purpose is to ensure that no other arch objects are associated with the same group object. The idea is that the blocks composing the address space of the resultant group object can be shared with all instances of the extended object.

The oprocs object is composed of a set of procedure objects. The representation of an oprocs object consists of an operation index map (OPM). Each procedure in the oprocs object is identified by an operation index. The OPM maps from the operation index (op_ind) to the representation of the corresponding procedure object. This representation, as we had seen, consists of the component addresses of the procedure's components. Hence, each OPM entry is similar to a procedure's capability map entry. The OPM object is, itself, allocated logical space in the group and is, therefore, identified by a component address. The Oprocs Capability Map (OCM) maintains a set of attributes for each oprocs name (Op_n). These are shown in Fig. 5.6a.

The oprocs object is different from the other arch type objects using groups, because the group object cannot be used to group any other arch objects. Hence, the CREATE operation does not require a group capability, but requires the maximum values of the number of blocks and group indexes in order to create a new group object. This newly created group object is used for the aprocs implementation. Since no capability exists for this group object, it cannot be accessed externally. There are two forms of the ADD operation, with which a procedure can be added
OCM:

\[ \text{Ops}_n \rightarrow G_n, \text{as}_n, \text{opm}_a, \text{op_max} \]

OPM: map of the form -

\[ \text{op\_ind} \rightarrow \text{code}_a, \text{lof}, \text{lob}_a^+, \text{efp}, \text{epm}_a^+, \text{aright}_n \]

Fig. 5.6a The Oprocs Capability Map (OCM)

CREATE_OPS_OS (Tp_c(ops), blk_max, g_max, op_max, Ops_c^+)

DESTROY_OPS_OS (Ops_c")

ADD_OPS_PE (Ops_c", op_ind, S_c", epoint_max, Loc Obj. 1, \ldots, \text{Loc Obj. n})

ADD_OPS_PC (Ops_c", op_ind, Pd_c")

DELETE_OPS_PE (Ops_c", op_ind)

SET_OPS_EP (Ops_c", op_ind, epoint_n, s_ind)

SET_OPS_AR (Ops_c", op_ind, aright_n)

CALL_OPS_PE ([Ops_c"], op_ind, epoint_n)

RETURN_OPS_PE ([Ops_c"], op_ind)

COMPACT_OPS_OS (Ops_c", g_used, blk_used)

a set of GET operations to get the group's attributes

Fig. 5.6b The 'Oprocs' Operations
to the oprocs. One form requires a procedure capability identifying the procedure, while the other form allows the procedure to be specified in terms of its components. The DELETE operation deletes a procedure from the oprocs object.

A special feature of the oprocs object is that it allows an access right name (aright_n) to be associated with an operation index. Access rights are used in the definition of extended objects. An entry point of any procedure in the oprocs set can also be entered. A couple of SET operations are defined for these purposes. A pair of CALL, RETURN operations are defined for an oprocs object. When an operation is invoked on any extended object, the oprocs object used by the extended object is implicitly addressable. This is to allow any internal procedures in an oprocs set to be called. The CALL, RETURN operations on oprocs objects are defined for this purpose. The oprocs abstraction defines interrogative GET operations to obtain information about the group object used in an oprocs's implementation.

e) The 'Extended object' arch type

The set of arch objects supported by the ARCH abstraction are the primitive objects supported by the architecture. The architecture supports a type extension mechanism which allows extended objects and extended types to be defined in terms of the primitive arch objects and the arch operations. An extended object consists of a representation (rep) component and an oprocs component. The rep component defines the extended object's representation in terms of primitive arch objects and already defined extended objects. The oprocs component defines the set of procedures which can manipulate this representation. Most of these procedures are available external to the extended object, as operations
defined on the object. The rest are internal procedures.

The group abstraction again plays a vital role in the implementation of the extended object. The rep component of the object is a shared copy of an original representation segment. The blocks allocated to the rep component are shared with the blocks composing the rep segment. A subtle advantage of this implementation is that after the extended object is created, the original rep segment can be destroyed by the restricted form of the DESTROY_SEG operation without affecting the rep component. The oprocs object, as we had discussed, uses a single group object to group a set of procedures. It also prevents any other arch objects from being associated with the group. The implementation of the operation procedures of an extended object is straightforward. The extended object abstraction only has to allocate logical space for the operations component and share the blocks with the blocks used by the oprocs object. The component address of the defining 'operation index map' (OPM) of the oprocs object is maintained in an entry in the Extended Capability Map (ECM). This map is defined on the 'extended object name_type' (Eo_n) and also contains the component address of the rep component (Fig. 5.7a).

There are three operations defined on extended objects (Fig. 5.7b). The CREATE operation requires a type capability authorizing extended object creation, a segment capability identifying the rep segment, an oprocs capability to the object procedure set, and a group capability identifying the group to be used for the implementation. The CREATE operation returns a capability identifying the newly created extended object. The DESTROY operation destroys an extended object by removing its capability map entry and destroying its components. The INVOKE operation is the most widely used operation in an object based system for it is defined
ECM:

\[ E_{o_n} \rightarrow G_{n}, as_{n}, rep_{a}, opm_{a}, op_{max}, aright_{max} \]

REP: representation component

OPM: 'operation index map' of the oprocs component

**Fig. 5.7a The Extended object Capability Map (ECM)**

CREATE_EXO_EO \( (T_{p_c(\text{exo)}}, S_{c''}, Ops_{c''}, G_{c''}, Eo_{c''}) \)

DESTROY_EXO_EO \( (Eo_{c''}) \)

INVOKE_EXO_OP \( (Eo_{c''}, op_{\text{ind}}, epoint_{n}) \)

**Fig. 5.7b The 'Extended object' Operations**
for the invocation of an operation on an extended object. The execution
of the INVOKE operation causes a pointer to be formed to the rep compon-
ent of the extended object and an execution transfer to occur to the pro-
cedure object implementing the operation. Implicit addressability to
the oprocs component's representation is also defined.

The operations defined for an extended object can each be associated
with an access right name (aright_n). An access right can be associa-
ted with more than one operation. The INVOKE operation first checks for
the presence of the associated access right in the capability, before in-
voking the operation.

We have shown that the group object can be used to collectively im-
plement the components of an extended object. Each operation invocation
on an extended object requires the following:

1) Formation of addressability to the object's rep segment.

2) Formation of addressability to the procedure's code segment.

3) Formation of addressability to the procedure's local object
capability set.

4) Formation of implicit addressability to internal procedures.

By implementing all the components in terms of the same address space,
the overhead of the above actions is significantly reduced. This is be-
cause the hidden implementation of the segments used by these components,
removes the overhead of forming the above addressability through the in-
direction of capability maps. Moreover, the group object solves the prob-
lem of implementing small objects. The same group object can be used to
implement other arch objects. A possible approach would be to use one
group object to implement all the objects in the representation structure
of an extended object.
f) The 'Extended type' arch type

An extended type defines a class of extended objects. It maintains a sample representation of its extended object and has an operation defined on it with which additional instances of the extended object can be created. In Chapter II, we had introduced the notion that a type is itself an object and has an operation defined on it. We had also seen that a class of types may, themselves, be defined as a type object. A type defining a class of types is called a 'parameterized type'. We had shown that a generalized form of linguistic abstraction can be supported by recognizing that types and parameterized types are objects and, hence, can have operations defined for them. The extended type abstraction is designed towards this end.

Each extended type is composed of a rep segment and an oprocs set defining a set of procedures. The two components can be implemented on the basis of a group object. The oprocs capability identifying the set of operation procedures defined for the extended object, is contained in the type's rep component. This oprocs set is different from the operation procedures defined for the type itself.

If the extended type defines a class of objects, then one of its operation procedures has to be defined to create an instance of the object. Similarly, if the extended type defines a class of types (parameterized type), then one of its operation procedures has to be defined to create an instance of a type belonging to this class. By convention, the operation procedure associated with the first operation index is assumed to provide this function. Depending on the nature of the extended type, an OBJCREATE or a TYPCREATE operation invoked on an extended type causes this procedure to be activated. These correspond to the
CREATE_OBJ and CREATE_TYP operations of Chapter II. Any other operations defined by an extended type definition can be invoked by the TYPINVOKE operation. This is the equivalent of the INVOKE operation defined an extended objects.

Both the OBJCREATE and TYPCREATE operations require a group capability. The corresponding group object is used in the implementation of newly created instances of an extended object or type. Note that this group object is different from the group object used by the defining extended type. The OBJCREATE operation first creates a rep component for the extended object. It then uses the type capabilities maintained in the extended type's rep component to create the representation objects for the extended object. The capabilities to these objects are then inserted into the rep component of the extended object. The capability which identifies the oprocs object defining the extended object's operation procedures, is also maintained in the type's rep component. This capability is used to make a shared copy of the oprocs object. The shared copy is used as the oprocs component of the extended object. In effect, the OBJCREATE operation defined on an extended type, uses the information maintained in the type's rep component and executes the actions for the CREATE operation defined for an extended object. The OBJCREATE operation returns a capability identifying the newly created extended object.

The TYPCREATE operation is defined on an extended type which defines a class of types. The corresponding operation procedure is specified by the creator of the extended type. This procedure uses the information in the rep component of the extended type and creates a new instance of a type belonging to the defined class. The TYPCREATE operation returns with a type capability identifying the newly created extended type.
TCM:

$$Te_n \rightarrow G-n, \ as_n, \ trep_n, \ topm_a, \ op_max, \ aright_max$$

TREP: type's representation component

TOPM: 'operation index map' of the type's oprocs component

Fig. 5.8a The Extended type Capability Map (TCM)

CREATE_TYE_TE (Tp_c(tye), S_c", Ops_c", G_c", Te_c+)

DESTROY_TYE_TE (Te_c")

OBJCREATE_TYE_TO (Te_c", G_c", epoint_n, Eo_c+)

TPYCREATE_TYE_TT (Te_c", G_c", epoint_n, Te_c+)

TPYINVOKE_TYE_TE (Te_c", op_ind, epoint_n)

Fig. 5.8b The 'Extended type' Operations
The CREATE and DESTROY operations are defined to create or destroy an extended type. Their function is similar to the identical operations defined on extended objects. The Type Capability Map (TCM) maintains an entry for each extended type and contains the component addresses of the two components—the rep and the oprocs. The structure of this entry and the operations defined on extended types are specified in Fig. 5.8.

Apart from the type capabilities identifying extended types, we had seen that all the arch types require an appropriate type capability which authorizes creation of the arch object. These type capabilities identify the primitive types of the object based system*. The essential idea in the design of the extended type abstraction is to recognize that a type is a primitive arch object and that a type can define a class of objects or a class of types. Furthermore, apart from the operation which can create an object or type instance of this class, additional operations can also be defined for an extended type.

g) The 'Process' arch type

A process object is defined for each concurrent stream of execution recognized by the architecture. Each process object has a unique system processor ($procr supported by the VP3 abstraction) on which it executes. A process is a sequence of operation invocations. In an object based system, the starting point of an execution stream is either an operation invocation on an object or a procedure activation. In the former case, the operation procedure which is activated, specifies the sequence of operation invocations for the process, in the latter case, the procedure which is activated, specifies the sequence.

---

* They are identified as Tp_c in the operation specifications, while the extended type capabilities are identified as Te_c.
The important idea in an object based system is that each operation invocation in the starting sequence causes a series of operation invocations on lower levels of component objects. These lower levels of component objects can be primitive arch objects or extended objects. Therefore, the state of a process consists of the state of the original sequence, and the state of each lower level operation invocation, caused by the operation invocation being executed in the starting sequence. In fact, internal to the architecture, the state of execution of an arch operation is itself of the same form as the external state. The internal state is maintained as a set of interpreter states. Since the architecture is the sole interpreter for the external processes, the external state is maintained as a series of nested operation invocation states. After the current operation invocation is completed, the state of this operation invocation is removed from the external state. A stack abstraction is ideal for maintaining such a nesting of invocation states. Hence, each process has a stack component object. The stack is implemented using a stack segment.

A Process Capability Map (PCM) is defined on the process name_type (Ps_n) and maintains a pair of attributes for each process. One attribute is the name of the system processor which is uniquely assigned for the process execution. The other attribute identifies the name of the stack segment which is used to maintain the process state. The stack segment is implemented as a separately existing segment. The CREATE operation for a process requires the maximum length of this segment as input. The DESTROY operation destroys the two components of the process and removes the corresponding entry from the process capability map.

Since the starting point of a stream of execution is either an oper-
CREATE_PRS_PS (Tp_c(prs), len_max, Ps_c+)
DESTROY_PRS_PS (Ps_c")

INITIALIZE_PRS_EXO (Ps_c", Eo_c, op_ind, epoint_n)
INITIALIZE_PRS_PDE (Ps_c", Pd_c)

RUN_PRS_PS (Ps_c")
STOP_PRS_PS (Ps_c")

QUNSC_PRS_NP ( #(not in W_STP3) )
QRUN_PRS_NP ( #(in RUN3) )
QEXEC_PRS_NP ( #(in E_RUN3) )
QWEX_PRS_NP ( #(in W_RUN3) )
QEMAX_PRS_NP ( max #(in E_RUN3) )

Fig. 5.9 The 'Process' Operations
tion invocation, or a procedure activation, there are two forms of the INITIALIZE operation defined to initialize the starting state of a process. The process starts execution when the RUN operation is invoked on it. The RUN and STOP operations can be used to schedule the system processor on which the process executes. These operations are exactly equivalent to the RUN and STOP operations defined by the VP3 virtual processor abstraction. A process can be destroyed only after it has stopped. Since the VP3 abstraction supports an unlimited number of system processor objects, the ARCH abstraction can also support an unlimited number of process objects without any multiplexing.

The arch operations defined on process objects are shown in Fig. 5.9. There are a set of interrogative operations corresponding to the equivalent VP3 operations. These return the number of system processors in a particular VP3 status.

h) The 'Semaphore' arch type

The semaphore abstraction defined at the architecture interface provides a primitive synchronization mechanism. The implementation of the abstraction uses the 'exclusion semaphore' and 'external block list' objects supported by the VP3 virtual processor abstraction. A semaphore object is composed of a queue of system processors (wq), an exclusion semaphore (exs), a semaphore count (sem_count) value, and a semaphore maximum (sem_max) value.

The SIGNAL operation on the semaphore first checks if the 'wq' is empty. If so, it increments the sem_count by 1, otherwise the first system processor in the 'wq' is removed and unblocked. This causes the associated process to resume execution. The WAIT operation on the semaphore first checks if the sem_count value is equal to the sem_max value.
RCM : \( Se_n \rightarrow G_n, \text{ as}_n, \text{ esm}_n, \text{ wq}_a, \text{ sem}_\text{count}, \text{ sem}_\text{max} \)

ESM : exclusion semaphore

WQ : queue of waiting S_procs

\[ \text{CREATE}_\text{SEM}_\text{SE} (Tp_c(\text{sem}), G_c", \text{ sem}_\text{init}, \text{ sem}_\text{max}, \text{ Se}_c^+) \]

\[ \text{DESTROY}_\text{SEM}_\text{SE} (\text{Se}_c", \#(\text{processes unblocked}) ) \]

\[ \text{SIGNAL}_\text{SEM}_\text{SE} (\text{Se}_c") \]

\[ \text{WAIT}_\text{SEM}_\text{SE} (\text{Se}_c") \]

**Fig. 5.10a** The 'Semaphore' Operations

MCM : \( Mb_n \rightarrow G_n, \text{ as}_n, \text{ ese}_a, \text{ fse}_a, \text{ mq}_a, \text{ msg}_\text{max} \)

ESE : semaphore used to wait on empty buffer

FSE : semaphore used to wait on full buffer

MQ : queue of message elements

\[ \text{CREATE}_\text{MSB}_\text{MB} (Tp_c(\text{msb}), G_c", \text{ msg}_\text{max}, \text{ Mb}_c^+) \]

\[ \text{DESTROY}_\text{MSB}_\text{MB} (\text{Mb}_c", \#(\text{processes unblocked}) ) \]

\[ \text{SEND}_\text{MSB}_\text{MB} (\text{Mb}_c", \text{ message element}) \]

\[ \text{RECEIVE}_\text{MSB}_\text{MB} (\text{Mb}_c", \text{ message element}) \]

**Fig. 5.10b** The 'Message buffer' Operations
If so, the system processor associated with the invoking process is blocked and its name entered into the 'wq'. If the values are not equal, the sem_count value is decremented by 1. The exclusion semaphore ensures exclusive access to the semaphore components.

The CREATE operation for a semaphore requires the sem_max value and an initial sem_count value as input. The semaphore uses the logical space in a group for the implementation of the 'wq'. Therefore, a group capability is also required by the CREATE operation. The DESTROY operation destroys the semaphore's components and unblocks all system processors which are blocked on the semaphore. The operation returns the number of processors which are unblocked. The attributes for each semaphore are maintained in a Semaphore Capability Map (RCM). These attributes and the semaphore operations are shown in Fig. 5.10a.

1) The 'Message buffer' arch type

The message buffer abstraction defines a primitive form of synchronized message communication. It is composed of two semaphores and a queue of message elements (these can only be a data or a capability element). The SEND operation adds a message element to the message queue, while the RECEIVE operation removes an element from the queue. The two semaphores are used for synchronization. One semaphore blocks the invoker of the SEND operation if the message queue is full; the other blocks the invoker of the RECEIVE operation if the message queue is empty. The implementation of these operations is simple using these semaphores [Han 73].

The CREATE message buffer operation requires a group capability and the capacity of the message queue. The group object identified, is used for the two semaphores' processor queue (wq) implementation, as well as
for the message queue implementation. The attributes maintained for each entry in the Message buffer Capability Map (MCM), and the operations defined on message buffers are shown in Fig. 5.10b.

j) The 'Data' arch type

The data object class recognized at the architecture interface corresponds to a standard set of data elements—integer, character, boolean, etc. The operations defined by this arch type are similar to the host of operations defined in any conventional architecture and, hence, needs no elaboration.

k) An overview

In this section, we have described the arch types defined at the architecture interface. The thrust of the architecture design has been the support of an object based system. The arch types are the primitive types defined for such a system. We have stressed the efficient implementation of objects in such a system. Towards this end, we have shown that the use of group objects serves as a universal solution for efficient object implementation. The highlight of the mechanism is that different types of objects can be grouped together, thus reducing the access and implementation space overheads. The architecture implements two broad classes of resources—memory containers and processing elements. The arch types define a wide variety of functions with which these resources can be utilized.

A specific feature of the architecture is that it supports a generalized form of type extension. The significance of the type extension mechanism is the recognition of a type as a primitive arch object. A type can define a class of objects or a class of types. This is the salient feature which is used in designing a linguistic abstraction mechan—
ism based on the architecture.

Section V.5 The Addressing Structure of the Architecture

We take the view that the addressing structure of an architecture defines the interaction mechanism with the architecture. For the internal abstractions, we had observed that the interaction mechanism for an abstraction is a sub-issue of the meta-implementation, after the form of the abstraction's interpreter is decided. Since the architecture is defined as an abstraction—the ARCH abstraction—there should be an interaction mechanism defined for it. The addressing mechanisms serve this purpose.

a) Pointers

Pointers define direct addressability to memory space. The actual reading and writing of elements contained by the segment containers defined at the architecture interface, is effected as reading and writing through pointers. Pointers pointing to the virtual memory space occupied by a segment can be derived from the segment capability by the OPEN operations. A pointer points to a region of virtual memory by specifying the starting and limiting addresses of the virtual memory space. We had seen (in Chapter III) that a pointer contains a five tuple - address space name, (block name, offset), (block name, offset) - which identifies the blocks composing the virtual memory space.

Reading and writing through pointers is defined by a pair of READ, WRITE operations. These operations require a 'segment index' (s_ind) as an additional input. The segment index is treated as a (block index, offset) pair and is used to index into the memory space identified by the pointer.

A special feature of pointers is that they enforce the restrictions
required by the 'segment type' (s_type). There are four kinds of elements recognized by the architecture—data, capability, pointer, and code. The s_type of a segment restricts the kind of elements which can be contained in the segment (refer to Table 5.2). The pointer derived from a segment contains the s_type of the segment. Hence, for each read or write through a pointer, there is an implicit check between the kind of element read or written and the s_type of the segment. Since only one of the s_types, namely MIXED, allows pointers to be contained in it, the existence of pointers is restricted by controlling the existence of MIXED segments.

The reading and writing through pointers is implemented as a read or write operation defined on the address space abstraction. Since these operations can return as a fault, the READ and WRITE pointer operations should be able to cope with the fault. If the fault is due to the block being inactive, then the block is activated. If the block activation fails because of the page limiting, then the local policy is executed and a block is chosen for deactivation. This block's activation can be switched to the original block. If the fault is due to the address space being inactive, then the address space is activated with its full page limit. If this fails, then the system wide policy is executed to choose an appropriate address space for deactivation. This address space is then deactivated and the original address space can be activated. If the read/write fault is because of the absence of a home associated with the block, then a new home can be allocated depending on the nature of the pointer. Finally, a fault because of a destroyed block can be passed back to the architecture interface.

A pointer defines a virtual memory space by specifying the starting
and limiting addresses. The pointer can be modified to identify a subset of its original space by increasing the starting address or reducing the limiting address. These functions are defined by the CHANGE operations. Note that in all cases, only a restricted form of the original pointer can be formed. A pointer contains a set of access rights defined with respect to the segments it points to. These rights can be reset (but never set) by a RESET operation. Finally, a set of GET operations are defined to display various attributes of a pointer. The operations defined on pointers are specified in Fig. 5.11.

b) **The Architecture Registers**

As in any conventional architecture, the registers defined by our architecture serve as temporary memory containers providing fast access. Some of these registers serve a special purpose and cannot be directly accessed. The rest of the registers can be accessed directly and serve as general purpose registers. Since there are four kinds of elements, the general purpose registers are partitioned to contain specific kinds of elements. There are only three such partitions, since there is no necessity to provide registers to contain code elements. Since pointers define a region of memory space, an additional set of registers are defined to contain indexes into memory space. These indexes are of the form—block index, offset pairs. In all, there are two sets of special purpose registers and four sets of general purpose registers.

The special purpose registers of the architecture are defined with the purpose of identifying the current state of execution, based on the assumption that the architecture supports an object based system. We had seen that in an object based system, each operation invocation belonging to the starting sequence of operation invocations, causes a series of
Pointers are created and destroyed by the OPEN
and CLOSE operations defined on segments.

READ_PTR_PR (S_p, s_ind, elem)
WRITE_PTR_PR (S_p, s_ind, elem)
CHANGE_PTR_START (S_p, s_ind)
CHANGE_PTR_LIMIT (S_p, s_ind)
GET_PTR_STYP (S_p, s_type)
GET_PTR_START (S_p, s_ind)
GET_PTR_LIMIT (S_p, s_ind)
GET_PTR_ACCR (S_p, aright_n, setf)
RSET_PTR_ACCR (S_p, aright_n)

Fig. 5.11 The 'Pointer' Operations
lower level operation invocations. The stack component of each process maintains the series of invocation states, up to the level of the current operation that is being executed. The stack component also contains the temporary and parameter space for the current operation execution.

The stack component is implemented as a series of stack frames (no relation with real memory frames) with only the most recent frame being accessible to the current operation execution. This frame contains the temporary and parameter spaces. The frames are linked together such that a procedure call creates a new stack frame and a procedure return destroys the current stack frame and reaccesses the second last frame. Since the stack also contains the parameter space for a procedure call, two successive frames overlap with respect to this parameter space. The stack frame and the parameter spaces are defined by a special 'stack pointer' and a pair of special indexes, 'stack index' and 'mark index'. The latter index identifies the extent of the parameter space. Stack segments are the only kind of MIXED segments allowed by the architecture. Therefore, only the virtual memory space used by stack segments can contain pointers.

The rest of the special purpose registers contain the state of the current operation execution. In an object based system, for every operation's execution, there is an object being manipulated. The 'representation' (rep) pointer points to the rep component of this object. An operation is implemented as a procedure object. Hence, a set of special pointers identify the components of the currently executing procedure. The 'code pointer' points to the code component, the 'local object pointer' points to the local object component. A special index identifies the next code element which is to be executed as part of the current opera-
tion execution. This index is defined with respect to the code pointer. Since each extended object contains an 'object procedure set' (oprocs) component, a special pointer points to the 'operation index map' (OPM) which represents the oprocs component. This pointer is used in calling internal procedures in the current object's manipulation. The different sets of registers defined by the architecture are shown in Fig. 5.12.

Pointer registers have a set of hidden registers associated with them. General purpose registers only contain pointers to segments, and for each of these segments, there is an associated hidden register containing a policy name. This name identifies a local policy which is to be used if block deactivation is necessary. There is another hidden register associated with each general pointer register. If the pointer points to a separately existing segment, then this register contains the segment name. If the pointer points to a segment shared in a group, then the register contains a (group name, group index) pair which identifies the segment. The contents of this register is necessary in reactivating the address space if it has been deactivated since the last access.

The same two hidden registers exist for special pointer registers also. For the stack pointer, the segment name is maintained in the hidden register; for all other pointers, a (group name, group index) pair is maintained. The structure of a pointer register is illustrated in Fig. 5.13a. The hidden registers associated with pointer registers are shown in Fig. 5.13b.

The current state of a process can be defined as the contents of the architecture registers. The set of architecture registers should be regarded as a separate abstraction. This abstraction has a duplicated form of interpreter implementation and a program space defined at the
a) Special Pointer Registers (SPR)

RP (rep pointer) : points to the current object's representation
LOP (local obj. pointer) : points to the current procedure's
    local object set.
CP (code pointer) : points to the current procedure's code.
SP (stack pointer) : points to the current stack frame
OMP (OPM pointer) : points to the 'operation index map' (OPM)
    of the current object's oprocs component.

b) Special Index Registers (SIR)

CI (code index) : indexes the next code element to be executed
SI (stack index) : indexes the top of the stack
MI (mark index) : index which identifies the boundary of the
    current parameter space

c) General Purpose Registers

1) General pointer registers (GPR)
2) General index registers (GIR)
3) General capability registers (GCR)
4) General data registers (GDR)

Fig. 5.12 The sets of Architecture Registers
A pointer register:

<table>
<thead>
<tr>
<th>i_k</th>
<th>aright</th>
<th>s_type</th>
<th>as_n</th>
<th>b_n</th>
<th>offs</th>
<th>b_n</th>
<th>offs</th>
</tr>
</thead>
</table>

START | LIMIT

Fig. 5.13a The Structure of a Pointer Register

Hidden Registers for a 'small' segment pointer:

| G_n | g_ind | policy_n |

Hidden Registers for a 'large segment' pointer:

| S_n | policy_n |

Fig. 5.13b The Hidden Registers for a Pointer
level of the processor register space. This is an additional interpreter state which has to be maintained by all the general purpose processors internal to the architecture.

c) **Address forms and Implicit address spaces**

An architecture requires an interaction mechanism to communicate parameters for operation invocations at the architecture interface. The operations at the architecture interface are defined by the ARCH abstraction and are likened to the instruction set of the architecture. In a conventional architecture, an instruction is specified by an 'opcode' and the parameters for the instruction set are defined by address forms, one for each parameter. Designing the addressing structure of the architecture consists of specifying these address forms. In our architecture design, we assume that each code element contains an 'opcode' identifying the operation to be invoked, and a set of address forms for the parameters of the operation.

Address forms in our architecture identify an element either in virtual memory space or in the architecture register space. For the first case, the address form consists of the name of a pointer register and an index. The index can be contained in the code element itself (immediate value) or can be contained in an index register. In the second case, the element is contained in one of the general purpose registers.

If the address form identifies an element in memory space, then the pointer register specified in it can either be a general purpose register or a special purpose register. Since each pointer register is constrained by its segment type (s_type), the use of a pointer register in an address form depends on the kind of element which is identified. The stack pointer identifies a MIXED segment (all but code elements), the
code pointer identifies a CODE segment (only code and data elements), and
the rest of the pointers identify RESOURCE segments (only data and capa-
bility elements). The address forms which can be used to identify each
kind of element is shown in Fig. 5.14. For each address form which iden-
tifies an element in virtual memory space, the index can be specified
either as an immediate value or as the name of an index register. We
have defined a special address form using the pointer to the current ob-
ject's 'operation index map'. This form is used to call internal proce-
cures.

The significance of the special pointers is that they define a set
of implicit address spaces at all stages of execution. These address
spaces are of unlimited size, since they are either implemented as seg-
ments or as blocks in an address space. In an object based system, there
are four such address spaces—the current object's representation space,
the current procedure's code space, the current procedure's local object
space, and the current parameter space. Each of the above spaces is
identified by a special pointer contained in a special pointer register.
These implicit address spaces constitute the final virtual memory ab-
straction—variable address spaces (VM6).
a) Data Element Identifier

1) immediate (in the code element itself)
2) Gpr_n + Index
3) RP, LOP, SP, CP + Index
4) Gdr_n

b) Capability Element Identifier

1) Gpr_n + Index
2) RP, LOP, SP + Index
3) Gcr_n

c) Pointer Element Identifier

1) SP + Index
2) Gpr_n

d) Code Element Identifier

1) CP + Index

e) Internal Procedure Identifier

1) OMP + op_ind

Fig. 5.14 Address Forms to identify elements
Section V.6 The Abstraction Features at the Architecture Interface

From the discussion of the operations and primitive types at the architecture interface, it is apparent that the architectural features required for the linguistic abstraction mechanisms of Chapter II are provided. To recapitulate, the two main features required of the architecture are:

1) The support of the addressing environment of an object-based system.

2) The support of ptypes and ftypes as primitive classes of objects.

With respect to the first feature, there are three issues—'vertical' protection, object addressability, and dynamic change of object addressability. The ARCH interface supports a capability based environment which ensures that an object's representation is only accessible to the procedures defined for its manipulation. The addressing structure provides for addressability to three address spaces at each point of execution. These are the current object's representation space, the current procedure's representation space, and the temporary parameter space. In addition, there is implicit addressability to all internal procedures defined by the object's type. This satisfies the second issue of addressability. Finally, the implementation of the INVOKE operation defined on an extended object, causes the dynamic change of the above address spaces.

The ARCH abstraction supports an 'extended type' as a distinct arch type. Furthermore, this type is such that it can be used to define either a class of objects or a class of types. There are two forms of the CREATE operation defined on this arch type. One form is used if the
extended type defines a class of objects; the other is used if it defines a class of types. This satisfies the second abstraction feature required of the architecture.

The design of the abstraction features at the architecture interface takes a unique approach to the implementation considerations. The support of an 'oprocs' object allows the operation procedures of a type to be implemented efficiently. A lot of the segment indirection overhead is removed by providing the facility of grouping together components of objects. The design utilizes 'group' objects to provide for the efficient support of 'small' extended objects. This is by sharing the maintenance and access overhead between the objects in the group. Thus, the design of the architecture interface alleviates much of the overhead characteristic of object-based systems.
Section V.7  Summary

In this chapter, we have defined and specified the architecture interface of our system. We first presented our 'architectural philosophy'. On this basis, we justified our claim that the virtual memory and virtual processor subsystems of Chapters III and IV can be incorporated into the architecture. We then specified the architecture interface as a set of operations. These operations were likened to the instruction set of a conventional architecture. We designed a set of address forms which formed the means of interaction with the architecture interface. We finally showed that the architecture interface meets the requirements which were established in Chapter II.

The abstract design of a system of abstractions identifies the 'component' relationships between the abstractions. (This imposes a preliminary partial ordering on the set of abstractions.) We call these issues of the design as the abstraction issues. The issues arising out of actually implementing each abstraction are called the meta-implementation issues. The uniqueness of our architectural philosophy is in the treatment of the meta-implementation issues. The key idea is to recognize that each abstraction in a system of abstractions, can have its own kind of interpreter. The interpreter for an abstraction interprets the implementation programs designed for the operations of the abstraction. In a conventional system design, there is only one kind of interpreter—that defined by the hardware architecture. In our system, we define the term 'architecture' as including all the abstractions, which have their own kinds of interpreters.

Internal to the architecture, we identified two forms of implementing the interpreter for an abstraction. For each of the two forms for im-
plementation, there are three sub-issues, which need to be considered. We showed that the relative significance of a sub-issue depends on the form of the interpreter implementation. The utility of our architecture model is that it allows one to solve the problem of architectural incorporation of software subsystems. We outlined a simulation technique which facilitates the solution of this problem. The added advantage of this technique is that it identifies a means of performance tuning of the architecture.

The architecture serves as the interpreter for all abstractions defined external to it. We defined the architecture interface as a set of operations. The uniqueness of this definition is that we presented it as the primitive level of an object based system. At the same time, these architecture operations are such that they provide a wide variety of functions which can be used in the definition of a very flexible object based system.

The emphasis in the design of the architecture interface is on the efficient implementation of objects. We designed a powerful mechanism based on group objects. We showed that group objects can be used to group the implementation space of most kinds of architecture and user defined objects. A subtle advantage of groups is that they can be used as an accounting mechanism for resource usage.

The architecture interface defines a flexible type extension mechanism. An important feature of this mechanism is the support of an extended type as a primitive object. An additional feature is that an extended type can define a class of types as well as a class of objects. The type extension mechanisms are oriented towards the design of mechanisms supporting linguistic abstraction.
The architecture supports an unlimited number of concurrent execution streams, by providing virtual copies of the architecture interface. The architecture supports primitive objects which provide for synchronization between the concurrent streams. Since these synchronization mechanisms are defined as primitive objects, they can be used in conjunction with the type extension mechanism to define more sophisticated synchronization mechanisms.

The architecture's addressing structure defines the interaction mechanisms for invoking operations at the architecture interface. The essential idea in the design of this structure is the recognition of a set of implicit address spaces which are characteristic of an object based system.
CHAPTER VI

SUMMARY AND CONCLUSIONS

Section VI.1 Summary

This thesis has presented the design of an architecture which was strongly influenced in two ways by software. First, the architecture supported the design methodology of 'abstraction', the use of which is becoming prevalent in current software systems. Second, the architecture incorporated within it, two subsystems which are usually implemented in software. The emphasis of the design was structure. The architecture interface defined a mechanism which serves as a basis for structuring all software systems, which are to be implemented on the architecture. Internally, the architecture was itself structured using a formalism of the same structuring technique. This structuring technique is the tool with which the architecture was designed. We adopted a refined interpretation of the technique, which allowed us to separate the internal design considerations of the architecture from the internal implementation considerations.

Chapter I established the motivation for this research. First, since the levels of software design are being unified with respect to a common design methodology—abstraction—it was our contention that the architecture should also support this methodology. The rationale was that the architecture should evolve from supporting 'procedural' abstractions to supporting 'data' abstractions. Furthermore, since a secure environment for user interactions is becoming an important issue, the user interface of the architecture should be defined from this point of view. The second motivation was that the decreasing cost of hardware
is making software relatively more expensive. This suggested that a systematic approach to the change in the form of implementation for software systems, be developed. A straightforward 'hardware' implementation of a 'software' subsystem was realized as not being a proactical solution. Rather, the direction which was established was to distinguish between the essential characteristics of a software implementation, a hardware implementation and an architectural implementation. The key to the approach was the structuring of the software subsystem. Chapter I reviewed the background work in the area of structured operating systems, linguistic abstraction, type extension and incorporation of software subsystems.

Chapter II developed a linguistic model which described the features for 'abstraction', which can be provided by a programming language. The emphasis was on features which allowed a very generalized form of defining parameterized types (data abstractions). These features were also provided in a dynamic form, which allowed dynamic type creation at execution time. The mechanisms which supported these features, were then designed. The set of features required of the architecture was established on the basis of this design.

Chapter III designed and structured the virtual memory subsystem. The subsystem was structured as a linear ordering of 'mainstream abstractions'. The virtual memory abstraction was characterized as the removal of a set of restrictions imposed by the real memory abstraction. Each non-trivial mainstream abstraction removed some of these restrictions from its lower level, until the final virtual memory abstraction was defined. The extension mechanism between successive mainstream abstractions was decomposed into a set of primitive components. For
two of the important mainstream abstractions, the details of the design of the extension mechanism was presented in the form of an 'abstract implementation'. The usefulness of an 'abstract implementation' is that it can be used to formally specify and verify the design of a system. It provides an insight into the structure of the system. The main advantage is that it can be used as the basis for a heuristic approach to the internal implementation of the architecture.

Chapter IV designed the virtual processor subsystem as a three-level structure of processor abstractions. At each level, a synchronized mechanism for message communication was designed. The message communication abstractions provided for the interaction between processors at that level. Each level of processor abstraction was assumed to have a different grain of scheduling. The mutual dependency between the virtual memory and virtual processor subsystems was resolved by the technique of 'sandwiching' [Par 76a] the components of the respective subsystems. Finally, the interpretive role of the virtual processor abstractions was identified. This arose because abstractions could be implemented on separate dedicated processors and, hence, required the use of the message communication mechanisms.

Chapter V first presented the total internal structure of the architecture. We identified the internal implementation considerations of the architecture as a set of meta-implementation issues. These were presented so as to characterize the nature of the implementation within the architecture and distinguished such an implementation from a software implementation. A simulation technique for resolving the meta-implementation issues was also introduced. The second half of the chapter described the external interface of the architecture. This architecture
interface was described as a set of primitive types and the operations
defined for these types. The implication was that this interface con-
stituted the primitive level of types in an object based system, defined
by the software implemented on the architecture. Towards this end, a
subset of the types defined the mechanisms to support such a system.
This chapter also defined the addressing structure of the architecture.
This structure was regarded as constituting the interaction mechanism
between the software and the architecture.

Section VI.2 Results

There are three major contributions of this research.

A technique for incorporating software subsystems within the architec-
ture

The most important result of this research is the development of a
technique for incorporating software subsystems within the architecture.
The key idea which is brought forth is the characterization of the inter-
nal implementation of an architecture. Although the net result is the
postulation of a technique for incorporation, the essential ideas are
dispersed over the actual case study.

We established the feasibility of using the technique of type ex-
tension to structure a subsystem, so that the components of the struc-
ture are small and simple. In the example of the virtual memory sub-
system, the primitive components were all maps or managers of map ob-
jects. Furthermore, by specifying the abstract implementation, we
showed that the mainstream abstractions were quite comprehensible con-
sidering the size and flexibility of the system. Most important, the
structuring technique established a partial ordering not only with the
subsystem but also with respect to interactions with other subsystems.
The first step in the incorporation of a subsystem is to use the technique of type extension to structure the subsystem into a set of simple components which are partially ordered. We adopted a refined viewpoint of type extension, which allowed us to separate the design (abstraction) issues of a subsystem from the implementation (meta-implementation) issues. The result was a structuring based on 'mainstream abstractions'. We similarly applied the technique to structure the virtual processor subsystem. We identified and ordered the interactions between the two subsystems on the basis of the structuring technique. The second step in incorporating a subsystem is to specify an abstract implementation for it.

The crucial result of this research is the characterization of the meta-implementation issues of a subsystem. We showed that out of the five kinds of dependencies between the components of a type extension based structure [Jan 76], two were due to the design issues of the subsystem, and the other three arose because of the implementation issues. Of the latter, we identified the 'interpreter' dependency as being the most important. The crux of our characterization of the internal implementation of the architecture is to recognize that each component of the internal structure has its own interpreter.

Every component of the internal structure is characterized as an abstraction. Each such abstraction has an interpreter for the programs implementing the abstraction. We showed that there are two forms of implementation for an abstraction's interpreter. The first form is a centralized implementation on a dedicated processor, at one of the three levels. The second form is a duplicated implementation where a copy of the interpreter exists in every processor which can execute an operation
invocation on the abstraction. For either form, there are three sub-
issues to be considered—the efficiency of implementation for the inter-
preter, the efficiency of the interaction mechanism with the interpreter,
and the efficiency of the program space of the abstraction. Together
with the choice of a form of implementation, the resolution of the three
sub-issues constitutes an implementation of the abstraction. After each
abstraction within the architecture is implemented, the internal imple-
mentation of the architecture is complete.

The significance of this approach to architecture implementation
is that the conventional notions of 'software' implementation, 'hard-
ware' implementation, 'microprogrammed' implementation, etc., all fall
into the class of architectural implementations. This class in fact
offers a multitude of choices and the conventionally recognized forms
of implementation constitute only a small subset of these choices. We
contend that the true impact of an evolving hardware technology is that
we can now consider the rest of the choices as viable alternatives in
the internal implementation of the architecture.

As a means of establishing a practical basis for our result, we pro-
posed a simulation technique. This technique relied on the use of an
abstract implementation for each subsystem and suggested a heuristic ap-
proach to deciding the implementation of the component abstractions.
Specifically, the technique required that a choice of one of the two
forms of interpreter implementation be made. The simulator for this
technique should model the efficiencies of the three sub-issues. On the
basis of the performance requirements, this technique can suggest a
value of efficiency with respect to each of the sub-issues. The third
step in the incorporation of a subsystem is to chose one of the alterna-
tive forms of interpreter implementation for each component abstraction. The final step is to use the simulation technique and decide a level of efficiency for each of the sub-issues of the meta-implementation.

Our modeling of the internal implementation of the architecture has two major implications. First, it suggests the feasibility of performance tuning of the architecture. This can be done by using the simulation technique and varying the efficiency of implementation for the component abstractions. The second implication is that such a model adapts to the 'family' approach for architecture implementation. In such an approach, the internal structure of the architecture remains unchanged but depending on the cost performance tradeoff, the efficiencies of implementation for the component abstractions can change.

A new design for virtual memory

As an offshoot of structuring the virtual memory subsystem to a fine level, this research generated a new design for virtual memory. Many of the new ideas have been presented while discussing the details of the design. We will summarize them here.

We designed a paging abstraction as the initial non-trivial abstraction which extends from real memory. However, we designed the abstraction to support a large but fixed number of pages. We provided a justification for this approach. The next important mainstream abstraction was the 'address space' abstraction. The distinguishing features of this abstraction was that it provided for 'page limiting' local to an address space and supported the sharing of blocks across address spaces. The 'segmentation' abstraction was designed over the address space abstraction and was visible at the architecture interface. A unique feature of this design was the use of 'group' objects to alleviate the prob-
lem of maintaining 'small segments'. Another useful feature was the sharing of the contents of a segment across many 'copies' belonging to different groups. This sharing feature played a vital role in the support of extended objects at the architecture interface.

A new approach to the support of objects and abstraction mechanisms

One reason for calling the designed architecture as software directed is that it defines an architecture interface which supports the use of the software design methodology of abstraction. A significant result in this context is the development of a linguistic model for abstraction. We showed that this model defined extremely flexible, generalized and dynamic features for abstraction, with the accent being on the definition of parameterized types.

The crucial idea in the design of the mechanisms supporting these features was the recognition that a parameterized type is a distinct class of objects. We showed that the three level tree structure of objects common to most typed systems [Jon 73, Red 74], can now be extended to a four level tree structure of objects.

The design of the abstraction mechanisms necessitated a set of features at the architecture interface. This corresponded to a top-down design of the architecture interface. A few of these required features were related to the efficient support of objects. In the bottom-up design of the architecture, we showed how the architecture can be designed to satisfy these efficiency requirements. The salient feature was the use of a mechanism which avoided repeated indirection through capabilities. Furthermore, by associating the idea of 'groups' with the support for objects, we were able to design an efficient implementation for 'small objects'. A final feature of the primitive types at the archi-
tecture interface was that two elementary synchronization mechanisms were also recognized as primitive types. This feature, combined with the type extension mechanism allowed the design of sophisticated synchronization mechanisms in the object based system.

Section VI.3 Future Research

No research is complete without generating avenues for future exploration. An exploratory research such as ours suggests several ideas which need to be pursued.

The most pressing need in the structuring of subsystems is the development of automated tools for type checking and verification. Specifically, these tools should be oriented towards the design of an abstract implementation.

There is a need for the design of a complete simulator which can be used to implement the simulation technique. The simulator should be able to model different levels of efficiency corresponding to each of the sub-issues of the meta-implementation. We propose that the design of the simulator should be in conjunction with the automated tools aiding the abstract implementation. The resulting system would be an extremely powerful tool for the design of architectures based on our philosophy. Such a combined verifier/simulator would require as input, a formal specification of the structure, a set of abstract implementation programs, and a form of interpreter implementation chosen for each component abstraction. After the initial phases of design verification, the simulator can perform the simulations and can establish a set of criteria which decide how the performance of the architecture will be affected by the efficiencies of implementation of the component abstractions.

We have considered only one aspect of predicting the performance of
the architecture. Another equally important part of this problem is performance analysis. We expect that a completely new approach for performance modeling and analysis will have to be developed for architectures which are structured internally.

One important aspect of the architecture which we have ignored is the design of input/output (I/O) mechanisms. The question is, can these mechanisms be modeled in the same way as the subsystems which were incorporated within the architecture? We feel that the structuring technique of type extension, as it stands, is inadequate to model the nondeterminancy of I/O interactions. So the first issue will have to be to propose a new technique or a new viewpoint of the same technique, which can be used to structure the I/O subsystem.

In this research, we have only considered a simplified version of the file subsystem. The assumption was that this subsystem could be incorporated into the architecture. Incorporating such a subsystem seems to be feasible according to the structure established by Janson [Jan 76]. This needs to be investigated.

In a different line of thought, the linguistic model for abstraction can be generalized even further. This is by considering a parameterized type in which the parameters of the type can, themselves, be parameterized types which do not have their parameters fixed as yet. Thus, the idea of a parameterized type being a class of fixed types can be generalized to a parameterized type being a class of parameterized types. The implication is that the objects in the system can be structured as an 'infinite' level tree structure, or at least as an 'N' level tree structure, where 'N' is very large.

With the current emphasis on distributed processing, the pertinent
question is whether or not object based systems can be supported in a distributed environment? We feel that the very fact that a type defines a closed interpretation for a class of objects, supports this application. At a different level, we have modeled the internal structure of an architecture as a set of component abstractions, each having their own interpreter. The interesting question is whether the modeling of the architecture can be generalized over a distributed environment. We see no major obstacle to this approach since our model provides for physical separation of interpreters.

The essence of our philosophy is that we recognize that each abstraction in a system of abstractions, can have its own form of interpreter associated with it. Ideally, all abstractions in the system of abstractions composing the solution of a problem, should have their own form of interpreter. However, the sheer number of different forms of interpreters which would be required negates this idea. As a compromise, at least some of the lower level abstractions should have their own forms of interpreter. These abstractions can collectively define a common interpreter for the rest of the abstractions in the system. This common interpreter is called the architecture. The conventional view of architecture assumes that the only abstractions within the architecture are those at the primitive levels, such that their interpreters can be implemented in the form of physical circuits. Our view is that all abstractions below an intermediate level can constitute the architecture. A pure theorist or a true optimist takes the view that all abstractions up to the highest level should constitute the architecture. We leave the extent of the gap between the intermediate and highest levels as a poser to hardware technology.
REFERENCES


[Hun 76] Hunt, D. H., "A case study of intermodule dependencies in a vir-


[Par 72] Parnas, D.L., "On the criteria to be used in decomposing systems into modules," Commun. of the ACM, Vol. 15, No. 12,
APPENDIX I

The Conventions for Semi-Formal Specification

Operation Specification—

An operation is specified by identifying the name of the operation and specifying (i) the types of the parameters, which are to be passed to the functional abstraction implementing the operation, and (ii) the types of the parameters, which are returned. The parameters in the former set are called input parameters and those in the latter set are called return parameters. All operation names consist of three parts, separated by a '_'. The first part identifies the 'generic' name of the operation, e.g., GET, SET, CREATE, DESTROY, etc. The second part identifies the name of the type manager abstraction which has defined the operation. The third part serves as a qualifier to distinguish between operations belonging to the same generic class and defined by the same type manager.

The most frequent generic classes of operations are the SET and GET classes which identify operations which access attributes maintained in map entries. Other common generic classes are the ALLOCATE operations used to allocate limited resources (of the allocate/free kind), the FREE operations used to free such resources, the CREATE operations used to create unlimited resources (of the create/destroy kind), the DESTROY operations used to destroy unlimited resources, the TST operations used to 'test and set' a lock, the UNLOCK operations used to reset the locks, and so on. Variations of the TST operations, which perform an additional function to the 'test and set' have a generic class name starting with 'TST'. All interrogative operations (which do not affect the state of an object) have a generic class name starting with 'Q'. As a rule, most operation names are suggestive of the purpose of the operation.
During the various discussions, operation names are referred to only by their generic class names, if the context does not warrant any further distinction. If the context does call for further distinction, the generic part and the modifier part are usually sufficient to identify the operation. Only rarely does the full three-part name have to be used. In the abstract implementations, the full three-part name is always used to identify an operation.

Parameter passing of objects is by reference. Each type of object is referred to by its respective 'name_type'. When objects are to be passed to a functional abstraction defining an operation for the type manager of the object, it is the name identifying the object which is passed. The functional abstraction then uses this name to access the representation of the object. This conversion of an object into its representation is typical of type extension based systems. Since type extension is being used as a conceptual tool for the internal structuring, this conversion merely reflects a change in the point of view. Thus, external to the functional abstraction defining the operation, the name identifying the object can only be used as an attribute maintained in a map entry by some map abstraction.

An operation is specified by specifying the types of its input and return parameters. If the parameter, which is passed, is a name, then it could either mean that the representation of the identified object is to be accessed, or that the name is to be used as an attribute for some map entry. To distinguish the former case, the name_type specification of the parameter is qualified with a """" (double quote) notation. In the absence of this qualifier, the name can only be used as an attribute.

The return parameters of an operation are identified by underlining
the parameters. Parameters which are not underlined are input parameters of the operation. Most name_types are specified using an abbreviated notation. Table Al.1a contains the abbreviations used for the more common name_types. We use the superscript '+' notation to identify that a name parameter can be a null value also. The absence of this qualifier restricts the parameter to be a non-null name.

Apart from names, the parameters of an operation can also be flags. A flag can only take a value of 0 or 1. Flag parameters are specified by using the flag name in the operation specification. Most flag names end with an 'f'. Flag names are introduced and their interpretations given, the first time they are used in a semi-formal specification. Table Al.1b identifies some of the flag names used in the specification of the paging abstraction.

There is a third kind of parameter for an operation. These correspond to primitive types—data element, integer, etc. Quite often the range of the primitive type is defined e.g., 'offs' is an integer offset, which is bounded by the size of a frame. 'Offs' addresses a data element within a frame. Table Al.1c defines the two kinds of parameters, of this third kind, which are used in the specification of the paging abstraction.

We now have sufficient information to specify the operations of the paging abstraction. For example, the specification of the READ operation is:

\[ \text{READ} - \text{VM2} - \text{PG} (p_n", \text{offs}, \text{elem}) \]

The specification states that the READ operation is defined by the VM2 (paging) abstraction. The operation requires a 'page name' and an 'offset' as input parameters and returns a 'data element'. The page name is used to access the representation of the page, implying that the operation is
### Table A1.1a  Name_type Parameters

- \( f_n \): frame name_type  
- \( p_n \): page name_type  
- \( h_n \): home name_type  
- \( as_n \): address space name_type  
- \( aas_n \): active address space name_type  
- \( b_n \): block name_type

### Table A1.1b  Flag Parameters

- \( zf \): zero flag  
- \( cf \): core flag  
- \( mf \): modified flag  
- \( rf \): reference flag

### Table A1.1c  Primitive Element Parameters

- \( \text{offs} \): offset 
  
  \[
  \text{integer} < \text{frame size}
  \]
- \( \text{elem} \): memory element
defined on a page object.

The operation specification can also specify a fault return. Additional syntax to the specification of a normal return, specifies the fault return. The fault return specification contains a set of flag names. When the fault return occurs, the appropriate fault flag is set to 1. This indicates the kind of fault return. The interpretation of a flag may be such that the fault return is caused by the flag being zero. Therefore the fault flag (which is set to 1) is a complement of the actual flag. This is denoted by qualifying the flag name with a ' ' ' (single quote) notation. For instance, if an operation, say ASSIGN, returns as a fault if the 'bf' flag is zero, the operation specification is:

\[
\text{ASSIGN (...) : fault (bf', ...)}
\]

A fault return specification assumes that there is some means of distinguishing between a normal and fault return. This could be through a special 'fault flag' which is returned with every operation. The normal return parameters are not returned if a fault occurs.

**Map Abstraction Specification**

All abstractions can be defined in terms of the operations defined by the abstraction. Since our operation specification technique is incomplete, in that it does not specify the semantics of the operation, we need to specify additional information about the abstraction. An informal verbal description provides part of this information. In the case of mainstream abstractions, the figurative specification of the structure of the extension mechanism and an abstract implementation provides the rest of the additional information. In the case of map abstractions, we can suggest a representation for the abstraction in terms of the mapping pro-
vided by the map abstraction.

A map abstraction maintains a set of map entries and is defined for a name_type. Each name in the set of names defined by a name_type, has a map entry for it. A map entry contains a set of attributes for the name. These attributes can be names, flags, integers in a defined range, or 'locks'. A 'lock' can take a value of 0 or 1, the latter corresponding to the locked state. The same map abstraction may maintain different forms of map entries for different names belonging to the same name_type domain. The different forms are distinguished by one or more flag attributes. Different map entry names can also be given to identify the different forms of map entries supported by the same map. A map entry is specified as—'domain name_type → attribute 1, attrib. 2, attrib. 3, ...'. If the types of two attributes in the same map entry are the same, the attributes may be distinguished by identifying them with names, using the syntax—'attribute ID name: attribute'.

Sometimes the attribute in a map entry serves as a link or a header with respect to a linked list of map entries. This is identified by using the '{...}' notation for that attribute, specifying the name of the linked list and also specifying whether the attribute serves as a header or as a link. In addition to the set of map entries, a map abstraction also maintains a set of attributes which belong to the map as a whole and not to any map entry in particular. A name may also be identified with these attributes. They are specified as:

'Attribute ID name (optional) : + attribute'

The header for a linked list of map entries is often maintained as such a map-wide attribute. In any of the semi-formal specifications, comments can be used freely as long as they are enclosed by '/*...*/'.

Some map abstractions also maintain a set of associative map entries. These entries are such that an associative search for a name belonging to the domain name_type can be performed given the value of a name attribute in a map entry. This is specified as a separate map within the map abstraction. The domain for this map is an 'associative map index_type', which is a set of integer indexes from 1 to the size of the associative map. One of the attributes for each of the entries in the associative map, is a name belonging to the domain name_type for which the map abstraction is defined. Another attribute is the name on which the associative search is performed. Additional attributes can be the same as those maintained in the regular map entries. Thus, the associative map maintains a subset of the 'domain name_type, associative search name_type' associations. To distinguish the regular map from the associative map, an 'R' is prefixed to the name of the map abstraction to identify the regular map, and an 'A' is prefixed to identify the associative map.

Fig. A1.1 illustrates the specification of a map abstraction. The map defined is the Frame Map (FM) abstraction, which is used by the paging abstraction. The FM abstraction maintains a regular frame map (RFM) and an associative frame map (AFM). The latter is searched on the basis of a page name attribute. The RFM consists of map entries of two forms, corresponding to the entry for a free frame (mapf_e) and that for an allocated frame (mapa_e). Free frame entries are linked together in a linked list called FFL, the header for which is a map-wide attribute. Another map-wide attribute is called REM and corresponds to an integer bounded by the maximum number of frames. The RFM entry for an allocated frame, contains the name of the page to which the frame is allocated, and a set of flag attributes. The AFM contains a subset of the frame name to page name asso-
FM abstraction :-

flag interpretation :-

ff : frame free flag
rf : reference flag
mf : modified flag
apf : associative map presence flag
arf : associative entry reference flag

representation :-

1) RFM :-

mapf_e : f_n → ff=0, \{(f_n)^+ \ FFL link\}
mapa_e : f_n → ff=1, p_n, rf, mf, apf
FFL : → \{(f_n)^+ \ FFL header\}
REM : → f_rem : integer max(f_n)
LF : → f_n

2) AFM :- /* This is the associative map */

afm_ind → f_n, p_n, mf, arf
LAE : → afm_ind

operations :-

---

Fig. A1.1 An Example of a Map Specification - The Frame Map (FM)
citations. It also contains an attribute (called LAE) which is AFM wide.

The operations for a map abstraction are specified using the technique that we have already discussed. A majority of the operations are GET and SET operations which access the attributes in a map entry. A 'lock' maintained as an attribute is accessed by the 'test and set' (TST) and UNLOCK operations. Operations defined for the associative map have their generic class name prefixed with an 'A'.

Abstract Implementation Specification—

The abstract implementation for a mainstream abstraction is a set of abstract programs. These programs define the functional abstractions corresponding to the operations of the abstraction. Additional functional abstractions which are used internally, may also be defined. Functional abstractions implementing the operations are identified by the same name as the operation. Internal function names are prefixed with a '$' character. The abstract programming language is block structured and has a set of standard control constructs (Fig. A1.2). Reserved words used by the language are underlined. There are only two kinds of primitives defined by the language—a function invocation and an assignment. The assignment primitive assigns the results of an expression to a set of variables. A function invocation may be used in an assignment expression and in a conditional expression*. A single assignment statement can specify multiple assignments.

The abstract implementation program for each operation defined by a mainstream abstraction, first specifies the input and return parameters

* The conditional operators used are: ! (NOT), & (AND), | (OR).
1) begin

end

2) if (conditional expression)

then

else

3) while (conditional expression)

4) for (index = init. to final by increment)

5) return (return values)

6) break (n), where 'n' indicates the number of nested containing loops, which have to be exited.

Fig. Al.2 The 'CONTROL' constructs
of the operation. Local names may be declared in the parameter specification itself. A declaration such as, 'local name : type of the parameter', identifies the local name with the parameter passed to the functional abstraction. An additional set of local names may also be declared. All the local names can be used as variable names in the abstract program. A local name can be defined to be of the type 'fault'. If a fault occurs, the fault is specified as an assignment of the appropriate 'fault index' to this variable. The fault index is an integer which specifies the index of the fault flag which is to be set.

The abstract implementation of an operation is specified as a block structured abstract program. The functions (operations) which can be invoked in the abstract program for a mainstream abstraction are:

1) Operations defined by the next lower level mainstream abstraction.

2) Operations defined by the external abstractions in other mainstreams. These external abstractions are 'used' by this mainstream abstraction.

3) Operations defined by the primitive abstractions composing the extension mechanism for the mainstream abstraction.

4) Internal functions defined in the abstract implementation. Internal functions are defined by abstract programs. In all the abstract programs, comments are delimited by the standard '/*...*/' delimiters.
APPENDIX 2

Appendix 2a.1. The Map Abstractions (FM & PM) for VM2

The Page Map Abstraction—

The page map (PM) abstraction (Fig. A2.1) maintains a set of attributes for each page name. These are:

1) A 'home name' which identifies the home which serves as the page's backup object.

2) A 'cf' flag (core flag) which (if set to 1) indicates that a frame is allocated to the page.

3) A 'frame name' which identifies the frame allocated for the page.

4) A 'zf' flag (zero flag) which (if set to 1) indicates that the page is ZERO.

5) A 'p_lock' which can be used to lock a page.

The PM abstraction provides a set of GET and SET operations to access the first four page attributes. A test and set operation, TST, can be used to lock the 'p_lock'. The TSTC not only provides a test and set function on the p_lock, but also returns the value of the 'cf' attribute. The UNLOCK operation resets the p_lock value to zero. The SEARCH operation first checks if the home name and page name passed to it are associated with each other. If so, it returns this page name. Otherwise, the operations scans all the page entries to check if any page name is associated with the given home name. The value of such a page name is returned if this scan is successful, else a null value is returned. The SETSEARCH operation scans all the page entries for a page name which is associated with the home name passed to the operation. If such a page
PM abstraction :-

flag interpretation :-

zf : zero flag  /* if set to 1, implies page is ZERO */
cf : core flag  /* if set to 1, implies associated frame */

representation :-

PM :-

\[ p_n \rightarrow h_n, zf, cf, f_n^+, p_{lock} \]

operations :-

GET_PM_ZF (p_n, zf)
GET_PM_CF (p_n, cf)
GET_PM_FN (p_n, f_n^+)
GET_PM_HN (p_n, h_n)

SET_PM_ZF (p_n, zf)
SET_PM_CF (p_n, cf)
SET_PM_FN (p_n, f_n)
SET_PM_HN (p_n, h_n)

TST_PM_PL (p_n, p_lock)
TSTC_PM_PC (p_n, p_lock, cf)
UNLOCK_PM_PL (p_n)

SEARCH_PM_HN (h_n, p_n^+, p_n^+)
SETSEARCH_PM_HN (h_n, p_n, p_n^+)

Fig. A2.1 Specification of the Page Map (PM)
name exists, then the home name attribute corresponding to the page name passed to the operation is changed to the new home name value.

The Frame Map Abstraction—

The frame map (FM) abstraction (Fig. A2.2) serves two purposes. First, it maintains a set of attributes for each frame name. Second, it also manages the associative map which contains a subset of the frame name to page name associations. The frame name attributes are contained in a map (RFM) which has an entry for each frame name. A RFM entry can be any of two forms, corresponding to a free frame and to an allocated frame. These are distinguished by a 'ff' flag. The entry corresponding to a free frame (ff = 0) has only one attribute, which is a frame name serving as a link in a linked list of free frames. The entry corresponding to an allocated frame (ff = 1) has four attributes:

1) A 'page name' identifying the page to which the frame has been allocated.

2) A 'rf' flag (reference flag) which is a measure of the frequency of reference to the frame.

3) A 'mf' flag (modified flag) which (if set to 1) indicates that the frame has been modified, since the last updating of the backup home contents.

4) An 'apf' flag (associative presence flag), which (if set to 1) indicates that the page to frame association is being maintained in an associative map entry.

The FM abstraction provides a set of GET and SET operations which can access the page name and 'mf' flag attributes of a RFM entry. The GETR operation returns the 'rf' flag value. The operation also resets
The first part of the specification for the Frame Map is shown in Fig. A1.1.

operations :-

GET_FM_PN (f_n, p_n)
GET_FM_MF (f_n, mf)

SET_FM_PN (f_n, p_n)
SET_FM_MF (f_n, mf)

GETR_FM_RF (f_n, rf) /* this resets 'rf' to 0 */
SUCC_FM_FN (f_n, f_n)

GET_FM_LF (f_n)
SET_FM_LF (f_n)

ALLOCATE_FM_FR (f_n)
FREE_FM_FR (f_n)
REM_FM_FR (f_rem)

AGET_FM.FR (p_n, f_n)
AGET_FM_FW (p_n, f_n) /* Sets 'mf' to 1, if successful */

ASET_FM.FR (p_n, f_n)
ASET_FM_FW (p_n, f_n) /* Sets 'mf' to 1 */
ANUL_FM_AE (p_n, f_n)

Fig. A2.2 Specification of the Frame Map (FM)
the flag value to zero, if it was 1 initially. The ALLOCATE operation
removes a frame name from the free list and returns the name, the FREE
operation adds a frame name to the free list. The REM operation returns
the number of free frames in the free list. The SUCC operation returns
the name of the next allocated frame entry (in the RFM) which does not
have its 'amf' flag set to 1.

The map wide attributes maintained by the RFM are a frame name which
serves as a header to the free frame list, a 'f_rem' value which denotes
the number of frames in this free list, and a frame name attribute called
LF. The LF value is accessed by a pair of GET and SET operations. It
serves as a reference point in the implementation of the 'modified LRU'
frame replacement policy. This policy implementation is provided by the
$PAGE_REPLACE function. This function uses the two operations defined
for the access of the LF value, the GETR operation which accesses the 'rf'
flag and the SUCC operation which can be used to scan the appropriate
frame entries.

The associative map (AFM) supported by the frame map abstraction is
defined on the 'am index_type'. Each AFM entry associates a set of four
attributes — a page name, a frame name, a 'mf' flag and an 'arf' flag
(associative entry reference flag). The last flag attribute is a measure
of the frequency of reference to the AFM entry and is used internal to
the FM abstraction in deciding the AFM entry to be replaced.

There are two associative map search operations, AGET_FR and ASET_FW.
They both search the AFM entries for an entry containing an association
with the given frame name. If successful, the 'arf' flag is set to 1 and
the frame name is returned. The difference between the two operations is
that if successful, the AGET_FW operation also sets the 'mf' flag in the
entry. The two ASET operations allocate an AFM entry and enter the new association. Internal to the frame map abstraction, an existing AFM entry may have to be replaced. The ANUL operation searches the AFM for an entry which contains the given frame name to page name association. Such an entry is removed from the AFM. Whenever an AFM entry is replaced, either implicitly because of an ASET or explicitly because of an ANUL, the corresponding RFM entry has to be updated with the current 'mf' flag. Furthermore, the 'apf' flag is set to zero and the 'rf' flag is set to 1, in the updated RFM entry.

Appendix 2a.2 The External Mainstream Abstractions for VM2

The Secondary Storage Abstraction—

The secondary storage (SS) abstraction manages 'home' objects. The SS abstraction provides three operations defined on home objects. The FETCH and STORE operations require a home name and a frame name as input parameters. The FETCH operation transfers the contents of the home into the frame; the STORE operation transfers the contents of a frame into a home. Since both these operations also affect the respective frame objects, it seems a violation of the object based system's philosophy. However, the interaction could as well be modelled with the frame name parameter being replaced by the set of data elements constituting a frame. Hence, this is not a serious discrepancy. The SS abstraction also provides an HZERO operation with which all the contents of a home can be set to zero.

The Base Processor Abstraction (VP2)—

The base processor (VP2) abstraction provides the synchronizing operations for the paging abstraction. The VP2 abstraction manages the
processors on which the users of the VP2 abstraction execute. The processor used for the frame freer function is also managed by VP2. Therefore, all the VP2 operations are defined on these 'processor' objects, and, hence, require a 'processor name' as a parameter. At this stage, we will assume that this parameter is automatically passed whenever a VP2 operation is invoked. The 'processor' objects managed by VP2 are discussed in Chapter IV. The nature of the VP2 operations used by the paging abstraction are discussed in Section II.4e.
Appendix 2a.3

The Abstract Implementation

of the VM2 Operations
READ_VM2_PG (p : p_n", o : offs, elem) :-
  local x: f_n, h: h_n, r:elem

begin READ_VM2_PG
  x = AGET_PM_FR (p, f_n)
  if (x != null)
    then begin
      r = READ_VMO_FR (x, o, elem)
      return (r)
    end
  else if (GET_PMZF (p, zf) = 1)
    then return (0)
  else begin
    $BLOCK_PAGELOCK (p)
    if (GET_PM_CF (p, cf) = 1)
      then begin
        x = GET_PM_FN (p, f_n)
        ASET_PM.FR (p, x)
        r = READ_VMO_FR (x, o, elem)
      end
    else begin
      x = $FRAME_CLAIMER (f_n)
      SET_PM_FN (x, p)
      h = GET_PM_HN (p, h_n)
      FETCH_SS_HTF (h, x)
      SET_PM_FN (p, x)
      SET_PM_CF (p, l)
      ASET_PM.FR (p, x)
      r = READ_VMO_FR (x, o, elem)
    end
    $UNLOCK_PAGELOCK (p)
    return (r)
  end
end_READ_VM2_PG

Fig. A2.3a The READ_VM2_PG Operation
WRITE_VM2_PG (p : p_n", o : offs, e : elem) :-
local x : f_n, h : h_n
begin WRITE_VM2_PG
  x = AGET_FM_FW (p, f_n^+) /* try the Assoc. Map (AFM) */
  if (x != null) then
    WRITE_VMO_FR (x,0,e)
  else begin
    $BLOCK_PAGELOCK (p)
    if (GET_PM_CF(p,cf) = 1) /* if frame exists */
      then begin
        x = GET_PM_FN (p, f_n)
        ASET_FM_FW (p,x) /* add new Assoc. Map entry */
        WRITE_VMO_FR (x,o,e) /* write into frame */
      end
    else begin /* else page fault */
      x = $FRAME_CLAIMER (f_n)
      SET_FM_FN (x,p)
      if (GET_PM_ZF(p,zf) = 1) /* if page is zero */
        then begin /* then no sec. storage transfer*/
          SET_PM_ZF (p,1)
          FZERO_VMO_FR (x) /* ZERO the frame */
        end
      else begin
        h = GET_PM_HN (p,h_n)
        FETCH_SS_HTF (h,x) /* 'fetch' home to frame */
      end
      SET_FM_FN (p,x)
      SET_PM_CF (p,1)
      ASET_FM_FW (p,x)
      WRITE_VMO_FR (x,e,e)
    end
    $UNLOCK_PAGELOCK (p)
  end
end WRITE_VM2_PG

Fig. A2.3b The WRITE_VM2_PG Operation
$BLOCK_PAGELOCK (p : p_n) :-
local pl: p_lock, cg: cf
begin $BLOCK_PAGELOCK

pl, cg = TSTC_PM_PC (p, p_lock, cf)
while (pl = 1)
begin
while ((pl=1) & (cg=1)) /* while 'cf' is 1, busy */
pl, cg = TSTC_PM_PC (p, p_lock, cf) /* wait on p_lock */
while ((pl=1) & (cg=0)) /* while 'cf' is 0, */
begin
BLOCK_VP2_IQ (PQ, p) /* block in indexed Q */
pl, cg = TSTC_PM_PC (p, p_lock, cf) /* test and set again*/
end
end
SETS_VP2_IQ (PQ, p) /* set indexed Q semaphore */
end $BLOCK_PAGELOCK

Fig. A2.3c' The $BLOCK_PAGELOCK Function

$UNLOCK_PAGELOCK (p : p_n):-
begin $UNLOCK_PAGELOCK
UNLOCK_PM_PL (p)
UNBLOCK_VP2_IQ (PQ, p) /* Unblock any blocked procs. */
end $UNLOCK_PAGELOCK

Fig. A2.3c'' The $UNLOCK_PAGELOCK Function

$FRAMECLAIMER (f_n) :-
local f: f_n
begin $FRAMECLAIMER

ENTER_VP2_MSEM (FFL) /* Enter the FFL monitor */
f = ALLOCATE_PM_FF (f_n) /* Allocate a free frame */
if ( REM_PM_FF(f_rem) = 'thresh' - 1) /* if threshold */
then SIGEXIT_VP2_CQ(FFL,FFQ) /* then signal Frame Freer */
else EXIT_VP2_MSEM (FFL) /* else simple exit */
end $FRAMECLAIMER

Fig. A2.3c''' The $FRAMECLAIMER Function
$\text{FRAME\_FREER}()$ :-
local $f, f_n, p, p_n, pl, p\_lock, h, h_n$

begin $\text{FRAME\_FREER}$

$\text{WAIT\_VP2\_CQ(FFQ)}$  /* Initially,'wait' in FFQ */

while $(\text{REM\_FM\_FR}(f, f_n) < \text{thresh'})$

begin

while $(0 < \text{REM\_FM\_FR}(f, f_n) < \text{thresh'}) \& \& \text{empty}_2\_\text{MSEM}(\text{FFL}, ef) = 0)$  /* & FFL Q not empty */

$\text{EXCHANGE\_VP2\_MSEM(FFL)}$  /* exchange with procs. at FFL head*/

if $(\text{REM\_FM\_FR}(f, f_n) > \text{thresh'})$

then break (1)  /* quit loop */

$f = \text{FRAME\_REPLACE}(f, f_n)$  /* choose frame to be freed */

$p = \text{GET\_FM\_PN}(f, p_n)$  /* get associated page */

/* The Frame Freer has to ensure that the frame chosen for free*/
/* has not already been freed because of a PZERO operation.*/
/* Once the page is locked, its frame cannot be freed elsewhere*/

$pl = \text{TST\_PM\_PL}(p, p\_lock)$  /* test and set the p\_lock */

while $(pl=1) \& \& (\text{GET\_FM\_PN}(f, p_n)=null) \& \& (\text{GET\_FM\_RF}(f, rf)=1)$

begin

if $(pl=0)$ then

$\text{UNLOCK\_PAGELOCK}(p)$

if $(\text{REM\_FM\_FR}(f, f_n) > \text{thresh'})$  /* if $f, f_n >$ threshold */

then break (2)  /* then quit both loops */

$f = \text{FRAME\_REPLACE}(f, f_n)$

$p = \text{GET\_FM\_PN}(f, p_n)$

$pl = \text{TST\_PM\_PL}(p, p\_lock)$
end

if $(\text{GET\_FM\_MF}(f, mf) = 1)$  /* if frame has been modified */

then begin

$h = \text{GET\_FM\_HN}(p, h_n)$

$\text{STORE\_SS\_FTH}(h, f)$  /* 'store' frame into home */

end

$\text{SET\_PM\_CF}(p, 0)$  /* reset 'cf' to zero */

$\text{UNLOCK\_PAGELOCK}(p)$

$\text{FREE\_FM\_FR}(f)$  /* free the frame */
end

end $\text{FRAME\_FREER}$

Fig. A2.3d The $\text{FRAME\_FREER}$ Function
$\text{FRAME\ REPLACE\ (f\ n)}$ :- 

local\ f: f n

begin $\text{FRAME\ REPLACE}$

\[ f = \text{GET\_FM\_LF\ (f\ n)} \]
\[ f = \text{SUCC\_FM\_FN\ (f, f\ n)} \]

/* get the 'last frame freed' ref. */
/* get the next frame which can be */
/* considered for replacement */

while (GETR\_FM\_RF(f, rf) = 1) /* while reference flag is 1 */

\[ f = \text{SUCC\_FM\_FN\ (f, f\ n)} \]

/* try another frame */

/* 'f' now identifies the chosen frame */

SET\_FM\_LF (f) /* set the ;last frame freed' ref. */

return (f)

end $\text{FRAME\ REPLACE}$

Fig. A2.3e' The $\text{FRAME\ REPLACE}$ Function

$\text{PZERO\_VM2\_PG\ (p : p\ n')}$ :- 

local\ f: f n

begin $\text{PZERO\_VM2\_PG}$

\[ \text{if (GET\_FM\_ZF(p, zf) = 0) } \]
/* Only if page is not zero */

then begin

$\text{BLOCK\_PAGELOCK\ (p)}$

SET\_FM\_ZF (p, 1) /* Set 'zero flag' to 1 */

if (GET\_FM\_CF (p, cf) = 1) /* if frame exists */

then begin

\[ f = \text{GET\_FM\_FN\ (p, f\ n)} \]

/* get the frame name */

ANUL\_FM\_AE (p, f) /* remove any Assoc. Map entry*/

SET\_FM\_CF (p, 0) /* reset 'core flag' to 0 */

FREE\_FM\_FR (f) /* free the frame */

end

$\text{UNLOCK\_PAGELOCK\ (p)}$

end

end $\text{PZERO\_VM2\_PG}$

Fig. A2.3e' The $\text{PZERO\_VM2\_PG}$ Operation
PUPDATE_VM2_PG \( (\text{p} : \text{p}_n^\prime\prime, \text{zf}) \) :-

\text{local zg: zf, f: f}_n, h: h_n

begin PUPDATE_VM2_PG

$BLOCK_PAGELOCK (\text{p})

\text{h = GET_PM_HN (p, h}_n)

\text{zg = GET_PM_ZF (p, zf)}

\text{if (zg=1) then} \quad /* if page is zero */

\text{HZERO_SS_HM (h)} \quad /* set the home to zero */

\text{else if (GET_PM_CF(p, cf) = 1)} \quad /* else, if frame exists */

\text{then begin}

\text{f = GET_PM_FN (p, f}_n)

\text{ANUL_FM_AE (p, f)} \quad /* remove any Assoc. Map entry*/

\text{if (GET_FM_MF_f, mf) = 1) \quad /* if frame is modified */}

\text{then begin}

\text{STORE_SS_FM (h, f)} \quad /* 'store' frame into home */

\text{SET_FM_MF (f, 0)}

\text{end}

\text{end}

$UNLOCK_PAGELOCK (\text{p})

\text{return (zg)}

end PUPDATE_VM2_PG

---

**Fig. A2.3f** The PUPDATE_VM2_PG Operation

SEARCH_VM2_PG \( (\text{h : h}_n, \text{p : p}_n^+, p^+_n) \) :-

\text{return (SEARCH_PM_HN (h, p, p^+_n))}

GET_VM2_HN \( (\text{p : p}_n^\prime\prime, h_n) \) :-

\text{return (GET_PM_HN (p, h_n))}

QZERO_VM2_ZF \( (\text{p : p}_n^\prime\prime, \text{zf}) \) :-

\text{return (GET_PM_ZF (p, zf))}

---

**Fig. A2.3f** The SEARCH_VM2, GET_VM2 and QZERO_VM2 Operations
PSET_VM2_PH (p : p_n", h : h_n, zg : zf, p_n^+) :-
  local x: p_n, y: h_n, f: f_n

begin PSET_VM2_PH
  $BLOCK_PAGELOCK (p)
  y = GET_PM_HN (p, h_n)    /* get the old home name */
  x = SETSEARCH_PM_HN (h,p,p_n^+)    /* searches page to home assoc*/
  if (x = null)    /* if no home assoc. with page*/
    then begin
      if (GET_PM_ZF(p,zf) = 1) then    /* if page is zero */
        HZERO_SS_HM (y)     /* zero the home */
      else if (GET_PM_CF(p,cf) = 1) /* if frame exists */
        then begin
          f = GET_PM_FN (p,f_n)
          if (GET_PM MF (f,mf) = 1) then
            STORE_SS_FTH (y,f)
        end
      /* The old home is now consistent with the page contents */

  SET_PM_ZF (p, zg)
  if (GET_PM_CF(p, cf) = 1) /* if frame exists */
    then if (zg = 1) /* and new page contents zero */
      then begin
        f = GET_PM_FN (p,f_n)
        ANUL_FM_AE (p,f)
        SET_PM_CF (p,0)
        FREE_FM_FR (f)
      end
    else begin    /* else need to update frame */
      f = GET_PM_FN (p,f_n)
      ANUL_FM_AE (p,f)
      FETCH_SS_HTF (h,f)
      ASET_FM_FR (p,f)    /* enter new Assoc. Map entry */
    end
  end

$UNLOCK_PAGELOCK (p)
return (x)
end PSET_VM2_PH

Fig. A2.3g The PSET_VM2_PH Operation
APPENDIX 2b

Appendix 2b.1 The Passive Space Manager (PSM) Abstractions

Passive Space Map (PSM)

The passive space map (PSM) abstraction (Fig. A2.4) is a map which maintains a set of attributes for every existing 'address space name'. Three of these attributes are the parameters associated with any address space, -- len_max, plim_max and len_curr. Every address space has a PSM object associated with it, and the name of this object is also maintained as an attribute. A fourth parameter, called 'len_home', is not visible to the users of the address space abstraction. It identifies the length up to the last block in the address space, which has a home associated with it. All blocks beyond this length, do not have an associated home. The final two attributes in a PSM entry are a 'tf' flag and a 'ps_lock'. The 'tf' flag indicates if the address space is active (tf = 1) or not.

The standard GET and SET operations are defined to access the attributes maintained in a PSM entry. Address space names are not reusable as address space objects are of the create/destroy kind. A pair of operations, CREATE and DESTROY, define the lifetime of an address space name. A TSTES operation not only provides a 'test and set' function on the ps_lock but first checks if the address space name still exists. The 'ef' flag (existence flag) returned is set to 1, if the address space exists. An UNLOCK operation resets the ps_lock to zero.

Passive Block Map Manager (PBM)

The PBM abstraction (Fig. A2.5) manages PBM (passive block map) objects. A PBM object (identified by a pbm name) is a map which maintains a set of attributes for the 'block name_type'. The extent of the domain de-
PSM abstraction :-

attribute interpretation :-

\[ \text{len}_\text{max} : \text{integer } \leq \text{max. address space length} \]
\[ \text{plim}_\text{max} : \text{integer } \leq \text{max. page limit} \]
\[ \text{len}_\text{curr} : \text{integer } \leq \text{len}_\text{max} \]
\[ \text{len}_\text{home} : \text{integer } \leq \text{len}_\text{curr} \]
\[ \text{tf} : \text{address space active flag} \]

representation :-

PSM :-

\[ \text{as}_n \rightarrow \text{pbn}_n, \text{len}_\text{max}, \text{plim}_\text{max}, \text{len}_\text{curr}, \text{tf}, \text{len}_\text{home}, \text{ps}_\text{lock} \]

operations :-

CREATE_PSM_PSE (as_n)
DESTROY_PSM_PSE (as_n)

GET_PSM_PBN (as_n, pbn_n)
GET_PSM_JM (as_n, len_max)
GET_PSM_JLM (as_n, plim_max)
GET_PSM_LC (as_n, len_curr)
GET_PSM_LH (as_n, len_home)
GET_PSM_TF (as_n, tf)

SET_PSM_PBN (as_n, pbn_n)
SET_PSM_JM (as_n, len_max)
SET_PSM_JLM (as_n, plim_max)
SET_PSM_LC (as_n, len_curr)
SET_PSM_LH (as_n, len_home)
SET_PSM_TF (as_n, tf)

TEST_PSM_PSE (as_n, ef, ps_lock)
UNLOCK_PSM_PSL (as_n)

Fig. A2.4 Specification of the Passive Space Map (PSM)
PBR abstraction :-
The PBR abstraction is a manager of PBM (passive block map) objects. Each PBM object is a map defined as:

PBM object :-

flag interpretation :-

hf : home flag  /*if 1, implies associated home exists */
df : destroyed flag /*if 1, implies block is destroyed */
pbf : pbm fault  /*set to 1, if PBM's size cannot be changed

representation :-

PBM :-

map0_e : b_n → df=0, hf, h_n^+
mapd_e : b_n → df=1, 0, null

→ len_max

Operations (defined by PBR) :-

CREATE_PBR_PBM (len_max, pbm_n)
DESTROY_PBR_PBM (pbm_n'' )
REALLOCATE_PBR_PBM (pbm_n'', len_max, pbf)
GET_PBR_PBE (pbm_n", (df, hf, h_n^+ ) )
SET_PBR_PBE (pbm_n", (df, hf, h_n^+) )

Fig. A2.5 Specification of the Passive Block map Manager (PBR)
defined by the block name_type is a parameter of the PEM object. This domain is decided by the 'len_max' parameter value. A map entry for a PEM object can be either of two forms, corresponding to an existing block (df = 0) or to a destroyed block (df = 1). The first form of map entry maintains the 'home name' associated with a block and a 'hf' flag (home flag) which, if set to 1, indicates that a home is associated with the block. The second form of map entry maintains the same two attributes but with the home name attribute always set to null and the 'hf' flag to zero.

PEM objects are of the create/destroy kind and hence their lifetime is defined by the CREATE and DESTROY operations. The size of a PEM object is decided by the 'len_max' value. The size can be changed by the REALLOCATE operation. This operation returns a 'pbf' flag (passive block map fault) which, if set to 1, indicates that the reallocation is not possible. The GET and SET operations are defined so as to access all the attributes of a map entry, in one operation.

Appendix 2b.2 The Active Space Manager (ASM) Abstractions

Active Space Map (ASM)

The active space map (ASM) abstraction (Fig. A2.6) maintains a set of attributes for each 'active address space' name (aas name). Every active address space has to have an aas name allocated to it. (Note, that although this association is unique, the two names are different). The domain of the aas name_type is variable and is not enforced by the ASM abstraction. The use of an ALLOCATE operation adds an aas name to the domain of the ASM, while a FREE operation has the opposite effect.

The ASM maintains the following attributes for each aas name in its current domain:
ASM abstraction :-

attribute interpretation :-
asrf : address space reference flag
plim.curr : integer = plim_max

representation :-
ASM :-
  aas.n → as.n, plim.curr, len.curr, asrf, as_lock
  LAS: ← asme.n

operations :-

ALLOCATE_ASM_ASE (aas.n, as.n)
FREE_ASM_ASE (aas.n)
GET_ASM_PC (aas.n, plim.curr)
GET_ASM_LC (aas.n, len.curr)
SET_ASM_PC (aas.n, plim.curr)
SET_ASM_LC (aas.n, len.curr)
SEARCH_ASM_AAN (as.n, aas.n+)
TSTG_ASM_AAL (as.n, aas.n+, as.lock)
TSTV_ASM_AAL (aas.n, as.n, aas.n+, as.lock)
UNLOCK_ASM_ASL (aas.n)
SEARCH_ASM_ASRF (as.n, aas.n+, asrf, len.curr)
GETR_ASM_ASRF (aas.n, asrf)
SUCC_ASM ASN (asme.n, as.n, asrf, asme.n, len.curr)
GETR_ASM_LAS (asme.n)
SETR_ASM_LAS (asme.n)

Fig. A2.6 Specification of the Active Space Map (ASM)
1) An address space name, which identifies the address space to which the aas name has been allocated.

2) The 'len_curr' parameter associated with the address space.

3) The 'plum_curr' parameter associated with an active address space.

4) An 'asrf' flag (address space reference flag) which is a measure of the frequency of reference to the address space.

5) An 'as_lock' which is used to lock the active address space.

The GET and SET operations are defined to access these attributes of an ASM entry. The SEARCH_AAN operation causes a scan of the ASM entries and returns the aas name which has been allocated to the given address space. A null value is returned if the search is not successful. This scan function is also provided in a 'test and set' operation, TSTG, which 'test and sets' the as_lock if the search is successful. An alternative kind of test and set operation, TSTV, verifies that the given aas name has been allocated to the given address space. If the verification is true, the as_lock is tested and set. If not, the same function as in TSTG is carried out. The UNLOCK operation resets the as_lock.

The SEARCH_ASRF operation also scans the ASM entries for an aas name allocated to the given address space. If successful, the operation returns the 'asrf' flag value. The SETR_ASRF operation also returns the 'asrf' flag attribute in an ASM entry. Both these operations reset the 'asrf' flag if it is 1. The SUCC operation requires an 'asme name' as an input parameter. This serves as the index of an ASM map entry and causes the operation to return the address space name and 'asrf' flag attributes of the next ASM entry. The asme name identifying this map entry is also returned. The ASM abstraction also maintains a map wide asme name attri-
bute called LAS. This can be used as a reference point in implementing a 'modified LRU' policy with respect to address space deactivation. The GETR_LAS and SETR_LAS operations are defined to access this attribute. The SEARCH_ASRF, GETR, SETR and SUCC operations are directly used in implementing the set_8 operations of the address space abstraction. The SEARCH_ASRF and SUCC operations also return the 'len_curr' attribute of an active address space, so that it can be used as a criterion for the deactivation policy.

**Active Block Map Manager (ABR)**

The ABR abstraction (Fig. A2.7) manages AEM (active block map) objects. The ABR abstraction serves another important purpose as it manages the associative map used by the address space abstraction. The associative map is defined on a system wide basis.

An AEM object is identified by an 'aas name', and is a map defined for the 'block name_type'. The domain for the block name_type is decided by a 'len_curr' parameter value. An AEM object has to be reallocated if its len_curr parameter has to be changed. A limit is imposed on the sum total of the 'len_curr' parameters of all existing AEM objects. Thus an AEM object is an allocate/free kind of object although the total number of existing AEM objects can vary.

The map entries in an AEM object can take any of three forms. Two of the forms correspond to the two forms which a PBM map entry can take. These two forms maintain the attributes for an inactive block. The third form maintains the attributes for an active block. This form is distinguished from the other two forms by the 'bf' flag (block flag) being set to 1. Apart from the block flag, the map entry for an active block main-
ABR abstraction :-

1) The ABR abstraction is a manager of ABM (active block map) objects. Each ABM object is a map defined as:

ABM object :-

flag interpretation :-

bf : block active flag /* set to 1, if block is active */
lsf : local sharing flag /* set to 1, if active sharing blks*/
brf : block reference flag
amf : associative map presence flag
abf : abm fault flag /* implies ABM's size not changed */

representation :

mapd_e : b_n → bf=0, df=1,0,null
map0_e : b_n → bf=0, df=0, hf, h_n^+
map1_e : b_n → bf=1, p_n, p_ind, lsf, amf, brf, {(b_n)^+, BLL link} → len_curr
LB : → b_n

Operations (defined by ABR) :-

ALLOCATE_ABR_ABM (len_curr, aas_n^+)
FREE_ABR_ABM (aas_n")
REALLOCATE_ABR_ABM (aas_n", len_curr, abf)

GETO_ABR_ABE (aas_n", b_n, (df,hf,h_n^+) )
GETO_ABR_HF (aas_n", b_n, hf)
GETO_ABR_HN (aas_n", b_n, h_n^+)

SETO_ABR_ABE (aas_n", b_n, (df,hf,h_n^+) )
SETO_ABR_HF (aas_n", b_n, hf)
SETO_ABR_HN (aas_n", b_n, h_n^+)

DESTROY_ABR_BLK (aas_n", b_n)
RECREATE_ABR_BLK (aas_n", b_n)
QEXIST_ABR_DF (aas_n", b_n, df)

CHANGE_ABR_ITA (aas_n", b_n)
CHANGE_ABR_ATI (aas_n", b_n)

Fig. A2.7 Specification of the Active Block map Manager (ABR)
GET_ABR_HF (aas_n", b_n, \text{hf})
GET_ABR_BF (aas_n", b_n, \text{bf})

GET1_ABR_PN (aas_n", b_n, p_n)
GET1_ABR_PI (aas_n", b_n, p_ind)
GET1_ABR_LSF (aas_n", b_n, lsf)
GET1_ABR_BLL (aas_n", b_n, b_n^+)

SET1_ABR_PN (aas_n", b_n, p_n)
SET1_ABR_PI (aas_n", b_n, p_ind)
SET1_ABR_LSF (aas_n", b_n, lsf)
SET1_ABR_BLL (aas_n", b_n, b_n^+)

GETR_ABR_BRF (aas_n", b_n, bf, brf)
SUCC_ABR_ABK (aas_n", b_n, b_n, brf)
GETR_ABR_LB (aas_n", b_n)
SETR_ABR_LB (aas_n", b_n)

2) The ABR abstraction also maintains an associative map defined as:

\textbf{Associative Map (AM)} :-

\textbf{attribute interpretation} :-
ucount : integer $\leq$ max. number of VP2 processes
amrf : associative map reference flag

\textbf{representation} :-
i) AM :-
\[ am\_ind \rightarrow aas\_n, as\_n, b\_n, p\_n, ucount, amrf \]
\[ \text{LAE} \rightarrow am\_ind \]
ii) ANM :-
\[ amm\_ind \rightarrow aas\_n, b\_n, ucount \]

\textbf{operations (defined by ABR)} :-

AGET_ABR_AAP (as\_n, b\_n, aas\_n^+, p\_n^+)
ASET_ABR_AAP (as\_n, b\_n, aas\_n, p\_n)
DECR_ABR_AUC (aas\_n, b\_n)
ANUL_ABR_AAP (aas\_n, b\_n, amf)

\textbf{Fig. A2.7 (contd.) Specification of the ABR}
tains the following attributes:

1) The 'page name' which identifies the page which is used by the
   active block.

2) The 'page index' which is allocated to the active block.

3) A 'l$ flag (local sharing flag) which if set to 1, implies
   that there is another active block in the same address space,
   which shares a home (and also the page and page index) with
   this block.

4) A 'block name' link to the next block in the BLL (block linked
   list). A BLL links all active blocks in the same address space
   which share a home.

5) A 'brf' flag (block reference flag) which is a measure of the
   frequency of access to the block.

6) An 'amf' flag (associative map presence flag) which, if set to
   1, implies that the block to page association is maintained in
   the associative map.

The lifetime of AEM objects are defined by a pair of operations,
ALLOCATE and FREE. The size of an AEM object is decided by the 'len_curr'
parameter value. The size can be changed by reallocating the object through
the REALLOCATE operation. This operation returns an 'abf' flag (active
block map fault) value of 1, if it is not possible to reallocate an AEM
object of a different size. An AEM object, when it is initially allo-
cated has all its block entries initialized to that corresponding to a
destroyed block. Once allocated, the entries have to be updated from
the block entries in the PBM object. The GET$ ABE and SET$ ABE opera-
tions can be used in conjunction with the similar operations defined on
PBM objects, to transfer the block map between a PBM and an AEM object.
GET∅ and SET∅ operations are also defined to access single attributes maintained in a block entry for an inactive block.

The DESTROY and RECREATE operations are used to change between an entry form corresponding to a destroyed block and that for an existing inactive block. The QEXIST operation returns a 'df' flag value of 1, if the block entry corresponds to a destroyed block. The two CHANGE operations provide for changing between entries of the forms corresponding to an existing inactive block and an active block. The GET_BF operation returns the 'bf' flag value of 1 if the block is active, zero otherwise.

The attributes for an active block entry can be accessed by the GET1 and SET1 operations. The GETR_BRF operation returns the inclusive OR of the 'brf' and 'amf' flags. Therefore, as long as the block to page association is maintained in the associative map, this operation returns a 'brf' value of 1. When a block to page association is removed from the associative map, the 'brf' attribute and 'amf' attribute in the corresponding block entry are set to 1 and ∅, respectively. A GETR_BRF in this state returns a 'brf' value of 1 but resets the 'brf' flag in the block entry to zero. The SUCC operation is defined to return the next active block name. Each AEM object maintains a map wide 'block name' attribute called LB. This serves as a reference point in implementing the 'modified LRU' policy with respect to block deactivation.

The ABR abstraction is also responsible for the system wide associative map (AM). The associative map is defined for an 'associative index type'. Each entry of this map associates together an address space name, an ass name, a block name and a page name. Each AM entry has two additional attributes, a 'ucount' (use count) integer value, and an 'amrf' flag (associative map reference flag). The last flag is used internally
in deciding which AM entry to replace. The associative map is searched with the AGET operation. If the associative search is successful, this operation returns a non-null aas name and a non-null page name and sets the 'amrf' flag to 1. If the search is partially successful, a non-null aas name is returned. If the search is not successful, both null names are returned. Each successful associative memory search operation increments (by 1) the 'ucount' attribute of the AM entry.

The 'ucount' is used to delay the successful completion of an ANUL operation. The ANUL operation searches the AM entries for an entry containing the given 'aas name, block name' association. If an AM entry is found, the entry is removed. If the 'ucount' in the removed AM entry was zero, then the ANUL operation returns with the 'amf' flag value set to 0, indicating a successful completion. However, if the 'ucount' in the AM entry is non-zero, then the 'aas name, block name' association and the corresponding 'ucount' are transferred to another associative map called ANM. The ANUL operation returns with a 1 value in this case. It is also possible for the ANUL operation to find no AM entry containing the 'aas name, block name' association, but an ANM which contains it. This occurs if the associative entry had been explicitly or implicitly replaced a little earlier. In this case also, the ANUL operation returns with a flag value of 1.

The 'ucount' value is decremented (by 1) by the DECR operation. This operation decrements the ucount whether it is being maintained in an AM or ANM entry. In the latter case, the ANM entry is removed if the ucount becomes zero.

The ASET operation enters a new association into the associative map. This may require that an existing AM entry be replaced, so as to make
space for the new entry. This implicit AM entry replacement can be based on a local policy which uses the 'amrf' flag and the 'ucount' attribute. The entry with the least ucount is preferred for replacement. If the chosen entry has a non-zero ucount, then the ucount value and the 'aas name, block name' association has to be transferred to the ANM*, where it can be maintained until the ucount becomes zero.

Page Index Map Manager (PIR)

Every active address space has to be allocated a PIM (page index map) object, in addition to the ABM object. The PIR abstraction (Fig. A2.8) manages PIM objects which are identified using the 'aas name'. Since each active address space is uniquely allocated an aas name, an ASM entry, an ABM object and a PIM object, we can use the aas name as the common identifier. The aas name identifying a PIM object can actually be regarded as a 'pim name'.

A PIM object is a map defined for the 'page index_type' and maintains a set of attributes for each page index. The domain of the page index_type is decided by the 'plim_curr' parameter value. This is a parameter associated with each active address space. A PIM object is also a limited resource, but its lifetime is tied to that of the corresponding ABM object. The implication is that if an 'aas name' is allocated by allocating an ABM object, then a corresponding PIM object can also be allocated. This is quite a reasonable assumption since the sum of the sizes

* The size of the ANM need be only enough to accommodate N entries, where N is the maximum number of base processors. This is obvious since the sum of all the ucount attributes corresponds to the number of users which are executing a VM2 access operation after a successful AM search.
PIR abstraction :-

The PIR abstraction is a manager of PIM (page index map) objects. Each PIM object is uniquely identified by an 'aas name'. A PIM object is defined as

PIM object :-

flag interpretation :-

asf : across address space sharing flag
pif : page index free flag

representation :-

mapx_e : p_ind ← pif=0, {(p_ind)+ ILL link}
mapy_e : p_ind ← pif=1, p_n, lsf, {(b_n) BLL header},
         asf, {(aas_n,p_ind)+ ALL link}
         → {(p_ind)+ ILL header}
         → plim_rem : integer ≤ max #(p_ind)

Operations (defined by PIR) :-

ALLOCATE_PIR_PIM (aas_n, plim_curr)
FREE_PIR_PIM (aas_n")
CHANGESIZE_PIR_PIM (aas_n", plim_curr)

ALLOCATE_PIR_PI (aas_n", p_ind+)
FREE_PIR_PI (aas_n", p_ind)
REM_PIR_NI (aas_n", plim_rem)

SEARCH_PIR_PI (aas_n, p_n, p_ind+)

GET_PIR_PN (aas_n", p_ind, p_n)
GET_PIR_LSF (aas_n", p_ind, lsf)
GET_PIR.BLL (aas_n", p_ind+, b_n+)
GET_PIR.ASF (aas_n", p_ind, asf)
GET_PIR.ALL (aas_n", p_ind, (aas_n, p_ind)+)

SET operations corresponding to the above

Fig. A2.8 Specification of the Page Index map Manager (PIR)
('plim_curr' parameters) of all PIM objects can be related* to the total number of pages available to the address space abstraction. Hence, the ALLOCATE and CHANGESIZE operations are always successful.

Page index entries are of two forms, the first corresponding to a free page index \((pif = 0)\) and the second to an allocated page index \((pif = 1)\). The entry for a free page index has only one attribute which is used as a link in a linked list of free page indexes (the PLL list).

The attributes maintained for an allocated page index entry are:

1) A 'page name' identifying the page which is associated with an allocated page index.

2) A 'lsf' flag (local sharing flag) which is set to 1 if the page index is being shared by multiple active blocks in the address space.

3) A 'block name' header for the BLL (block linked list). The BLL links all active blocks in the address space, sharing this page index.

4) An 'asf' (across address space sharing flag) flag which is set to 1 if there are active blocks in other address spaces which share the same page (and home).

5) An 'aas name, page index' pair which serves as a link in an ALL (across linked list). Since a 'aas name, page index' uniquely identifies a page index, the ALL links all page indexes (in different address spaces), which have been allocated to the

* The sum of the sizes of all PIM objects will be greater than the total number of pages, since active sharing blocks in different address spaces, share the same page but have different page indexes. An estimate can be made of the percentage of such sharing blocks. This establishes the relation between the total number of page indexes (equal to sum of the sizes of all PIM objects) and the total number of pages.
active blocks sharing the same page.

The ALLOCATE_PIM, FREE_PIM operations define the lifetime of a PIM object. The CHANGESIZE operation changes the size of a PIM object. Page indexes are themselves of the allocate/free kind and have the pair ALLOCATE_PI, FREE_PI defined for them. The REM operation returns the number of free page indexes in a PIM object. The SEARCH operation searches all the page index entries in a PIM object for a page index which has the given page name as an attribute. If such a page index is found, then it is returned, else a null value is returned. GET and SET operations are defined to access the attributes maintained for an allocated page index.

Appendix 2b.3 The Page Name Map (PNM)

The page name map (PNM) is defined for the 'page name_type' (Fig. A2.9). The domain is the set of page names which identify the pages used by the address space abstraction, in extending from the partitioned paged memory abstraction. Page entries are of two forms, the first corresponding to a free page (pff = 0), the second corresponding to an allocated page (pff = 1). The first form contains a page name as its only attribute. This page name serves as a link in a linked list of free pages (PLL). The second form of page entry corresponds to an allocated page and has three attributes:

1) An 'asf' flag which is set to 1 if there is more than one page index associated with the page.
2) A 'as_name, page index' pair which serves as the header to a linked list of page indexes (ALL) associated with the page.
3) A 'pg_lock' which is used to lock the page.

There is another attribute of a page name, the home name, which is
PNM abstraction :-

flag interpretation :-

pff : page free flag

representation :-

\[ \text{mapf}_e : p_n \rightarrow \text{pff}=0, \{(p_n)\}^+ \text{ PLL link}\]

\[ \text{mapa}_e : p_n \rightarrow \text{pff}=1, \text{asf}, \{(aas_n,p\_ind)\} \text{ ALL header}, \text{pg\_lock} \rightarrow \{(p_n)\}^+ \text{ PLL header} \rightarrow \text{page\_rem} : \text{integer} \leq \text{max \#(pages)} \]

operations :-

\[ \text{ALLOCATE\_PNM\_PG} (p_n^+) \]

\[ \text{FREE\_PNM\_PG} (p_n) \]

\[ \text{REM\_PNM\_NP} (\text{page\_rem}) \]

\[ \text{GET\_PNM\_ASF} (p_n, \text{asf}) \]

\[ \text{GET\_PNM\_ALL} (p_n, (aas_n,p\_ind)^+) \]

\[ \text{SET\_PNM\_ASF} (p_n, \text{asf}) \]

\[ \text{SET\_PNM\_ALL} (P_n, (aas_n, p\_ind)) \]

\[ \text{TST\_PNM\_PGL} (p_n, \text{pg\_lock}) \]

\[ \text{UNLOCK\_PNM\_PGL} (p_n) \]

Fig. A2.9 Specification of the Page Name Map (PNM)
maintained by the paging abstraction and can be accessed by the GET_VM2_HN operation. A 'free page' is recognized only at the level of the address space abstraction (VM4). Within the paging abstraction, all pages are considered to be allocated and are associated with a home. Thus, although a page is free at the level of the VM4 abstraction, it is still associated with its last home at the level of the VM2 abstraction.

Pages are allocate/free kind of objects and their lifetime, with respect to the address space abstraction, is defined by the ALLOCATE and FREE operations. The page name map abstraction defines the standard GET and SET operations to access the attributes of an allocated page entry. A 'test and set' operation, TST, is defined to lock the pg_lock and an UNLOCK operation is defined to reset the pg_lock to zero.
Appendix 2b.4

The Abstract Implementation

of the VM4 Operations
READ VM4 ASB (a; as_n, b: b_n, o: offs, elem) :fault(tf',df,hf',bf)

local x:aas_n, p:p_n, r:elem, al:as_lock, bg:bf, ft:fault

begin READ VM4 ASB

x,p = AGET_ABR_AAP(a,b,aas_n+,p_n+) /* try Assoc. map */
if (p != null) then /* if full success */
begin

r = READ_VM2_PG(p,o,elem) /* 'read' page */
DECR_ABR_AUC(x,b) /* decrement 'ucount' */
return (r)
end
else if (x = null)

then x,al = TSTG_ASM_ASL(a,aas_n+,as_lock)
else x,al = TSTV_ASM_ASL(x,a,aas_n+,as_lock)
while ((x != null) & (al = 1)) /* busy wait on as_lock */
x,al = TSTV_ASM_ASL(x,a,aas_n+,as_lock)
if (x = null) then return (fault(1)) /* 'tf' fault */
bg = GET_ABR_BF(x,b,bf)
if (bg = 0) then /* if block is inactive */
begin

if (QEXIST_ABR_DF(x,b,df) = 1) /* if block destroyed */
then ft = fault(2) /* 'df' fault */
else if (GET0_ABR_HF(x,b,hf) = 0) /* if no associated home*/
then ft = fault(3) /* 'hf' fault */
else ft = fault(4) /* 'bf' fault */
UNLOCK_ASM_ASL(x) /* unlock 'as_lock' */
return (ft)
end
else begin

p = GET_ABR_PN(x,b,p_n) /* get the page name */
ASET_ABR_AAP(a,b,x,p) /* add the new AM entry */
UNLOCK_ASM_ASL(x) /* unlock as_lock */
r = READ_VM2_PG(p,o,elem) /* 'read' the page */
DECR_ABR_AUC(x,b) /* decrement the 'ucount' */
return (r)
end
end READ VM4 ASB

Fig. A2.10a The READ VM4 ASB Operation
WRITE_VM4_ASB :- Same as READ_VM4_ASB with the invocations of

READ_VM2_PG replaced by WRITE_VM2_PG

BZERO_VM4_ASB :- Same as READ_VM4_ASB with the invocations of

READ_VM2_PG replaced by PZERO_VM2_PG

Fig. A2.10b' The WRITE_VM4_ASB and BZERO_VM4_ASB Operations

$LOCK_PSL (a: as_n, ef) :-

local eg:ef, pl:ps_lock

begin $LOCK_PSL

eg,pl = TSTE_PSM_PSL (a, ef, ps_lock)

while ( (eg=1) & (pl=1) )

eg,pl = TSTL_PSM_PSL (a, ef, ps_lock)

return (eg)

end $LOCK_PSL

Fig. A2.10b'' The $LOCK_PSL Function

$LOCK_ASL (a: as_n, aas_n^+) :-

local x:aas_n, al:as_lock

begin $LOCK_ASL

x, al = TSTC_ASM_AAL (a, aas_n^+, as_lock)

while ( (x != null) & (al = 1) )

x, al = TSTV_ASM_AAL (x, a, aas_n^+, as_lock)

return (x)

end $LOCK_ASL

Fig. A2.10b''' The $LOCK_ASL Function
CREATE_VM4_AS (lm: len_max, lc: len_curr, pm: plim_max, as_n): fault(f4)

local a: as_n, pb: pbm_n

begin CREATE_VM4_AS

if (lc > lm) then /* if len_curr > len_max */
  return (fault(1)) /* 'f4' fault */
a = CREATE_PSM_PSE (as_n) /* create new PSM entry */
pb = CREATE_PSR_PBM (lc, pbm_n) /* create new PBM object */
SET_PSM_PBN (a, pb)
SET_PSM_LM (a, lm)
SET_PSM_LC (a, lc)
SET_PSM_PLM (a, pm)
UNLOCK_PSM_PSL (a)
return (a)
end CREATE_VM4_AS

Fig. A2.10c' The CREATE_VM4_AS operation

DESTROY_VM4_AS (a: as_n): fault(ef', tf)

local eg: ef, pb: pbm_n, lc: len_curr, i: b_n, h: h_n

begin DESTROY_VM4_AS

eg = $LOCK_PSL (a, ef)
if (eg = 0) then /* if AS does not exist */
  return (fault(1)) /* 'ef' fault */
if (GET_PSM_TF(a, tf) = 1) then /* if AS is active */
  return (fault(2)) /* 'tf' fault */
pb = GET_PSM_PBN (a, pbm_n)
lc = GET_PSM_LC (a, len_curr)
for (i = b_n+1 to b_n+lc by b_n+1) /* for all blocks */
begin /* in the address space, */
  h = GET_PBR_HN (pb, i, h_n) /* free any associated homes */
  if (h != null) then
    FREE_HNR_HM (h)
end
DESTROY_PBR_PBM (pb)
DESTROY_PSM_PSE (a)
end DESTROY_VM4_AS

Fig. A2.10c" The DESTROY_VM4_AS Operation
SETLMAX_VM4_LM (a: as_n, lm: len_max) : fault(ef', f4, pbf)

local eg: ef, ft: fault, pb: pbm_n, pg: pbf

begin SETLMAX_VM4_LM

  ft = null
  eg = $LOCK_PSL (a, ef)
  if (eg = 0) then
    return (fault(1))
  if (lm < GET_PSM_LC (a, len_curr)) then
    ft = fault(2)
  else begin
    pb = GET_PSM_PBN (a, pbm_n)
    pg = REALLOCATE_PBR_PBM (pb, lm, pbf) /* reallocate PBM object */
    if (pg = 1)
      then ft = fault(3) /* 'pbf' fault */
    else SET_PSM_LM (a, lm) /* enter new len_max */
  end
  UNLOCK_PSM_PSL (a)
  return (ft)
end SETLMAX_VM4_LM

Fig. A2.10d' The SETLMAX_VM4_LM Operation

SETPMAX_VM4_PLM (a: as_n, pm: plim_max) : fault(ef', f1)

local eg: ef, x: aas_n, ft: fault

begin SETPMAX_VM4_PLM

  ft = null
  eg = $LOCK_PSL (a, ef)
  x = SEARCH_ASM_AAN (a, aas_n+)
  if ((x != null) & (pm < GET_ASM_PC(x, plim_curr))
    then ft = fault(2)
  else SET_PSM_PLM (a, pm)
  UNLOCK_PSM_PSL (a)
  return (ft)
end SETPMAX_VM4_PLM

Fig. A2.10d" The SETPMAX_VM4_PLM Operation
ACTIVATE_VM4_AS (a: as_n , lc: len_curr⁺, pc:plim curr):

      fault (ef',tf,f1,f4,f3,abf)

local eg:ef,ft:fault,pb:pbm_n,lr:len_curr,i:b_n,x:aas_n,dg:df,hg:hf,h:n

begin ACTIVATE_VM4_AS

  ft = null
  eg = $LOCK_PSL (a,ef)  /* first lock the ps_lock  */
  if (eg = 0) then
    return (fault(1))     /* 'ef' fault       */
  else begin
    if (SEARCH_ASM_AAN(a,aas_n⁺) = null) then /* if AS active  */
      ft = fault(2)          /* 'tf' fault      */
    else if ( pc > GET_PSM_PLM(a,plim max))
      then ft = fault(3)     /* 'fl' fault      */
    else if ((lc != null) & (lc > GET_PSM_LM(a,len_max)))
      then ft = fault(4)     /* 'f4' fault      */
    else begin
      if (lc != null) then
        if (lc < GET_PSM_LH(a,len home))
          then ft = fault(5)
        else SET_PSM_LC (a,lc)  /* enter new len_curr   */
        else lc = GET_PSM_LC (a,len curr) /* else get old one */
          x = ALLOCATE_ABR_ABM (lc,aas_n⁺)  /*allocate ABR obj. */
          if (x = null) then        /* if not possible */
            ft = fault(6)          /* 'abf' fault      */
        else begin
          ALLOCATE_ASM_ASE (x,a)   /* add new ASM entry*/
          ALLOCATE_PIR_PIM (x,pc)
          SET_ASM_PC (x,pc)
          SET_ASM_LC (x,lc)
          SET_PSM_LF (a,1)
          for (i=b_n⁻¹ to b_n⁻¹lc by b_n⁻¹) /* transfer block */
            begin /* entries from PBM to ABM */
              dg,hf,h = GET_PBR_PBE(pb,i,(df,hf,h_n⁺))
            SET_ABR_ABE (x, i, (dg,hg,h))
          end
        end
    end
  end

Fig. A2.10 e The ACTIVATE_VM4_AS Operation (contd. on next page)
UNLOCK_ASM_ASL (x) /* unlock the as_lock */
end

end

end

UNLOCK_PSM_PSL (a)
return (ft)

end ACTIVATE_VM4_AS

Fig. A2.10e (contd.) The ACTIVATE_VM4_AS Operation

SETPLIM_VM4_AS (a: as_n, pc: plim_curr) : fault (ef',tf',fl'f2)
local ft:fault, eg:af, x:aas_n, pr:plim_curr
begin SETPLIM_VM4_AS
eg = $LOCK_PSL (a, ef)
if (eg = 0) then
    return (fault(1))

x = $LOCK_ASM (a, aas_n+)
if (x = null) then /* if AS is inactive */
    ft = fault(2) /* 'tf' fault */
else begin
    if (pc > GET_PSM_PLM(a, plim_max)) then
        ft = fault(3)
    else begin
        pr = GET_ASM_PC (x, plim_curr)
        if (pc < pr) then /* if new plim_curr is less */
            if (pr - pc > REM_PIR_NI(x, plim_rem))
                then ft = fault(4) /* 'f2' fault */

        if (ft = null) then /* if no fault */
            begin
                SET_ASM_PC (x, pc)
                CHANGESIZE_PIR_PLM (x) /* change PIM object's size */
            end
        end
    end

UNLOCK_ASM_ASL (x)
end

UNLOCK_PSM_PSL (a)
end

SETPLIM_VM4_AS

Fig. A2.10f The SETPLIM_VM4_AS Operation
DEACTIVATE VM4 AS (a: as n) : fault (ef',sf)

\[
\text{local } ft: \text{fault,x:aas_n,eg:ef,pb:pbm_n,lc:len_curr,lh:len_home,i:b_n,}
\]
\[
dg:df,hg:hf,h:h_n
\]

begin DEACTIVATE VM4 AS

\[
ft = \text{null}
\]
\[
x = \text{SEARCH ASM AAN (a, aas n)}^+
\]
\[
\text{if (x = null) then } \text{return ()} /* return if AS already inactive */
\]
\[
eg = \text{LOCK PSL (a, ef)}
\]
\[
\text{if (eg = 0) then}
\]
\[
\text{return (fault(1))} /* 'ef' fault */
\]
\[
x = \text{LOCK ASL (a,aas n)}^+
\]
\[
\text{if (x != null) then}
\]

begin

\[
pb = \text{GET PSM PBN (a,pbm n)}
\]
\[
lc = \text{GET PSM LC (a,len_curr)}
\]
\[
lh = 0
\]
\[
\text{for (i = b_n to b_nlc by b_nl)} /* for all blocks in AS */
\]
\[
\text{if (GET ABR BF(x,i,bf) = 1)} /* if block is active */
\]
\[
\text{then begin}
\]
\[
ft = \text{fault(2)} /* 'sf' fault */
\]
\[
\text{break (1)}
\]
\[
\text{end}
\]
\[
\text{else begin} /* else, transfer block entry*/
\]
\[
dg,hg,h = \text{GETO ABR ABE(x,i,(df,hf,h_n)}^+) /*from ABM */
\]
\[
\text{SET PBR PBE (pb, i, (dg,hg,h) )} /* to PBM */
\]
\[
\text{if (GETO ABR HF(x,i,hf) = 1)}
\]
\[
\text{then lh = index(i)}
\]
\[
\text{end}
\]
\[
\text{if (ft = null)} /* if no fault occurred */
\]
\[
\text{then begin}
\]
\[
\text{SET PSM TF (a,0)}
\]
\[
\text{SET PSM LH (a,lh)}
\]
\[
\text{FREE PIR PIM (x)} /* free the PIM object */
\]
\[
\text{FREE ABR ABM (x)} /* free the ABM object */
\]

Fig. A2.10g The DEACTIVATE VM4 AS Operation (contd. on next page)
FREE_ASM_ASE /* free the ASM entry */
end
UNLOCK_ASM_ASL (x)
end
UNLOCK_PSM_PSL (a)
return (ft)
end DEACTIVATE_VM4_AS

Fig. A2.10g (contd.) The DEACTIVATE_VM4_AS Operation

ASSIGND_VM4_ASB (a: as_n, b:b_n, h: h_n) : fault (tf',bf,df,hef')
local ft:fault, x:aas_n, hm:h_n, hg:hef

begin ASSIGND_VM4_ASB
  ft = null
  x = $LOCK_ASL (a, aas_n+)
  if (x = null) then
    return (fault(1))
  if (GET_ABR_BF(x,b,bf) = 1) then /* if block is active */
    ft = fault(2) /* 'bf' fault */
  else if (QEXIST_ABR_DF(x,b,df) = 1)
    then ft = fault(3) /* 'df' fault */
  else begin
    hm = GETO_ABR_HN (x,b,h_n)
    if ( (hm != null) & (hm != h) ) then /* if different */
      FREE_HNR_HM (hm) /* then free old home */
    if (hm != h) then
      begin
        hg = SHARE_HN_HM (h, hef) /* increment share count */
        if (hg = 1) then /* if new home exists */
          SETO_ABR_HN(x,b,h) /* update block entry */
        else ft = fault(4) /* else 'hef' fault */
      end
    end
    UNLOCK_ASM_ASL (x)
  return (ft)
end ASSIGND_VM4_ASB

Fig. A2.10h The ASSIGND_VM4_ASB Operation
SETLEN VM4 AS (a: as_n, lc: len_curr) : fault (ef',f4,f3,abf)
local ft:fault, eg:ef, x:aas_n, lr:len_curr, i:b_n, tg:tof

begin SETLEN_VM4_AS
  ft = NULL
  eg = $LOCK_PSL (a, ef)
  if (eg = 0) then
      return (fault(1))
  x = $LOCK_AS (a, aas_n)
  if (x = NULL) then
    if (lc > GET_PSM_LM(a, len_max)) then
      ft = fault(2)
    else if (lc < GET_PSM_LH(a, len_home)) then
      ft = fault(3) /* 'f3' fault */
    else
      SET_PSM_LC (a,lc)
    end
    /* else AS is active */
  else
    if (lc > GET_PSM_LM(a, len_max)) then
      ft = fault(2)
    else
      lr = GET_PSM_LC(a, len_curr)
      if (lc < lr) then /* if new len_curr is smaller */
        for (i = b_nlc+1 to b_nlr by b_n1)
          if (GET_ABR_IF(x,i,hi) = 1) then
            begin
            ft = fault(3) /* 'f3' fault */
            break(1)
            end
        end
        if (ft = NULL) then /* if no fault */
        begin
          tg = REALLOCATE_ABR_ABM(x,lc,abf) /* reallocate ABM */
          if (tg = 1) then
            ft = fault(4) /* 'abf' fault */
          else
            begin
            SET_PSM_LC (a,lc) /* update PSMs len_curr */
            SET_ABM_LC (x,lc) /* update ASMs len_curr */
            end
          end
        end
  end
end

Fig. A2.10i The SETLEN_VM4_AS Operation (contd. on next page)
end
end

UNLOCK_ASM_ASL (x)
end
UNLOCK_PSM_PSL (a)
return (ft)
end SETLEN_VM4_A5

Fig. A2.101 (contd.) The SETLEN_VM4_A5 Operation

ASSIGNW_VM4_A5B (a: as_n, b: b_n, h_n) : fault (tf',bf,df)

local ft:fault, x:aas_n, h:h_n, hm:h_n

begin ASSIGNW_VM4_A5B

ft = null
x = $LOCK_A5L (a, aas_n+)
if (x = null) then
  return (fault(1))
if (GET_ABR_BF(x,b,bf) = 1) then
  ft = fault(2)
else if (QEXIST_ABR_DF(x,b,df) = 1) then
  ft = fault(3)
else begin
  h = ALLOCATE_HNR_HM(h_n)
  if (GETO_ABR_HF(x,b,hf) = 1) then
    begin
      h = GETO_ABR_HN (x,b,h_n)
      FREE_HNR_HM (h)
    end
    SHARE_HNR_HM (h,hef) /* no need to check 'hef' */
    SETO_ABR_HN (x,b,h)
  end
  UNLOCK_ASM_ASL (x)
  if (ft = null) then /* if no fault */
    return (h)
  else return (ft) /* else return fault */
end ASSIGNW_VM4_A5B

Fig. A2.10j The ASSIGNW_VM4_A5B Operation
RELEASE VM4 ASB (a: as n, b: b n) : fault (tf',bf,df)
local ft:fault, x:aas n, h:h n
begin RELEASE VM4 ASB
  x = $LOCK ASL (a, aas n+)
  if (x = null) then
    return ( fault(1) )
  if (GET_ABR BF(x,b,bf) = 1) then
    ft = fault(2)
  else if (QEXIST ABR DF(x,b,df) = 1) then
    ft = fault(3)
  else begin
    h = GET ABR HN (x,b,h n+)
    if (h != null) then /* if home exists */
    begin
      SET ABR HN (x,b,null)
      SET ABR HF (x,b,0)
      FREE HNR EM (h)
    end
  end
UNLOCK ASM ASL (x)
end RELEASE VM4 ASB

Fig. A2.10k' The RELEASE VM4 ASB Operation

RECREATE VM4 ASB (a: as n, b: b n) : fault(tf',df)
local ft:fault, x:aas n
begin RECREATE VM4 ASB
  x = $LOCK ASL (a, aas n+)
  if (x = null) then
    return ( fault(1) )
  if (index(b) > GET ASM LC(x,len curr) ) then /* if block outside */
    ft = fault(2) /* len curr, then 'df' fault */
  else if ((GET ABR BF(x,b,bf)=0) & (GET ABR DF(x,b,df)=1) )
    then RECREATE ABR BLK(x,b) /* recreate block */
  UNLOCK ASM ASL (x)
  return (ft)
end RECREATE VM4 ASB

Fig. A2.10k' The RECREATE VM4 ASB Operation
\texttt{ACTIVATE\_VM4\_BK} (a: as\_n, b: b\_n) : fault (tf', df, hf', if, mof)
\texttt{local ft:fault, x:aas\_n, h:h\_n, p:p\_n, pi:p\_ind, zg:zf}

\begin{verbatim}
begin ACTIVATE\_VM4\_BK
  ft = null
  x = $LOCK\_ASL \(a, aas\_n\)
  if (x = null) then
    return (fault(1))
    if (QEXIST\_ABR\_DF(x,b,df) = 1) then
      ft = fault(2) /* 'df' fault */
    else if (GET\_ABR\_BF(x,b,bf) = 0) then
      if (GET\_ABR\_HF(x,b,hf) = 0) then
        ft = fault(3)
      else begin
        h = GET\_ABR\_HN \(x,b,h\_n\)
        p = SEARCH\_VM2\_PG \(h,null,p\_n\) /* search if page exists already */
        RPT: if (p != null) then /* if so */
          p = $LOCKVER\_PAGEHOME(p,h,p\_n) /* lock pg\_lock and verify */
          if (p != null) then /* if page exists, then 'lsb' */
            begin /* or 'asb' exists for block */
              pi = SEARCH\_PIR\_PI(x,p,p\_ind) /* check if 'lsb' exists */
              if (pi != null) then /* if 'lsb' exists */
                $ADDO\_BL\_LL(x,b,pi,p) /* add block to existing BL\_L */
                else begin /* else only an 'asb' exists */
                  pi = ALLOC\_ATE\_PIR\_PI(x,p,p\_ind) /* allocate a page index */
                  if (pi=null) then
                    ft = fault(4) /* 'if' fault */
                  else begin
                    $ADDO\_ALL(x,pi,p) /* add block to existing ALL */
                    $ADDN\_BL\_LL(x,b,pi,p) /* add block to new BL\_L */
                  end
                end
              end
            end
          end
e else begin /* else, no 'lsb' or 'asb' */
            if (p = null) then
              ft = fault(5) /* 'mof' fault */
end
\end{verbatim}

\textbf{Fig. A2.10m \textit{The ACTIVATE\_VM4\_BK Operation}} (contd. on next page)
else begin
  zg = GET_HNR_ZF(h, zf)  /* get 'zero flag' from HNR */
  pt = PSET_VM2_PH(p,h,zg, p\_n\^+\_n)  /* change page home assoc. */
if (pt != null) then /* if page already allocated */
begin /* free and start all over again */
  FREE_PNM_PG (p)  /* free page */
  FREE_PIR_PI (x,pi)  /* free page index */
p = pt
  goto RPT  /* tsk! tsk! */
end
else begin /* else, all is well */
$ADDN_ALL (x,pi,p)  /* add block to new ALL */
$ADDN_BLL (x,b,pi,p)  /* add block to new BLL */
end
UNLOCK_PNM_PGL
end
end
end

UNLOCK_ASM_AS (x)

return (ft)
end ACTIVATE_VM4_BK

Fig. A2.10m (contd.) The ACTIVATE_VM4_BK operation

$ADD_BLL (x: aas\_n, b: b\_n, pi: p\_ind, p: p\_n) :-
local nb: b\_n
begin $ADD_BLL

  CHANGE_ABR_ITA (x,b)  /* change block entry to active*/
  nb = GET_PIR_BLL (x,pi,b\_n\^+)  /* get BLL header */
  SET1_ABR_LSF (x,nb,l)  /* set 'lsf' flag */
  SET1_ABR_LSF(x,b,l)
  SET1_ABR_BLL (x,b,nb)  /* enter BLL link */
  SET1_ABR_PN (x,b,p)  /* enter page name */
  SET1_ABR_PI (x,b,pi)  /* enter page index */
  SET_PIR_LSF (x,pi,l)  /* set 'lslf' in page ind. entry */
  SET_PIR_BLL (x,pi,b)  /* set BLL header */
end $ADD_BLL

Fig. A2.10n The $ADD_BLL Function
DESTROY_VM4_ASB (a: as_n, b: b_n) : fault(tf')

local x:as_n, p:p_n, h:h_n

begin DESTROY_VM4_ASB

x = $LOCK_ASL (a, as_n+)

if (x = null) then
  return ( fault(1) )

if (GET_ABR_BF(x,b,bf) = 1) then /* if block is active */

begin
  p = $DEACT_BLOCK (a,x,b,p_n+) /* then deactivate block */
  if (p != null) then /* if necessary */
    FREE_PNM_PG /* free the page */
end

h = GETO_ABR_HN (x,b,h_n+)

if (h != null) then

begin
  SETO_ABR_HN (x,b,null)
  SETO_ABR_HF (x,b, 0 )
  FREE_ENR_HM (h)
end
DESTROY_ABR_BLK (x,b)
UNLOCK_ASM_ASL (x)
end DESTROY_VM4_ASB

Fig. A2.10o' The DESTROY_VM4_ASB Operation

$ADD0_ALL (x: aas_n, pi: p_ind, p: p_n) :-

local na:aas_n, np:p_ind

begin $ADD0_ALL

na, np = GET_PNM_ALL (p, (as_n,p_ind) ) /* get ALL header */
SET_PIR_ASF (na, np, l) /* set 'asf' flag */
SET_PIR_ALL (x, pi, (na,np)) /* set ALL lin- */
SET_PIR_PN (x, pi, p) /* enter page name */
SET_PIR_ASF (x, pi, l)
SET_PNM_ALL (p, (x,p1) ) /* set ALL header */
SET_PNM_ASF (p, l)
end $ADD0_ALL

Fig. A2.10o" The $ADD0_ALL Function
$\text{LOCKVER\_PAGEHOME}$ \(\text{pg: p\_n, h: h\_n, p\_n^+}\) :=

\texttt{local p:p\_n, pl:pg\_lock}

\texttt{begin}$\text{LOCKVER\_PAGEHOME}$

/* This function first locks the given page and checks if its page */
/* name is associated with the given home name. If not, it searches */
/* to find if any other page name is associated with the home name */
/* and repeats the actions. It terminates if the search fails to find*/
/* any page name associated with the home name, or if after locking a*/
/* page, the check on the page to home name association holds. */

\(pl = 1\)

\(p = \text{pg}\)

\texttt{while ( (p != null) & (pl = 1) )}

\texttt{begin}

\(pl = \text{TST\_PNM\_PGL} (p, pg\_lock) /\ast \text{test and set the pg\_lock} /\ast\)

\texttt{if (pl = 1) then}

\(p = \text{SEARCH\_VM2\_PG} (h, p, p\_n^+)\)

\texttt{else if (h != GET\_VM2\_PG(p, h\_n) ) then}

\texttt{begin}

\(\text{UNLOCK\_PNM\_PGL} (p)\)

\(p = \text{SEARCH\_VM2\_PG} (h, \text{null, p\_n^+})\)

\texttt{end}

\texttt{end}

\texttt{return (p)}

\texttt{end}$\text{LOCKVER\_PAGEHOME}$

\textbf{Fig. A2.10p'} The $\text{LOCKVER\_PAGEHOME}$ Function

$\text{ADDN\_BLL} (x: a\_n, b: b\_n, pi: p\_ind, p: p\_n) :=$

\texttt{begin}$\text{ADDN\_BLL}$

\texttt{CHANGE\_ABR\_ITA} (x, b)

\texttt{SET1\_ABR\_PI} (x, b, pi)

\texttt{SET1\_ABR\_PN} (x, b, p)

\texttt{SET\_PIR\_PLL} (x, pi, b)

\texttt{end}$\text{ADDN\_BLL}$

\textbf{Fig. A2.10p''} The $\text{ADDN\_ALL}$ Function
DEACTIVATE_VM4_BK (a: as_n, b:b_n) : fault(tf',df)

local ft:fault, x:aas_n, p:p_n

begin DEACTIVATE_VM4_BK

ft = null
x = $LOCK_ASL (a, as_n+)
if (x = null) then
    return (fault(1))
if (QEXIST_ABR_DF(x,b,df) = 1)
    ft = fault(2)
else begin
    p = $DEACT_BLOCK (a,x,b, p_n+)
    if (p != null) then
        FREE_PNM_PG (p)
end
UNLOCK_ASM_ASL (x)
return (ft)
end DEACTIVATE_VM4_BK

Fig. A2.10q' The DEACTIVATE_VM4_BK Operation

$ADDN_ALL (x: aas_n, pi: p_ind, p:p_n) :-

begin $ADDN_ALL
    SET_PIR_PN (x,pi,p)
    SET_PNM_ALL (p, (x,pi))
end

Fig. A2.10q" The $ADDN_ALL function

$NULL_AMENTRY (x: aas_n, b:b_n) :-

local ag: amf'

begin $NULL_AMENTRY

/* This function busy waits on ANUL, until an Assoc. Map entry is */
    ag = ANUL_ABR_AAP (x,b,amf)  /* successfully removed */
while (ag = 1)
    ag = _NULL_ABR_AAP (x,b,amf)
end $NULL_AMENTRY

Fig. A2.10q"' The $NULL_AMENTRY function
$DEACT\_BLOCK\ (a: as\_n, x: aas\_n, b: b\_n, p_n^+ ) :-
  local r:p_n, p:p_n, pl: pg\_lock, h:h_n, pi:p\_ind, lg: lsf, ag: asf, zg: zf
begin $DEACT\_BLOCK

/* This function deactivates a block. If this results in a page which*/
/* can be freed, the page name is returned. Otherwise a null name is */
/* r = null */

p = GET\_ABR\_PN (x, b, p_n)
pl = TST\_PNM\_PGL (p, pg\_lock)
while (pl = 1)
  /* Busy wait and */
  pl = TST\_PNM\_PGL (p, pg\_lock) /* lock the pg\_lock */
  h = GET\_VM2\_HN (p, h_n) /* get associated home */
  pi = GET1\_ABR\_PI (x, b, p\_ind) /* get page index */
  lg = GET1\_ABR\_LSF (x, b, lsf)
  ag = GET\_PIR\_ASF (x, pi, asf)
$NULL\_AMENTRY (x, b) /* function which removes AM entry*/
if (lg = 1) then /* if 'lsb' exists */
  $REM\_MBLL (x, b, pi, h) /* removes block from mult. BLL */
else if (ag = 1) then /* else if 'asb' exists */
  begin
    $REM\_MALL (x, pi, p) /* removes entry from mult. ALL */
    $REM\_SBLL (x, b, h) /* removes entry from single BLL */
    FREE\_PIR\_PI (x, pi) /* free page index */
  end
else begin
  $REM\_SBLL (x, b, h) /* removes block from single BLL */
  zg = PUPDATE\_VM2\_PG (p, zf) /* invoke VM2 to make home cons. */
  SET\_HNR\_ZF (zg) /* update HNR with 'zf' value */
  FREE\_PIR\_PI (x, pi) /* free page index */
  r = p
end
if ( (lg = 1) \or (ag = 1)) then /* if page is not to be freed */
  UNLOCK\_PNM\_PGL (p)
return (r)

Fig. A2.10r The $DEACT\_BLOCK Function
$REM_MALL (x: aas_n, pi: p_ind, p: p_n) :-
local curr_a, prev_a, link_a: aas_n, curr_p, prev_p, link_p : p_ind
begin $REM_MALL
    curr_a, curr_p = GET_PNM_ALL (p, (aas_n,p_ind) )
    prev_a, prev_p = null, null:
    while ( (curr_a != x) | (curr_p != pi) )
begin
    prev_a,prev_p = curr_a,curr_p
    curr_a,curr_p = GET_PIR_ALL (prev_a, prev_p, (aas_n,p_ind) )
end
    link_a,link_p = GET_PIR_ALL (curr_a, curr_p, (aas_n,p_ind) )
if ( (prev_a = null) & (prev_p = null) ) /* if header entry */
then SET_PNM_ALL (p, (link_a,link_p) )
else SET_PIR_ALL (prev_a, prev_p, (link_a,link_p) )
    curr_a,curr_p = GET_PNM_ALL (p, (aas_n,p_ind) )
    link_a,link_p = GET_PIR_ALL (curr_a, curr_p, (aas_n,p_ind) )
if ( (link_a = null) & (link_p = null) ) then /* if only one entry */
begin
    SET_PNM_ASF (p,0) /* reset 'asf' in page entry */
    SET_PIR_ASF curr_a,curr_p,0) /* and in page index entry */
end
end $REM_MALL

Fig. A2.10s' The $REM_MALL Function

$REM_SBLLL (x: aas_n, b: b_n, h: h_n) :-
begin $REM_SBLLL
    CHANGE_ABR_ATI (x,b) /* change block entry to inactive */
    SET0_ABR_HN (x,b,h) /* set home name */
    SET0_ABR_HF (x,b,1) /* set home name flag */
end $REM_SBLLL

Fig. A2.10s" The $REM_SBLLL Function
$REM_MBLL (x: aas\_n, b: b\_n, pi: p\_ind, h: h\_n) :-$

local curr\_b, prev\_b, link\_b : b\_n
begin $REM_MBLL$
  curr\_b = GET\_PIR\_BLL (x, pi, b\_n)
  prev\_b = null
  while (curr\_b != b)
    begin
      prev\_b = curr\_b
      curr\_b = GET1\_ABR\_BLL (x, prev\_b, b\_n)
    end
  link\_b = GET1\_ABR\_BLL (x, curr\_b, b\_n^+)
  if (prev\_b = null) then /* if header entry in BLL */
    SET\_PIR\_BLL (x, pi, link\_b)
  else
    SET1\_ABR\_BLL (x, prev\_b, link\_b)
  curr\_b = GET\_PIR\_BLL (x, pi, b\_n)
  if (GET1\_ABR\_BLL(x, curr\_b, b\_n^+) = null) then /* if only 1 entry */
    begin
      SET\_PIR\_LSF (x, pi, 0)
      SET\_ABR\_LSF (x, curr\_b, 0)
    end
  CHANGE\_ABR\_ATT (x, b) /* change block entry to inactive */
  SETO\_ABR\_MN (x, b, h)
  SETO\_ABR\_HF (x, b, l)
end $REM_MBLL$

Fig. A2.10t The $REM_MBLL$ Function
SWITCHACT_VM4_LOC (a: as_n, bs: b_n, bt: b_n) : fault (tf',dfs,df_t,bf',
hf', mof)

local ft:fault, x:aas_n, h:h_n, p:p_n, pi:p_ind, bx:b_n, zg:zf, pt:p_n

begin SWITCHACT_VM4_LOC

ft = null
x = $LOCK_AS_L (a, aas_n)
if (x = null) then
    return ( fault(1) )

if (QEXIST_ABR_DF(x,bs,df) = 1) then /* if source block destroyed*/
    ft = fault(2)
    /* 'df_s' fault */
else if (QEXIST_ABR_DF (x, bt, df) = 1) then /* if target block dest.*/
    ft = fault(3)
else if (GET_ABR_BF(x,bs,bf) = 0) then
    ft = fault(4)
else if (GET_ABR_BF(x,bt,bf) = 0) then
    if (GETO_ABR_HF(x, bt, hf) = 0) then
        ft = fault(5)
    else begin
        h = GETO_ABR_HN (x, b, h_n) /* the code is similar to */
        p = SEARCH_VM2_PG (h, null, p_n) */ ACTIVATE_VM4_EX */

        RPT: if (p != null) then /* However, if a p_ind is */
            p = $LOCKVER_PAGEHOME(p, h, p_n) /* reqd., all blocks in*/
        if (p != null) then /* the source BLL are deact.*/
            begin
                pi = SEARCH_PIR_PI (x, p, p_ind) /* check for 'lsb' */
                if (pi != null) then /* if it exists */
                    $ADD0.BLL(x, bt, pi, p)
            else begin /* else no 'lsb' but 'asb' */
                if (GET_ABR_BF(x,bs,bf) = 1) then /*because of RPT */
                    begin /* loop, check if source blk*/
                        pi = GET1_ABR_PI(x, bs, p_ind) /* still active */
                        bx = GET_PIR_BLL(x, pi, b_n)
                    while (bx != null)
                    begin
                        $DEACT_BLOCK(a, x, bx)
        end
    end
end

Fig. A2.10u The SWITCHACT_VM4_LOC Operation (contd. on next page)
bx = GET_PIR_BLL(x,pi,b_n)
end
end
pi = ALLOCATE_PIR_PI(x,p_ind)
$ADDO_ALL(x,pi,p)
$ADDN_BLL(x,bt,pi,p)
end
end
else begin
pi = GET1_ABR_PI(x,bs,p_ind)
/* Now deactivate all blocks in the source BLL */
bx = GET_PIR_BLL(x,pi,b_n)
while (bx != null)
begin
p = $DEACT_BLOCK(a,x,bs,p_n) /* deacts the block */
end
bx = GET_PIR_BLL(x,pi,b_n)
/* If no 'asb' existed for source blk, then 'p' nonnull*/
/* If 'p' is non null, its pg_lock is still locked */
if (p = null) then /* implies source page cannot be */
p = ALLOCATE_PNM_PG /* used, so allocate new page */
if (p = null) then /* If not possible, */
ft = fault(s) /* 'mof' fault */
else begin
zg = GET_HNR_ZF(h,zf)
pt = PSET_VM2_PH(p,h,zg,p_n+)
if (pt != null) then
begin
FREE_PNM_PG(p)
FREE_PIR_PI(pi)
p = pt
goto RPT /* start all over again */
end
/*

Fig. A2.10u (contd.) The SWITCHACT_VM4_LOC Operation (contd. overleaf)
else begin
  $ADDN_ALL (x, pi, p)
  $ADDN_BLL (x, bt, pi, p)
end
UNLOCK_PNM_PGL (p)
end
end
UNLOCK_ASM_AS (x)
return (ft)
end SWITCHACT_VM4_LOC

Fig. A2.10u (contd.) The SWITCHACT_VM4_LOC Operation

DEACTALL_VM4_AS (a: as_n) : fault(tf')
local x:aas_n, lc:len_curr, i:b_n, p:p_n
begin DEACTALL_VM4_AS
  x = $LOCK_ASM (a, aas_n+)
  if (x = null) then
    return ( fault(1) )
  lc = GET_ASM_LC (x, len_curr)
  for (i = b_n1 to b_nlc by b_n1)
    if (GET_ABR_BF(x, i, bf) = 1) /* if block is active */
      begin
        p = $DEACT_BLOCK (a, x, i, p_n+)
        if (p != null) then
          FREE_PNM_PG (p)
      end
    unlocked_ASM_ASM (x)
  end
DEACTALL_VM4_AS

Fig. A2.10v The DEACTALL_VM4_AS Operation
SUCC_VM4_AAS (ae:asme_n, as_n, asrf, asme_n) :-
  return (SUCC_ASM ASN(ae, as_n, asrf, asme_n))

GETR_VM4_LAS (asme_n) :-
  return (GETR_ASM_LAS(asme_n))

SETR_VM4_LAS (ae : asme_n) :-
  SETR_ASM_LAS (ae)

Fig. A2.10x' The SUCC_VM4_AAS, GETR_VM4_LAS, SETR_VM4_LAS Operations

QREF_VM4_ARF (a: as_n, tf, asrf, len_curr) :-
local x:aas_n, ag:asrf, lc:len_curr
begin QREF_VM4_ARF
  x, ag, lc = SEARCH_ASM ASRF (a, as_n, asrf, len_curr)
  if (x != null) then
    return (1, ag, lc)
  else return (0, 0, null)
end QREF_VM4_ARF

Fig. A2.10x'' The QREF_VM4_ARF Operation

SUCC_VM4_ABK (a: as_n, b: b_n, b_n, brf) : fault (tf)
local x:aas_n, brb_n, bg:brf
begin SUCC_VM4_ABK
  x = $LOCK ASL (a, as_n)
  if (x = null) then
    return (fault(1))
  br, bg = SUCC_ABR_ABK (x,b,b_n, bref)
  UNLOCK_ASM ASL (x)
  return (br, bg)
end SUCC_VM4_ABK

Fig. A2.10x''' The SUCC_VM4_ABK Operation
QREF_VM4_BRF (a: as_n, b: b_n, bf, brf) : fault(tf')
local x: aas_n, bg: bf, brg: brf
begin QREF_VM4_BRF
  x = $LOCK_ASAL (a, aas_n+)
  if (x = null) then
    return (fault(1))
  bg, brg = GETR_ABR_BRF (x, bf, brf)
  UNLOCK_ASM_ASAL (x)
  return (bg, brg)
end QREF_VM4_BRF

Fig. A2.10y' The QREF_VM4_BRF Operation

GETR_VM4_LB (a: as_n, b_n) : fault(tf')
local x: aas_n, b: b_n
begin GETR_VM4_LB
  x = $LOCK_ASAL (a, aas_n+)
  if (x = null) then
    return (fault(1))
  b = GETR_ABR_LB (x, b_n)
  UNLOCK_ASM_ASAL (x)
  return (b)
end GETR_VM4_LB

Fig. A2.10y' The GETR_VM4_LB Operation

SETR_VM4_LB (a: as_n, b: b_n) : fault(tf')
local x: aas_n
begin SETR_VM4_LB
  x = $LOCK_ASAL (a, aas_n+)
  if (x = null) then
    return (fault(1))
  SETR_ABR_LB (x, b)
  UNLOCK_ASM_ASAL (x)
end SETR_VM4_LB

Fig. A2.10y''' The SETR_VM4_LB Operation
Appendix 2b.5  Qualitative Analysis of Blocking in VM4.

There are three locks—ps-lock, as-lock, pg-lock—used in the abstract implementation of the VM4 operations. These are used to lock an address space, active address space and page, respectively. They are locked in the above order and unlocked in the reverse order, thus preventing any deadlock conditions. Table 3.1 illustrates the locks which are locked by each VM4 operation.

The abstractions composing the extension mechanism of the VM4 abstraction are considered as primitive abstractions. This implies that invoking their operations cannot cause a significant blocking overhead. The significant overhead occurs if a VM2 operation is invoked, such that it leads to a blocking within the VM2 abstraction. Note that the VM2 operations, READ, WRITE, PZERO, PUPDATE and PSET can all block while obtaining exclusive access to a page. Since a secondary storage transfer can be occurring with respect to this page, this blocking overhead can be appreciable. With respect to the VM4 implementation, a significant blocking overhead occurs only if all the following conditions hold:

1) Some of the three VM4 locks have been set,
2) a VM2 operation capable of blocking is invoked, and
3) the VM2 operation is blocked within the VM2 abstraction.

All the three set_l operations of VM4, invoke a VM2 operation only after any locks which have been set are unlocked. Thus the first condition does not hold. The ACTIVATE operation invokes the PSET operation of VM2. However, this is done only after a new page has been allocated. Since this automatically sets the pg_lock, no other accesses to the page are possible and, hence, the third condition cannot occur. The same argument holds for the SWITCHACT operation which also invokes PSET.
The $DEACT_BLOCK function invokes the PUPDATE operation of VM2. This function is used by any VM4 operation which requires a block to be deactivated. Since this block to be deactivated is chosen on the basis of infrequency of reference, it is unlikely that simultaneous references to the same page occur. Still there is a small probability of a blocking overhead because of this function. Another aspect of the blocking overhead because of this function. Another aspect of the blocking overhead in the $DEACT_BLOCK function is the busy wait because of removing an associative map entry. Once again, it is unlikely that the block which is to be deactivated has an associative map entry.

Finally, any other VM2 operations which are invoked are only interrogative operations and cause no internal blocking. Thus the second condition does not hold. Thus, we have shown that the blocking overhead, because of the busy wait on the VM4 locks, is minimal.
APPENDIX 2c

Appendix 2c.1  The Group Object Manager (GOR)

The group object manager (GOR) abstraction (Fig. A2.11) manages 'group' objects. Every group object consists of two maps, GIM (group index map) and BIM (block index map), defined on the 'group index_type' and 'block name_type', respectively. The domain for the GIM is decided by the 'g_max' parameter which specifies the number of group indexes. A group index entry can be one of four forms. The forms are distinguished by an 'icode' value. Since each 'small' segment belonging to a group is allocated a unique group index, the icode value corresponds to the 'implementation kind' value of the small segment.

The first form of group index entry (icode = 0) corresponds to a 'very small' segment. The attributes for this form are:

1) A 'block name' which identifies the block which is used in the very small segment implementation.

2) A BASE 'offs' (offset) which indicates the starting logical address of the logical space (in the above block) used by the very small segment.

3) A LIMIT 'offs' which indicates the ending logical address of the logical space.

4) A 'buf' flag (block using flag) which is set to 1, if more than one g_ind is using the same block, that is multiple 'very small' segments are being implemented with the same block.

5) A 'group index' link in the linked list of group indexes (SSL) which have been allocated to very small segments using the same block.
GOR abstraction:

The GOR abstraction is a manager of GROUP objects. A GROUP object contains two maps and is defined as

GROUP object:

attribute interpretation:

icode : denotes implementation kind of segment
sdf : 'security deficiency condition' flag
df : destroyed flag, set to 1 if block is destroyed
bff : block free flag, set to 1 if block is free
icf : icode flag, set to 0 if used by icode=0
buf : block using flag

representation:

1) GIM :-

map0_e : g_ind → icode=0, b_n, BASE: offs, LIMIT: offs, buf,
(g_ind)^+ SSL link

map1_e : g_ind → icode=1, b_n
map2_e : g_ind → icode=2, BASE: b_n, LIMIT: b_n
map3_e : g_ind → icode=3, df

2) BIM :-

mapa_e : b_n → df=1
mapb_e : b_n → df=0, bff=1
mapc_e : b_n → df=0, bff=0, icf=1
mapd_e : b_n → df=0, bff=0, icf=0, (g_ind) SSL header,
LEN_USED: offs

EM: → blk_max : integer = len_max of addr. space
BA: → blk_a loc : integer <= blk_max
BU: → blk_used : integer <= blk_alloc
GM: → g_max : integer
GA: → g_alloc : integer <= g_max
GU: → g_used : integer <= g_alloc

(contd. on the next page)

Fig. A2.11 Specification of the Group Object Manager (GOR)
Operations (defined by GOR) :-

CREATE_GOR_GRP (blk_max g_max, G_n^+)

DESTROY_GOR_GRP (G_n'')

CREATE_GOR_G10 (G_n'',offs,g_ind^+,offs^+(BASE),offs^+(LIMIT))

CREATE_GOR_G11 (G_n'', g_ind^+, b_n^+)

CREATE_GOR_G12 (G_n'', b_size, g_ind^+, b_n^+(BASE), b_n^+(LIMIT))

DESTROY_GOR_G1 (G_n'', g_ind, sdf)

GET_GOR_SP0 (G_n'',g_ind,b_n^+,offs^+(BASE),offs^+(LIMIT))

GET_GOR_SP1 (G_n'',g_ind,b_n^+)

GET_GOR_SP2 (G_n'',g_ind,b_n^+(BASE),b_n^+(LIMIT))

GET_GOR_IC (G_n'',g_ind,i_code)

QCHECK_GOR_SDF (G_n'',g_ind,sdf)

GET_GOR_BM (G_n'', blk_max)

GET_GOR_BA (G_n'', blk_alloc)

GET_GOR_BU (G_n'', blk_used)

GET_GOR_GM (G_n'', g_max)

GET_GOR_GA (G_n'', g_alloc)

GET_GOR_GU (G_n'', g_used)

Fig. A2.11 (contd.) Specification of the GOR
The second form of group index entry (icode = 1) corresponds to a 'little' segment. The only attribute maintained is the 'block name' identifying the block which is solely used by the little segment. The third form of group index entry (icode = 2) corresponds to a 'medium' segment. There are two attributes for this form, a BASE 'block name' and a LIMIT 'block name'. These identify the starting and ending block addresses of the logical space used in the medium segment implementation. The last form of group index entry (icode = 3) corresponds to an unallocated or destroyed group index. The 'df' flag (destroyed flag) is set to 1 if the group index has been destroyed, and is set to 0, if the group index has not yet been allocated.

The block index map (BIM) is defined for the 'block name_type'. The domain of the BIM is decided by the 'blk_max' parameter value which decides the number of blocks available to the segments associated in the group. This is equivalent to the 'len_max' parameter value of the address space which is to be used for the implementation of the segments in the group. A block index map entry corresponds to one of - a destroyed block, a free block, a block used in a 'little' or 'medium' segment implementation and a block being used in implementing multiple 'very small' segments. If blocks are allocated sequentially for little and medium segment implementations, then only the entries of the last form need to be maintained. Thus the BIM has to maintain an entry for each block being used on a very small segment implementation. The attributes for such an entry are a header to the SSL list of group indexes using the same block, and an 'offs' value which indicates the portion of the logical space in the block which has already been used. When a new group index is allocated for a very small segment, the BIM block entries can be
scanned to allocate logical space in some block, based on a 'first fit' or 'best fit' policy.

A group object maintains six object wide attributes. Two of these are the g_max and blk_max values. The 'blk_alloc' attribute indicates the total number of blocks which have already been allocated, while the 'blk_used' attribute indicates the total number of blocks still being used. (The two values are different as some blocks may have been allocated and then destroyed). Similarly, the 'g_alloc' and 'g_used' attributes indicate the number of group indexes already allocated and the number still in use.

Group objects and group indexes are of the create/destroy and hence CREATE and DESTROY operations are defined for each of them. GET operations are defined to access the attributes of a group index and another set of GET operations are defined to access the object wide attributes. An interrogative operation is defined to check if a 'security deficiency condition' exists.

Appendix 2c.2 The Group Space Map (GSM)

The group space map (GSM) (Fig. A2.12) is defined for the 'group name_type*' and maintains a set of attributes for each group name. These attributes are:

1) An 'address space name' identifying the address space which is used for the implementation of the segments in the group.

2) A 'plim_max' value which indicates the maximum page limit for this address space.

* Group names are obtained from capabilities and are identified with a capital letter—'G_n'.
GSM abstraction :-

attribute interpretation :-

policy_n : policy name which identifies block deactivation policy

representation :-

GSM :-

\[ G_N \rightarrow \text{as}_n, \text{plim}_\text{max}, \text{len}_\text{max}, \text{plim}_\text{curr}, \text{policy}_n, \text{g}_\text{lock} \]

operations :-

ALLOCATE_GSM_GME (G_n, as_n)
FREE_GSM_GME (G_n)

GET_GSMASN (G_n, as_n)
GET_GSMPLM (G_n, plim_max)
GET_GSMLM (G_n, len_max)
GET_GSMPLC (G_n, plim_curr)
GET_GSM_PYNN (G_n, policy_n)

SET_GSMASN (G_n, as_n)
SET_GSMPLM (G_n, plim_max)
SET_GSMLM (G_n, len_max)
SET_GSMPLC (G_n, plim_curr)
SET_GSM_PYNN (G_n, policy_n)

TST_GSM_GL (G_n, g_lock)
UNLOCK_GSM_GL (G_n)

Fig. A2.12 Specification of the Group Space Map (GSM)
3) A 'len_max' value which indicates the maximum number of blocks available to the segment implementation.

4) A 'plim_curr' value which indicates the page limit which is to be used in activating the address space.

5) A 'policy name' which identifies a local policy which is to be used for block deactivation in the address space.

6) A 'g_lock' which can be used to lock a group object.

The GSM abstraction defines SET and GET operations to access these attributes and a TST, UNLOCK pair to access the 'g_lock'.
APPENDIX 3

Appendix 3a. Local Abstractions Used by VP2

The VP2 abstraction uses a set of primitive abstractions which are local to the dedicated physical processor, which it uses for its implementation. The first such abstraction is a Base Processor Map (BPM) defined on the 'bpr name_type'. This map maintains for each 'bpr' (base processor) name, the current status of the base processor, the name of the P-procr if one has been allocated for the base processor's execution, and a 'dedf' flag (dedicated flag) which is set to 1 if the processor is dedicated. The BPM also maintains a set of map wide attributes. These identify the number of currently executing base processors, the maximum number of concurrently executing base processors allowed, and the total number of base processors which are executing or ready for execution.

The rest of the local abstractions of VP2 maintain the various queues of blocked base processors. Each queue has an associated semaphore count. The MSEM (monitor semaphore) queue has a capacity of BMAX processor entries, where BMAX is the maximum number of B-procrs which exist. The associated semaphore count can only take a binary value. In the proposed abstract implementations, only one such monitor, namely FFL, is required. The conditional queue (CQ) has a capacity of one processor entry. The indexed queue (IQ) maintains an additional parameter with each processor entry. The range of this parameter is decided by a name_type, which is associated with the indexed queue. The abstract implementation requires only one such indexed queue -- PQ, which is associated with the 'page name_type'. An indexed queue is required to maintain a binary semaphore value for each name in the associated name_type.
All the three operations defined on an indexed queue require a name belonging to the associated name_type as an input parameter.

The synchronization operations defined for message communication, require that a SSEM (synchronization semaphore) queue be maintained for each receiver processor. The capacity of the queue is only one processor entry, but the semaphore count can take any value. The SSEMs are identified by a 'ssem name'. Since each receiver base processor requires a unique SSEM, the 'bpr name' can be used to identify the SSEM. All the sender processors share a common 'sender block list' (SBL) which is simply a list of processor entries. Entries can be removed from the list in any order. There are three other lists which are required by the VP2 abstraction in order to implement the STOP operation. These are discussed in Appendix 3c.

The local abstractions of VP2 can be specified by using the semi-formal techniques of Chapter III. An abstract implementation for the VP2 operations can be specified on this basis.

Appendix 3b. The Base Message Manager (MRL)

The base message manager (MRL) manages the logical message containers and the unsynchronized queues associated with receiver B_procrs. The MRL abstraction consists of a BCR abstraction which manages the base message containers and a BQR abstraction which manages the queues of containers associated with the receiver processors. Message containers are allocate/free kind of resources. The BCR abstraction can be extended from an appropriate virtual memory abstraction. The partitioned real memory abstraction is ideal for this purpose. Physical registers can be used, if a more efficient implementation is called for. Each base mes-
sage container contains, in addition to the message, the name of the sender \texttt{B\_procr} and an 'acknowledgement flag' (af). The BCR defines operations to read and write messages and to access the sender name and 'af'.

The BQR abstraction manages the queue of message containers uniquely associated with each receiver \texttt{B\_procr}. The queues are identified by a 'ssem name', which also identifies the synchronization semaphore maintained by VP2. As we had pointed out, the 'bpr name' can itself be used to identify the message container queues. The capacity for each queue can be set equal to the total number of \texttt{B\_procrs}, if every sender processor is made to wait for an acknowledgement, even if no reply is to be sent. Alternatively, the queue capacity can be decided by the expected number of pending messages at a receiver processor. The BQR abstraction defines an INSERT and a REMOVE operation to insert and remove a message container from the specified queue.

Fig. A3.1 specifies the abstract implementation for the four MRI operations. Observe that the \texttt{READ\_MSG} operation frees the message container if no reply message is to be sent. The \texttt{READ\_ACK} operation always frees the message container.

The design of the message managers at the levels of the physical and system processor abstractions, is analogous. In the former case, the message containers and unsynchronized queue abstractions are implemented at the level of physical circuits.

\section*{Appendix 3c. The STOP Implementation}

The key to the implementation of the STOP operation is the recognition that the invoker processor's synchronization is managed by the VP abstraction which defines the STOP operation. For example, the VP3 ab-
WRITEM MR2 MSG (s: ssem_n, b: bpr_n, m: msg, ag: af, bmc_n) :-
begin WRITEM MR2 MSG

    lm = ALLOCATE BCR_MC (bmc_n+)
    if (lm != null) then
      begin
        WRITE BCR MG (lm, m)
        SET BCR SP (lm, b)
        SET BCR AF (lm, ag)
        INSERT BQR MC (s, lm)
      end
    return (lm)
end WRITEM MR2 MSG

READM MR2 MSG (s: ssem_n, bpr_n, msg, af, bmc_n+) :-
begin READM MR2 MSG

    lm = REMOVE BQR MC (s, bmc_n)
    b = GET BCR SP (lm, bpr_n)
    ag = GET BCR AF (lm, af)
    m = READ BCR MG (lm, msg)
    if (ag = 0) then
      begin
        FREE BCR MC (lm)
        lm = null
      end
    return (b, m, ag, lm)
end READM MR2 MSG

WRITEM MR2 ACK (lm: bmc_n, m: msg) :-
    WRITE BCR MG (lm, m)

READM MR2 ACK (lm: bmc_n, msg)
begin READM MR2 ACK

    m = READ BCR MG (lm, msg)
    FREE BCR MC (lm)
    return (m)
end READM MR2 ACK

Fig. A3.1 The Implementation of the MR2 Operations
straction is the invoker of the STOP_vp2 operation. The VP3 abstraction is itself implemented on a B_procr, which is managed by the VP2 abstraction. Similarly, VP2 which is the invoker of the STOP_vp1 operation is implemented on a P_procr managed by VP1. In the implementation scheme chosen, the VP abstraction defining the STOP operation, informs the invoker of the completion of the STOP action by simulating a message communication from the stopped processor. For instance, at the VP2 level, VP3 does not wait for the completion of the STOP action after invoking a STOP_vp2 operation. When it finally stops the B_procr, the VP2 abstraction has to inform the invoker (which is VP3) of the completion of the STOP action. It does this by simulating a message from the stopped B_procr to the B_procr being used by VP3. As far as VP3 is concerned, a message sent to it at the B_procr level corresponds to an operation invocation. Here, this is a pseudo-operation called 'ACKSTOP' (acknowledge STOP). The point is that when this pseudo-operation is invoked, VP3 can perform the actions which it would have done if it had waited for the STOP_vp2 action to be completed.

When a STOP operation is invoked on a processor, its status can correspond to one of the following:

1) The 'W_STFx' status
2) The 'E_RUNx' status
3) Any other WAIT status.

In the first case, the processor is already in the stopped status, and hence an ACKSTOP pseudo-operation invocation is immediately simulated by the abstraction implementing the STOP operation. In the second case, the processor is executing (physically or virtually) on a lower level processor. The STOP implementation in this case causes a STOP operation
to be invoked on the next lower level VP abstraction. The identity of
the invoker processor and the identity of the lower level processor, to
which the STOP action is transmitted, are entered into a STOP_EXT (stop
external) list of such pairs. Finally, in the last case, the identity of
the invoker processor and the identity of the $B_{procr}$ which is to be
stopped, are entered into a STOP_WAIT list. Before each processor
scheduling, the STOP_WAIT list at that level is searched to check if a
STOP has been invoked on the processor. If so, the processor is sched-
uled for execution by allocating a lower level processor and a STOP is
invoked on this lower level processor. The entry in the STOP_WAIT list
is removed and an entry is entered into the STOP_EXT list. The rest of
the action is similar to the second case. In the last two cases, the
STOP action is completed only when the lower level PR abstraction simu-
lates an ACKSTOP operation invocation to signal the completion of the
STOP action at that level.

It is also possible for a processor which has a STOP invoked on it
to be blocked, either because of invoking a synchronization operation or
because of the VP multiplexing. Thus, the processor status changes from
the second case to the third case. If this happens, the entry correspond-
ing to the processor is removed from the STOP_EXT list and transferred to
the STOP_WAIT list. This ensures that the corresponding VP abstraction
always keeps track of the processors on which a STOP has been invoked.

A VP abstraction simulates the ACKSTOP invocation on the higher level
abstraction by using the appropriate message manager. This manager is at
a lower level in the partial ordering structure. For instance, the VP2
abstraction in simulating the ACKSTOP_VP3 invocation uses the MRI abstrac-
tion to write the message into the message container. It can then perform
the actions as if a SIGNAL_SSEM had been invoked, which causes the message to be delivered to VP3. The algorithms for the STOP_VP2 operation and the simulation of the ACKSTOP_VP3 invocation are shown in Fig. A3.2.

The ACKSTOP pseudo-operation invocation on a VP abstraction can either be due to the VP abstraction having invoked a STOP on a lower level VP for its own multiplexing purposes or because of transmitting the STOP operation which was invoked on it by a higher level VP. In the latter case, the STOP_EXT list contains the identity of the invoker and the lower level processor. The ACKSTOP pseudo-invocation passes, as an input parameter, the identity of the lower level processor on which STOP was invoked. Hence, this is used to search the STOP_EXT list to determine if the completion of the STOP action has to be relayed back to a higher level VP abstraction, by simulating an ACKSTOP at that level. If the ACKSTOP is in response to a local STOP invocation, then the identity of the lower level processor is contained in a STOP_INT list. The actions to be carried out when the pseudo-operation ACKSTOP_VP2 is invoked, are illustrated in Fig. A3.3.

In all the operation invocations on VP abstractions, the name of the sender processor is an implicit parameter. However, the sender name is at the level at which the communication takes place. For example, when an operation is invoked on VP2, the sender is identified by a 'ppr name'. However, the VP2 abstraction maintains the association between the 'ppr' and 'bpr' names and, hence, it can find out the 'bpr name' identity of the sender. This is shown as a SENDER_ID function in Fig. A3.2.

The implementation of the STOP operation at the VP3 level is analogous if a user-defined scheduler is implemented on a dedicated $S_{procr}$ and requires a similar STOP implementation. At the $P_{procr}$ level, the
STOP VP2 PR
(bp: bpr_n)

bs = SENDER_ID
(bpr_n)

Is
STATUS(bp) =
W STEP3

SIMULATE_ACKVP3
(bp, bs)

Is
STATUS(bp) =
E_RUN3

p = 'ppr_n' of
P_procr alloc.
to B_procr

Add the pair
(bp, bs) to the
STOP_WAIT list

Simulate_ACKVP3
(bp, bs: bpr_n)

s = 'sseem_n'
corres. to
B_procr = bs

msg = ACKSTOP-
VP3_PR, bp

Invoke WRITE-
MR2_MSG
lm = (s, bp, msg,
0, lmc_n)

SET_BMC_STATE
(1m, W_STEP3)

Execute
actions for
SIGNAL VP3-
_SESEM (s)

End
ACKSTOP_VP3_PK
(p: ppr n)

bp = GET_BPID
(p, bpr n)

bs = SEARCH_STOP_EXT
(p, bpr n+)

ls

N

bs = null

Y

SIMULATE_ACKVP3
(bp, bs)

Remove entry from STOP_INT list

Schedule a W_RUN2 B_procr on the freed P_procr = p

End

Fig. A3.3 The Actions on an ACKSTOP_VP3 pseudo-invocation
VP1 abstraction simulates (through the physical communciation mechanism) the ACKSTOP_VP2 pseudo-invocation, when the STOP_VP1 action is completed. However, the implementation of the STOP_VP1 operation is different. Unlike the STOP_VP2 operation, the STOP_VP1 operation does not guarantee that the stop action always completes. This is because, before the P_procr completes the current external operation execution, the P_procr executing on it may invoke a VP2 operation which causes the P_procr to be freed and, subsequently allocated to some other P_procr. The VP1 abstraction does guarantee that if the P_procr completes an external operation execution, then it will be stopped and an ACKSTOP_VP1 pseudo-operation will be invoked.

Another point of difference with the VP2 implementation is that, at the VP1 level, there is no lower level VP abstraction to which the STOP can be transmitted. The VP1 abstraction does, however, rely on the physical circuits level to detect the condition when all the internal interpreters in the processor reach a null state, as this implies the completion of an external operation execution.

The significant characteristic of the STOP_VP1 action is that another form of stop operation, called HALT_VP1, can override its effect. The HALT_VP1 operation is used by the VP2 abstraction to stop a P_procr's execution for multiplexing at the VP2 level. This operation does not wait for the current external operation to complete execution. This prevents a deadlock situation if all the P_procrs are executing a busy wait on some internal lock. VP2 uses the STOP_VP1 operation only as part of implementing the STOP_VP2 operation. However, both the STOP_VP1 and HALT_VP1 operations are implemented such that the completion of the stop action is signalled as an ACKSTOP_VP2 pseudo-invocation.
APPENDIX 4

Appendix 4a  A Simulation Experiment

Description—

A simplified version of the simulator proposed in Section V.3c was implemented on a DEC PDP 11 minicomputer, using the mini-UNIX [RiT 74] operating system. The operating system supported a set of concurrent processes. These processes were scheduled on a round robin basis with no priority bias. Since mini-UNIX did not provide an inter-process communication mechanism, this was implemented separately. The operating system allowed only a very inefficient means of passing information between processes and, hence, the universal message communication mechanism proved to be a severe bottleneck in the simulation.

The paging abstraction (VM2) was the one chosen for the simulation. Only three of the paging operations—READ, WRITE and PZERO were implemented. The abstract implementation of these operations are given in Appendix 2a.3. As shown in the structure of the paging abstraction, the extension mechanism is composed of functional abstractions and two map abstractions. Of the functional abstractions, only the Frame Freer required a special form of interaction mechanism, and the others could be treated as ordinary procedures. The two map abstractions are the Page Map and Frame Map (Appendix 2a.1). The paging abstraction interacts with two external mainstream abstractions, VP2 (Base Processor) and Secondary Storage.

The Page Map and Frame Map abstractions were implemented as receiver processes. The VP2 abstraction was used to interact with the Frame Freer and in gaining exclusive access to a page. This abstrac-
tion was also implemented as a receiver process. The Secondary Storage operations were implemented as part of the VP2 abstraction. There were four sender processes. Of these three were 'user' processes which concurrently executed the abstract implementations for the paging operations; the fourth implemented the Frame Freer. Thus the simplified simulator consisted of a set of seven processes, with a set of synchronized communication mechanisms between all sender and all receiver processes.

Each of the three user processes was driven by a page reference string. This driving input also consisted of one of the three paging operations for each page in the reference string. The reference string was generated using an LRU stack model. Each user process could reference one of 24 pages, but the total number of pages which were referenced by all three processes was 55. This was because a few pages could be referenced by all three processes; others could be referenced by only two of the three processes, while a few were referenceable by only one process. The simulation was performed with two sets of page references. In one set, only the READ or WRITE operations were invoked, with equal probability of either invocation. In the other set, there was a small probability (1/25) that a PZERO operation be invoked.

Results—

There were three parameters of the simulation. The ratio between the number of frames (f_max) and the number of Associative Map entries (a_max) was kept constant. The f_max (or a_max) value served as one parameter. The second parameter was the threshold number of frames at which the Frame Freer was activated. The third parameter was the presence of PZERO operations in the input page reference string. In all,
eight simulation runs were performed, with each parameter taking one of
two values. A ninth simulation run was performed with a change in the
f_max/a_max ratio.

The following values were measured at each of the user processes:

1) The fraction of Associative Map (AM) successes for READ
    and WRITE operations.

2) The fraction of Page faults for READ and WRITE operations.

3) The fraction of AM failures, but without a page fault, for
    READ and WRITE operations.

The average values are tabulated in Fig. A4.1.

Another set of values were measured at the VP2 receiver process.

These were:

1) The average length of the queue (Fault Q—FFL) in which
    processes have to wait to be allocated a frame.

2) The idleness of the Frame Freer.

3) The activity of the Frame Freer. (The Frame Freer is idle
    only if it is not executing (active) or not waiting in the
    Fault Q).

4) The average number of processes waiting for completion of
    Secondary Storage activity.

These values are tabulated in Fig. A4.2

The purpose of the simulation was not to measure performance charac-
teristics but to demonstrate the feasibility of a simulator which exe-
cutes abstract programs. The results obtained are more significant in
the context of paging algorithms. This is emphasized by the fact that
the simulator is capable of simulating concurrent reference strings,
while taking into account the nature of implementation for the paging
<table>
<thead>
<tr>
<th>f_max</th>
<th>18</th>
<th>27</th>
<th>27</th>
<th>18</th>
<th>18</th>
<th>27</th>
<th>27</th>
<th>18</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_max</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>thresh</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>PZEROS?</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>.784</th>
<th>.822</th>
<th>.882</th>
<th>.855</th>
<th>.859</th>
<th>.898</th>
<th>.820</th>
<th>.781</th>
<th>.772</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM Read Succ</td>
<td>.805</td>
<td>.845</td>
<td>.902</td>
<td>.859</td>
<td>.870</td>
<td>.907</td>
<td>.848</td>
<td>.808</td>
<td>.803</td>
</tr>
<tr>
<td>AM Write Succ</td>
<td>.047</td>
<td>.025</td>
<td>.025</td>
<td>.040</td>
<td>.039</td>
<td>.021</td>
<td>.024</td>
<td>.044</td>
<td>.025</td>
</tr>
<tr>
<td>Read Page Fault</td>
<td>.027</td>
<td>.012</td>
<td>.019</td>
<td>.041</td>
<td>.041</td>
<td>.017</td>
<td>.010</td>
<td>.026</td>
<td>.019</td>
</tr>
<tr>
<td>Write Page Fault</td>
<td>.077</td>
<td>.061</td>
<td>.087</td>
<td>.098</td>
<td>.096</td>
<td>.074</td>
<td>.060</td>
<td>.079</td>
<td>.105</td>
</tr>
<tr>
<td>AM Fail R Succ</td>
<td>.079</td>
<td>.051</td>
<td>.068</td>
<td>.090</td>
<td>.078</td>
<td>.065</td>
<td>.050</td>
<td>.077</td>
<td>.096</td>
</tr>
<tr>
<td>AM Fail W Succ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Fig. A4.1  Success and Fault Rates at the User Processes*
<table>
<thead>
<tr>
<th></th>
<th>f_max</th>
<th>a_max</th>
<th>thresh</th>
<th>PZEROS?</th>
<th>Avg. FFL Q len.</th>
<th>FF idleness</th>
<th>FF activity</th>
<th>Avg. SS activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_max</td>
<td>18</td>
<td>27</td>
<td>27</td>
<td>18</td>
<td>27</td>
<td>27</td>
<td>18</td>
<td>27</td>
</tr>
<tr>
<td>a_max</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>thresh</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>PZEROS?</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

| Avg. FFL Q len. | .723 | .303 | .487 | 1.15 | 1.30 | .770 | .524 | .876 | .313 |
| FF idleness    | .253 | .510 | .243 | .060 | .087 | .369 | .616 | .385 | .508 |
| FF activity    | .609 | .423 | .487 | .701 | .734 | .494 | .322 | .520 | .398 |
| Avg. SS activity| .187 | .144 | .197 | .209 | .221 | .213 | .129 | .164 | .177 |

**Fig. A4.2 Measurement of Synchronization Activity**
operations. We will now discuss the results.

In Fig. A4.1, we observe that increasing the number of frames from 1/3 the number of pages to 1/2 the number, decreases the fault rate by about 50%. At the same time, increasing the number of associative map entries by 50% brings about only a marginal increase (<10%) in the AM success rate. In any case, we observe that the AM success rate is significantly high while the fault rate is significantly low. Varying the threshold value has no effect on the measured rates.

Fig. A4.3 summarizes the effects of varying the parameters, on the values tabulated in Fig. A4.2. Decreasing the f_max value increases the number of faults. Therefore, the Fault Q waiting, the Frame Freer (FF) activity, and the Secondary Storage activity, all increase while the FF idleness decreases. The inclusion of PZERO operations in the reference string causes the freeing of additional frames, since no frame is to be associated with a zero page. Therefore, this has the same effect as increasing the f_max value.

When the threshold value decreases, the Fault Q waiting increases. This indicates the need for a non-zero threshold value and a concurrently executing Frame Freer (FF). The tradeoff in increasing the threshold is that the FF idleness decreases. When there are no PZERO operations, the FF activity is unaffected by the threshold as the fault rate only depends on the f_max value. However, with PZERO operations, because of the effect of freeing frames associated with zero pages, the FF has to perform superfluous activity because of a higher threshold. Hence, if the threshold decreases, this superfluous activity is decreased. Finally, changing the f_max/a_max ratio has an effect on the above values, only if the f_max value is changed.
As \( f_{\text{max}} \) decreases:

- Avg. FFL Q length - increases
- FF idleness - decreases
- FF activity - increases
- Avg. SS activity - increases

Absence of PZEROs causes:

- Avg. FFL Q length - increases
- FF idleness - decreases
- FF activity - increases
- Avg. SS activity - increases

As threshold decreases:

- Avg. FFL Q length - increases
- FF idleness - decreases
- FF activity - same, if no PZEROs
decreases if PZEROs present
- Avg. SS activity - same

Fig. A4.3 Trends shown by Synchronization Activity Measurements
Appendix 4b. The Segment Capability Map (SCM)

The segment capability map (SCM) is defined for the 'segment name_' type. Since there are four kinds of segment implementation, there are four kinds of SCM entries. A further classification of entries is through the 'mgf' flag. This flag is set to 1 if the segment has multiple copies belonging to different groups of segments. In this case the segment's entry contains a list of—group name, group index, open count—tuples, one for each shared copy. The address of each logical space used by the segment is available only from the appropriate group object. If multiple copies of the segment do not exist (mgf=∅), then the address of the logical space occupied by the segment can be maintained as part of the SCM entry itself.

The SCM abstraction is specified in Fig. A4.4. Each SCM entry contains the segment type (s_type) and segment lock (s_lock) attributes. SCM entries corresponding to 'mgf' being ∅, contain the 'address space name' and 'open count' attributes also. All the SCM entries contain the current length (LENx) and maximum length (LMAXx) attributes of the segment. For segments without multiple copies (mgf=∅), the 'base' (BASx) attribute identifies the starting logical address of the segment.

The SCM abstraction has a CREATE, DESTROY pair of operations which determine the lifetime of SCM entries and the corresponding 'segment name'. A set of GET and SET operations are defined to access the various attributes of the SCM entries. The kind of SCM entry can be changed by a pair of CHANGE operations. An SCM entry for a segment which has multiple (logical) copies, contains a list of 3 tuples. The ADDG, NEXTG and GETM operations are defined to manipulate this list.
SCM abstraction :-

attribute interpretation :-

\[ i_k \] : implementation kind
\[ mgf \] : multiple copies (group) flag
\[ s_type \] : segment type
\[ s_lock \] : segment lock
\[ ocount \] : open count
\[ LENx \] : length of segment
\[ LMAXx \] : maximum length of segment
\[ BASx \] : base of segment's logical space
\[ mcount \] : number of multiple copies

representation :-

\[ S_n \rightarrow i_k = 3, mgf = 0, s_type, s_lock, plim_curr, ocount, policy_n, LEN3: (b_n,offs), LMAX3: b_n, as_n \]

\[ S_n \rightarrow i_k = 3, mgf = 1, s_type, s_lock, mcount, as_n, LMAX3: b_n \]
\[ \text{list of (G_n,g_ind,ocount) tuples}, \text{LEN3: (b_n,offs)} \]

\[ S_n \rightarrow i_k = 2, mgf = 0, s_type, s_lock, ocount, as_n, \]
\[ LEN2: (b_n,offs), LMAX2: b_n, BAS2: b_n \]

\[ S_n \rightarrow i_k = 2, mgf = 1, s_type, s_lock, mcount, LMAX2: b_n, \]
\[ \text{list of (G_n,g_ind,ocount) tuples}, \text{LEN2: (b_n,offs)} \]

\[ S_n \rightarrow i_k = 1, mgf = 0, s_type, s_lock, as_n, ocount, \]
\[ LEN1: offs, BAS1: b_n \]

\[ S_n \rightarrow i_k = 1, mgf = 1, s_type, s_lock, mcount, \]
\[ LEN1: offs, \text{list of (G_n,g_ind,ocount) tuples} \]

\[ S_n \rightarrow i_k = 0, mgf = 0, s_type, s_lock, as_n, ocount, \]
\[ LEN0: offs, LMAX0: offs, BAS0: (b_n,offs) \]

\[ S_n \rightarrow i_k = 0, mgf = 1, s_type, s_lock, mcount, LEN0: offs, \]
\[ \text{list of (G_n,g_ind,ocount) tuples}, \text{LMAX0: offs} \]

\(\text{contd. on next page}\)

Fig. A4.4 Specification of the Segment Capability Map (SCM)
Operations:

CREATE_SCN_SE (i_k, S_n)
DESTROY_SCN_SE (S_n)

GET_SCN ASN (S_n, i_k, as_n^+)
GET_SCN_MG (S_n, mgf)
GET_SCN_STP (S_n, s_type)
GET_SCN_OCT (S_n, oct)

GET_SCN_LEN (S_n, i_k, (b_n^+,offs^+))
GET_SCN_LMX (S_n, i_k, b_n^+ or offs^+)
GET_SCN_BAS (S_n, i_k, (b_n^+,offs^+))

TST_SCN_SGL (S_n, s_lock)
UNLOCK_SC_SGL (S_n)

CHANGE_SCN IK (S_n, i_k)
CHANGE_SCN_MG (S_n, mgf)

ADDG_SCN_MG (S_n, (G_n,g_ind))
NEXT_SCN_MG (S_n, (G_n,g_ind))
GETM_SCN_OCT (S_n, G_n, g_ind, oct)
GETM_SCN_MCT (S_n, mcount)

Fig. A4.4 (contd.) Specification of the Segment Capability Map