MIT/LCS/TR-164

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

David Patrick Reed

June 1976

The research reported here was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641 which was monitored by ISTAO under contract No. F19628-74-C-0193.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

ACKNOWLEDGMENTS

A very large number of persons and organizations deserve my thanks for helping me complete this research. I am sure there are some who I will forget to mention, so let me apologize in advance for any omissions.

Professor Schroeder, my thesis supervisor, contributed a great deal of time and effort to help me develop and clarify a large set of ideas. I am especially grateful for the quick turnaround he has given the many drafts of chapters I have given him in the last hectic weeks of thesis preparation.

Professor Saltzer and Dr. David Clark provided much inspiration along the way, and helped crystallize a number of the ideas in the thesis.

Raj Kanodia, Bob Mabee, Doug Wells, and Bernie Greenberg helped by providing a sounding board for my early ideas at innumerable luncheon discussions.

Phil Janson and Doug Hunt have helped me understand the issues involved in structuring an operating system. Phil's work on abstract type structures especially helped in the development of some of the central ideas in the thesis.

Bob Frankston has taken the time to read several of the drafts of my thesis, and has been very helpful in designing the implementation of some of my ideas.

The CSR Volleyball Crew has helped me keep in shape mentally and physically through all the trials of thesis preparation.

The final two people I would like to thank are Lynn, my spouse, and Colin, my newborn son. They both have put up with my non-stop pace during the last days of the thesis. Without their love and understanding, I doubt if I would have succeeded in finishing the thesis.

This research was performed in the Computer Systems Research Division of the M.I.T. Laboratory for Computer Science. It was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641, which was monitored by ISTAO under contract No. F19628-74-C-0193.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM *

by

David Patrick Reed

ABSTRACT

This thesis presents a simply structured design for the implementation of processes in a kernel-structured operating system. The design provides a minimal mechanism for the support of two distinct classes of processes found in the computer system -- those which are part of the kernel operating system itself, and those used to execute user-specified computations. The design is broken down into two levels, one which implements a fixed number of virtual processors, which are then used to run kernel processes, and are multiplexed to provide processes for user computations. Eventcount primitives are provided, in order to provide a simple unified interprocess control communication mechanism. The design is intended to be used in the creation of a secure kernel for the Multics operating system.

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

---

*This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on June 14, 1976 in partial fulfillment of the requirements for the degree of Master of Science.

TABLE OF CONTENTS

ACKNOWLEDGMENTS 2
ABSTRACT 3
TABLE OF CONTENTS 4
LIST OF FIGURES 7

1. Introduction 9

1.1 Brief Statement of Problem and Results 11
1.2 Example System 15
1.3 Abstract Types 17
1.4 Layering of Abstract Types 19
1.5 Related Work 21
1.6 Plan of Thesis 25

2. Model of Processor Multiplexing 29

2.1 Definition of Processor 30
2.2 Definition of Process 32
2.3 Processor Multiplexing 33
2.4 Processor Multiplexing Model 36
2.4.1 Centralized Control of Processor Multiplexing 38
2.4.2 Distributed Control of Processor Multiplexing 40
2.4.3 Comparison of Distributed and Centralized Control 42
2.5 Processor Reconfiguration 44
2.6 Interprocess Control Communication 49
2.7 The Virtual Processor Stopped State 57
2.8 Summary 59

3. Multiple Levels of Processor Multiplexing in a Layered System 61

3.1 The Cache Management Pattern of Type Extension 62
3.2 Building Two Levels of Virtual Processors 66
3.3 Disentangling Virtual Memory from Processor Multiplexing 70
3.4 Use of Processes as Abstract Type Managers 71
3.5 Two Levels of Scheduling 79
3.6 Problems of a Processor Hierarchy 80
3.6.1 Efficiency of Multiple Levels of Scheduling 80
3.6.2 Protection of Low-level Type Managers from Level 2 82
3.6.3 Cross-level Interprocess Control Communication 84
3.6.3.1 Level 2 Advance and Await Algorithms 86
3.6.3.2 Inward Signalling 87
3.6.3.3 Outward Signalling 87
3.7 Summary 90

4. Level 1 Virtual Processor Interfaces 91

4.1 Level 1 Virtual Processor Interface 92
4.2 Limited Supply of Level 1 Processors 94
4.3 Multiprogramming of Real Processors Among Level 1 Processors 95
4.4 Execution States of Level 1 Processors 96
4.5 Scheduling Controls 99
4.6 Changing the Bindings of Level 1 Processors 100
4.7 Interprocess Control Communication 102
4.8 Special Eventcounts 104
4.9 Fault Interface 105
4.10 Processor Interrupt 107
4.11 Processor Reconfiguration 108
4.12 Parameter Passing To Level 1 Processor Operations 109

5. Level 1 Processor Implementation 113

5.1 Overall Structure of the Implementation 114
5.2 Hardware Architecture 118
5.2.1 The Processor Control Processor 119
5.2.2 General-Purpose Processors 120
5.3 Data Bases 127
5.4 Operation of the Processor Control Processor 130
5.5 GPP operation 139
5.6 Implementing Level 1 Processors on Traditional Hardware 146
5.7 Simulating the Processor Control Processor 146
5.8 I/O Devices That Send Interrupts 149
5.9 Summary 150

6. Level 2 Processor Interface and Implementation 151

6.1 Level 2 Processor Interfaces 152
6.1.1 Creation and Deletion of Processors 153
6.1.2 IPCC Interfaces 155
6.1.3 Processor Interrupts 157
6.2 Structure of the Second Level Processor Manager 161
6.2.1 Level 2 Data Bases 163
6.2.2 Processes of the Second Level Manager 167
6.2.3 Eventcount Implementation 171
6.2.3.1 Advance 171
6.2.3.2 Await 173
6.2.3.3 Set_processor_interrupt 175
6.2.3.4 Outward Signalling 175
6.2.4 Scheduling Policy 176

7. Using Level 1 Processors in the Operating System 181

7.1 Permanently Bound Processes 182
7.2 I/O Device Management 183
7.3 Kernel Type Managers as Processes 187
7.4 Explicit Recognition of Parallelism in the System Design 190
7.5 Resulting Structure 192

8. Conclusions and Suggestions for Further Research 195

BIBLIOGRAPHY 201
Appendix A: Summary of Level 1 Interface 205
Appendix B: Summary of Level 2 Interface 207

LIST OF FIGURES

Figure 1.1: Removing Mutual Dependencies 13
Figure 1.2: Type Extension Hierarchy for VM Objects 19
Figure 2.1: Multiplexing 2 Real Processors 34
Figure 2.2: Processor Multiplexing Loop 37
Figure 2.3: Processor Reconfiguration States 45
Figure 2.4: Processor Multiplexing Loop with Reconfiguration 46
Figure 2.5: Processor Multiplexing Loop with IPCC 55
Figure 2.6: Processor Multiplexing Loop with Stopped State 58
Figure 3.1: Cache Mgmt. Pattern for Page Object 63
Figure 3.2: Cache Mgmt. Pattern for Virtual Processor 65
Figure 3.3: Two Level Processor Hierarchy 67
Figure 3.4: Two Level Processor Multiplexing Loop 69
Figure 3.5: Permanently Bound Type Manager Processes 78
Figure 4.1: States of Level 1 Processor 97
Figure 4.2: Level 1 State Data 100
Figure 4.3: Level 1 Fault Data 106
Figure 5.1: Processor Communication in Level 1 Implementation 115
Figure 5.2: Priority Queue and Await Table 116
Figure 5.3: Hardware Communication Paths 120
Figure 5.4: GPP Internal Memory 121
Figure 5.5: Level 1 Processor State Block 124
Figure 5.6: Basic GPP Cycle 126
Figure 5.7: PCP Algorithm Flow Chart 132
Figure 5.8: GPP Responses to UNBIND and INVOKE-LEVEL1 141
Figure 6.1: Processor Interrupt Model 158
Figure 6.2: Processors and Data Bases of Level 2 162
Figure 6.3: Level 2 Processor Table Entry 164
Figure 6.4: Await Table Structure 166
Figure 6.5: Actions of the Binder/Scheduler and Unbinder 168

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Chapter One

Introduction

A major goal of current research on computer systems is ensuring the correctness of operating system software. Although many complex operating systems have been designed and built, the best that can be said of these systems is that they seem to work correctly. It is not yet possible to prove, or otherwise ensure, that a complex operating system such as Multics [19] works correctly -- in fact, specifying what correct operation means in the case of systems like Multics is very difficult. One important part of specifying and proving the correct operation of a system like Multics is simplifying its design to a point where its operation is easily understood. A clear understanding of the basic operating system mechanisms and implementation techniques is a prerequisite to achieving this simplification.

The research reported here is an attempt to understand the impact of processor multiplexing on the design and operation of an operating system. The processes created by processor multiplexing serve two purposes in the design of an operating system. First, they are used to isolate user-specified computations from each other in order to prevent unpredictable or undesirable interactions. Second, they can be used as a tool for structuring the algorithms of the operating system itself. A clear understanding of the design and implementation of processor multiplexing mechanisms that support

these purposes is a necessary part of the understanding needed to simplify and structure the design and implementation of operating systems.

The research reported here is part of a project to design a security kernel [28] for the Multics operating system. The security kernel of an operating system is a part of the operating system that, if correct, guarantees that the operating system as a whole enforces constraints on information flow that prevent unauthorized release (to users) of information stored in the system. In Multics, individual user computations are isolated from each other as distinct processes executing on distinct virtual processors. This isolation is used as a tool for controlling the propagation of information within the system; consequently, the processor multiplexing mechanisms that implement the virtual processors must be part of the security kernel of the system. By simplifying the mechanisms of processor multiplexing, the security kernel is made simpler and easier to prove correct.

The security kernel also can be simplified by structuring it as a set of loosely coupled processes. Consequently, a simple processor multiplexing mechanism that enables the construction of the kernel as a set of processes contributes to the goal of kernel simplification.

1.1 Brief Statement of Problem and Results

In virtual memory operating systems such as Multics [19], TENEX [1], and VM/370 [8], the management of processors and the management of virtual memory cannot be considered separately. The processor multiplexing algorithm calls upon virtual memory management functions to perform such operations as loading into primary memory the environment description (1) of a process so that a processor can execute the process. The virtual memory management algorithm uses various functions of processor management in order to obtain resources to run, and to organize the mechanism processes use to wait for pages to arrive from secondary storage.

The initial goal of the research described in this thesis was to disentangle this mutual dependency. The first step has been described by Huber [10]. He has developed an implementation of part of the virtual memory system of Multics that runs in special processes created by the operating system. By slightly extending his work, the virtual memory algorithms can be built so that they need not use features such as interrupt masking and busy-waiting, which interact strongly with the operation of processor management.

---

(1) In Multics, the environment description is the descriptor segment.

In order to completely disentangle virtual memory management from processor management, however, the dependency of processor management on the virtual memory must be removed. The major source of this dependency is the need for processor management to load and unload per-process data bases that must be in primary memory while the process is executing on a processor, but are too large and too numerous to be permanently resident in primary memory.

To remove the mutual dependency between processor multiplexing and virtual memory, processor multiplexing is done at two levels, in the design proposed in this thesis. The first level of processor multiplexing does short-term multiprogramming among a small set of processes. The per-process data bases for these processes are in primary memory. This first level thus simulates the existence of a small number of virtual processors that subsequently will be called level 1 processors. Since at level 1 all per-processor data bases are in primary memory, there is no need for level 1 to depend on the virtual memory management algorithms.

The second level multiplexes these level 1 processors to create level 2 virtual processors that are used to run user processes. Level 2 is responsible for loading the per-process data bases into primary memory when a process is loaded into the level 1 processor. Level 2 thus depends on the virtual memory algorithms.

The virtual memory algorithms themselves are built out of special processes, called kernel processes, that are permanently loaded into level 1

processors. The second level of processor multiplexing does not multiplex level 1 processors running kernel processes, so kernel processes are not dependent on the second level of processor multiplexing. By this strategy, the dependencies between processor multiplexing and virtual memory management have been changed from that shown in figure 1.1a, to that shown in figure

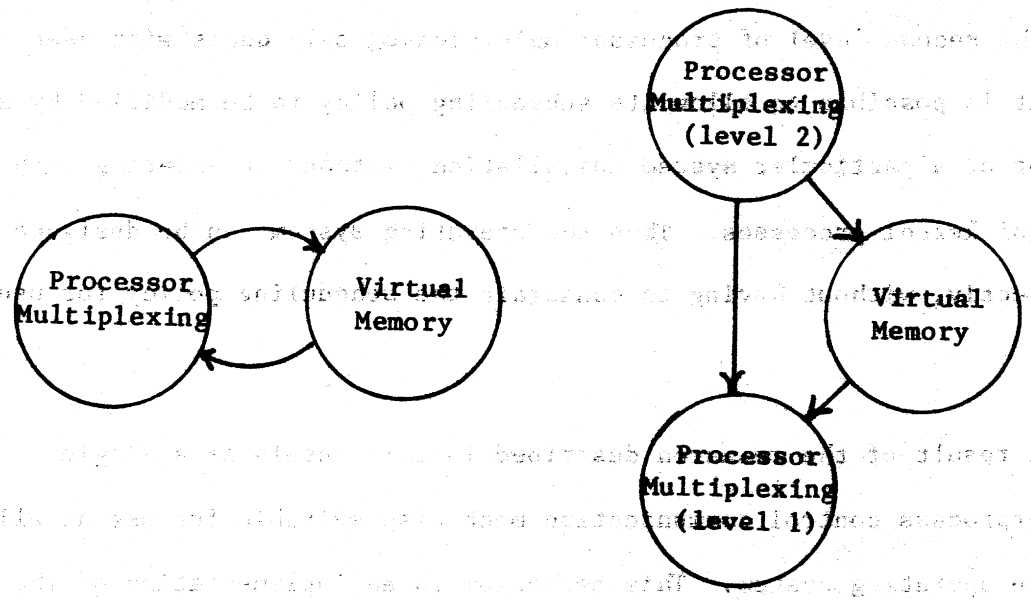


Figure 1.1
Removing Mutual Dependencies

The two-level structure has other advantages. It allows elimination of interrupt-driven code from the I/O device management part of the system. Instead of running I/O device management at interrupt time, I/O devices can be managed by from high-priority kernel processes running on level 1 processors, thus isolating and simplifying the control structure of such algorithms.

The interactions of processor reconfiguration with other functions of the operating system have been limited also by this structure. Only the first level of processor multiplexing need be cognizant of the number of physical processors on the system. Additions and deletions of physical processors can occur at any time, except when processors are in the middle of switching from one level 1 processor to another.

Since the second level of processor multiplexing only deals with user processes, it is possible to allow its scheduling policy to be modified by an administrator of a particular system installation, without interfering with the actions of kernel processes. Thus the operating system can be designed to operate correctly, without having to constrain the scheduling policy for user processes.

A final result of the research described in this thesis is a single unified interprocess control communication mechanism suitable for use at all levels of the operating system. This mechanism is an implementation of the eventcount model proposed by Kanodia and Reed [12]. Since this mechanism encompasses the capabilities of most known interprocess control communication mechanisms, it is flexible enough for all operating system and user interprocess control communication. In addition, the virtual memory is adequate for storage and protection of eventcounts. The processor multiplexing algorithms do not have to implement special objects for the purpose of interprocess control communication.

The proposed design is described in terms of abstract types. Janson [11] has provided a structure for the virtual memory of Multics based on an abstract type structure. This mode of description is quite natural for discussion of the modularization of a computer system, and causes the intermodule dependencies to stand out. I have extended his work a little bit, to deal with the problems of multiplexing processors to produce new abstract objects called virtual processors.

1.2 Example System

At times in this thesis, it will be useful to talk about an example operating system. A very simple system, modeled after Multics, will suffice. I will consider an operating system that provides a large number of user processes that can operate in a shared virtual memory. The virtual memory is composed of segments, built out of fixed-length pages. The data contained in pages resides permanently in a set of records on disks. The data is accessed by a demand paging algorithm that brings the contents of disk pages into primary memory as desired. Several hardware processors provide processing power for the system. In order to allow the processors to access the memory using virtual addresses, each processor has a hardware address translation mechanism, called a map. (1) The map is loaded with a set of (virtual

---

(1) The map consists of some hardware like the Multics address appending hardware, and some data that is interpreted by the map hardware such as the

address,primary memory address) pairs, so that if the map is presented with a virtual address that is the first component of a pair, it will give back the second component as the actual primary memory address to access. If a virtual address is presented that is not in the map, the processor will stop executing the current instruction, forcibly transferring control to a predefined address called the fault handler.

Processor multiplexing in this system will be done at two levels, for the reasons discussed earlier. The first level of processor multiplexing creates a set of virtual processors that can be used either to run processes directly, or to produce the next level of processors by a second level of processor multiplexing. This second level implements the processors for user processes, called user virtual processors.

I/O is done from primary memory buffers accessible to both the general purpose physical processors of the system, and to special purpose I/O processors that actually perform I/O. I/O processors communicate status information back to the general purpose physical processors through special buffer areas called mailboxes, and send interrupts in order to get their attention.

Multics descriptor segment and page tables. The data can reside in primary memory, and may be shared by several processors at once.

1.3 Abstract Types

An abstract type is a class of objects in the system for which there is a defined set of operations. The difference between an abstract type and the classic notion of type is that the user of an abstract type need not know the representation of the object, or the algorithms used to implement operations defined on the type. Further, the only operations allowed to be performed on the objects are specified by the definition of the type.

The concept of abstract type is quite attractive for the structuring of large systems because the actual implementation of a type of object is hidden from the algorithms that make use of the type. This results in the kind of structuring prescribed by Parnas's "information hiding principle" [21], for decomposing a system into modules. Further, abstract types fit naturally into the structure of an operating system since a major job of an operating system is to multiplex a set of physical resources to produce a set of virtual resources that can be viewed as objects of abstract type. I will show that this is exactly what happens in processor multiplexing.

An abstract type consists of a set of objects and a set of operations. The set of operations defined on the objects of the abstract type is implemented by algorithms collectively called the (abstract) type manager. Only the type manager algorithms are allowed to manipulate the representation

of the objects. The type manager may be actually implemented as a set of closed subroutines, or as a process (or set of processes) to which messages may be sent, or as macros (open subroutines) which are expanded into the code of programs using the abstract type. It is important to emphasize this point, because I will show later that it is sometimes useful to implement type managers using one or many of these techniques.

In the example system, there are several objects that can be viewed as having abstract type. A disk block, for example, is an object that has two defined operations -- read-block, which reads a block of data out of the disk block returning a string of bits of fixed size, and write-block, which takes a string of bits and moves it into the disk. A word in virtual memory is also an abstract object. Two operations that can be carried out by instructions in user processes are read-word, which obtains the contents of a word named by a particular virtual memory address, and write-word, which takes a bit string and stores it in the object specified by a particular virtual memory address.

Processors, both real and virtual, can be viewed as objects of abstract type. Viewing processors as objects that can be controlled by operations on the processor objects is basic to the structuring method I use in this thesis.

1.4 Layering of Abstract Types

The abstract type idea clearly furnishes a useful way to view the virtual objects seen at the external interface of an operating system, but for the design of a large operating system the abstract type idea is equally important in structuring the internal implementation of the system. Janson [11] discusses how this structuring might be applied to a system like Multics. For

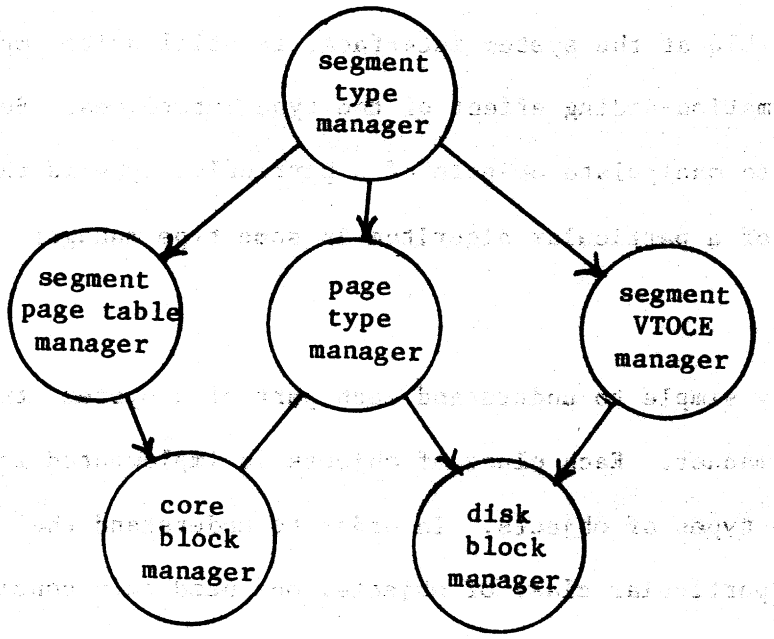


Figure 1.2
Type Extension Hierarchy for VM Objects

example, see figure 1.2, which shows the hierarchy of objects out of which the virtual memory of the example system is built. Each of the circles in the figure shows a type manager, labeled by the type of object implemented. The

arrows between the circles indicate that objects of the type at the tail of the arrow are represented in terms of objects of the type at the head of the arrow. (1) At the bottom, the physical storage objects of the system are shown. Pages, fixed size blocks of virtual storage, are implemented from these basic objects. Then out of pages and core blocks that hold map data, segments are built.

This is an example of using type managers inside the system for the structuring effect alone, since the lower level abstractions of the system are not visible to the user of the system. The use of abstract types at these levels, though invisible at the system interface, is still quite important because of the information-hiding effect of the type interfaces. Because the only module allowed to manipulate objects of a particular type is the type manager, the effect of a particular algorithm in some type manager can be localized.

It is relatively simple to understand each part of a system structured in such a hierarchical manner. Each class of objects is implemented in terms of a small set of other types of objects. In order to understand the implementation of a particular class of objects, one need only consider the behavior specified for objects of that class and the behavior specified for

---

(1) The representing object participates in this representation either as a storage container for objects, a mapping function to translate the external name of the abstract object into the names of objects in its representation, or as an agent to perform the operations that implement the abstract operations on the object.

objects in the classes used in the representation. It is not necessary to consider the implementation of objects used in the representation. Thus the implementation of each abstract type may be considered separately.

In this thesis, processor multiplexing at two levels is described in terms of abstract types and type managers. The abstract type structure of an operating system is used to show the interdependencies between modules of the operating system. The interdependencies between processor multiplexing and the rest of the operating system are shown clearly in this model. The problems resulting from these interdependencies can thus be discussed easily.

1.5 Related Work

There are several classes of related work. First of all, there is a large body of literature on concurrent processes. Second, there is some literature which talks about the implementation of concurrent processes by processor multiplexing on various systems, including Multics. Third, there is a growing body of literature on the use of abstract types to structure system design, and some recent work applying these ideas to hierarchical design of operating systems. Finally, the use of processes within the kernel of an operating system has a small body of associated literature.

It is not worthwhile to list here all possible references to literature on concurrent processes as a model for parallel, asynchronous computations.

The work of several authors in the application of these models to operating systems problems is directly relevant; other work on the modeling of parallel computations is not specifically related to the work in this thesis. Dijkstra [6] defined the notion of a sequential process, primarily as a mechanism for dealing with simultaneous activities. Dennis [5] among others has described the utility of the process concept in guaranteeing that independent computations do not interfere with each other. Saltzer [25] has described how processes can be used as a way of controlling the allocation of processor and memory resources to users of a computer system.

Actual implementations of the process concept also abound, so again I will only touch the high points. Saltzer [25] also outlines the basic algorithms of processor multiplexing. Rappaport [23] describes an early version of the Multics process implementation in his thesis, and discusses many of the engineering tradeoffs involved in its design. The Virtual Machine concept implemented in IBM's VM/370 (formerly CP/67) operating system [17] is also a form of the process concept.

Work on abstract types and their use in structuring systems is progressing rapidly. SIMULA [4] and CLU [13] are programming languages that include abstract type definition as basic structuring tools. Liskov [14] is currently investigating the structuring of programs using abstract types. The Hydra operating system kernel [30] is designed to support abstract types that can be used to build operating systems. Janson [11] has investigated the use of abstract types in structuring the design of operating system kernels, and

described the cache management pattern of type extension that is extended to processor multiplexing in this thesis.

The area of literature closest to the topics discussed in this thesis describes the use of processes to structure the kernel of an operating system. Dijkstra's THE system [7] was the first kernel in which the process concept was introduced at a low level in the kernel. Unfortunately, there is little reference in the available literature on the THE system to show how processes are actually used in the kernel. Unlike the design proposed here, the process implementation is at a lower level in the THE system than the virtual memory. Consequently, the per-process data must remain permanently loaded into primary memory, so the number of processes allowed is severely limited. Dijkstra proposes the idea of structuring an operating system into modules in a hierarchy based on frequency of use of the modules. In the design proposed here, the two levels of processor multiplexing satisfy this criterion.

Brinch-Hansen [3] has described an operating system for the RC4000 computer that uses processes communicating via messages to structure the kernel. Sturgis [29], in describing the CAL TSS system, shows how processes are used to structure the kernel of that system. Rowe, of the University of California at Irvine, [24] has described a distributed operating system where processes are used as building blocks to make up the kernel, and where control of the communication paths among the processes is used to provide reliability. Huber [10] has described how processes might be used to simplify the structure of part of the virtual memory implementation in Multics, and has made use of a

primitive version of the kernel processes designed in this thesis. Hoare [9] has described the implementation of a virtual memory system as a set of processes where each page is assigned a process -- while this is probably not practical as a way of implementing a virtual memory interface, nonetheless it suggests several potentially practical ways of implementing a virtual memory system.

More recently, at SRI a structured design for the kernel of a complex operating system was completed. In this design, described by Neumann et. al. [20], processes are implemented at a low level, and then enhanced at a higher level. This idea is quite similar to the design discussed in the present thesis, but unfortunately the SRI design is only a specification and does not incorporate any notion of a reasonable implementation -- or even what the algorithms executed by the implementation might be. The SRI design is concerned only with structuring of the system, not with the performance costs or efficient implementation of their design. Bredt and Saxena [2] have described the algorithms of a layered system similar to the SRI design where two levels of virtual memory implementation are interleaved with two levels of process implementation. As in the SRI design itself, a framework is provided for a two-level process implementation, but incorporating such features as multiple real processors, interprocess interrupts, and variable scheduling policy is ignored. They do not discuss the problem described later in the thesis as the outward signalling problem, which seems to be an inherent problem in a layered operating system design. Another problem with their

paper is that they do not take into account the other uses to which processes might be put in an operating system, such as I/O device multiplexing, and the peculiar requirements imposed on the design of processes by those applications.

1.6 Plan of Thesis

The material presented in the rest of the thesis falls naturally into three parts. The first part, covered in chapters two and three, will discuss the issues involved in the design of a process implementation at an overview level. The second part, covered in chapters four, five, and six, discusses the functionality of the proposed design and describes a particular implementation for the Multics operating system. Finally, chapter seven discusses the effect of the design in simplifying the rest of the operating system, and chapter eight summarizes the thesis, suggesting areas of further research.

Chapter two specifically covers the basic model of process implementation used in the thesis -- that of multiplexing a relatively small number of functional processing units (either actual hardware processors or software virtual processors) among a larger number of processes. I define several terms, including processor, virtual processor, and process. The model developed in this chapter will be used as the basis for the model of processor multiplexing at two levels, and to describe the design proposed in chapters

four, five and six. In addition to processor multiplexing, processor reconfiguration and interprocess control communication are incorporated in the model.

Chapter three develops the two level processor multiplexing structure. I show how the implementation fits the cache management pattern of type extension described by Janson [11]. I also model the actions of the implementation in terms of the model developed in chapter two. Three problems that can result from this structure, having to do with efficiency and interaction between the levels, are described and their solutions are shown to be possible within the structure.

Chapter four begins the discussion of the actual design. It contains a complete description of the interface presented by level 1 virtual processors.

Chapter five completes the discussion of level 1, by discussing implementations that can achieve the level 1 interface efficiently on a computer system such as Multics. A new hardware architecture is proposed to simplify the control of processor multiplexing. Mechanisms for simulating this architecture on a more conventional architecture are described, to show that level 1 can be built on more conventional systems.

Chapter six describes the interface and implementation of level 2 processors. The functionality of level 2 processors differs from level 1; these differences, such as administratively variable scheduling policy, creation and deletion of level 2 processors, processor interrupts, and outward signalling eventcounts are described.

Chapter seven shows how an operating system is built on the basis provided by level 1 processors. The use of level 1 processors within the operating system to provide resources to abstract type managers and to I/O device management is described. The advantages of using processes running on dedicated level 1 processors inside the kernel of the operating system are briefly described.

Chapter eight summarizes the work done, attempts to give an indication of the difficulty of integrating an implementation into the present Multics system, and the benefits deriveable therefrom. It also discusses how closely the initial goals of the project were met, and the impact of the general approach taken in this design on future development of kernel-based operating systems.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Chapter Two

Model of Processor Multiplexing

In order to understand how two levels of processor multiplexing can work, one must thoroughly understand what processor multiplexing does. In this chapter, the concepts of process and processor are carefully defined. From this basis, a model of processor multiplexing is developed, showing clearly how real processors can be multiplexed to provide multiple virtual processors for the execution of processes.

Along the way, reconfiguration of processors and interprocess control communication are incorporated into the basic processor multiplexing model.

In the next chapter, the model of processor multiplexing is extended to two levels of processor multiplexing. To enable the extension to be made, the model developed here incorporates the idea of a stopped virtual processor whose state can be manipulated.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

2.1 Definition of Processor

In this thesis, several kinds of processors are discussed. These entities are all called processors because they share certain properties. To make certain that my assumptions are understood, I take the trouble to define processors here.

The basic function of a processor is to perform a sequence of operations on objects in its environment. The environment of a processor is a set of objects. For example, the environment of a physical processor is that portion of memory that it can access through its address mapping hardware. Typically the environment is specified by an object, such as the Multics descriptor segment, that in turn names another object. I shall assume that the objects that specify environments can be shared among several processors, thus giving the processors identical accessing environments. (1)

A processor has internal memory, called its state, that it uses to pass information from one operation to the next. The processor determines the next operation to perform by interpreting an instruction, found in the processor's environment by an instruction pointer that is part of the processor state.

---

(1) This does not imply identical access permissions, however. The access rights specified in the environment specification are interpreted relative to the domain of execution (part of the processor state), as in the Multics descriptor segment.

The environment specification used by the processor is named by a value in the processor state. Also included in the processor state is the name of the current protection domain in which the processor is executing.

Each operation performed may modify the contents of the processor's internal memory. In particular, it changes the instruction pointer to select the next instruction to be interpreted.

As an object of abstract type, a processor may be part of the environment of other processors. The operations that can be performed on a processor object are: loading a new state into the processor, extracting the current state from the processor, causing the processor to run, and causing the processor to stop.

A processor can be a physical object, such as the Honeywell 68/80 CPU that is used to implement Multics. The processor registers comprise the state of that processor. The environment of the processor includes all of the primary memory that is accessible through the processor's descriptor segment.

In this thesis, two other kinds of processors are described. These processors are virtual processors -- meaning that they have no direct hardware manifestation. Instead, they are simulations of processors, achieved by using physical processors to interpret the instructions to be executed by the virtual processor.

2.2 Definition of Process

The word process has been used in many senses in the literature of computer science. Usually, it has been used to refer to one of two things -- a virtual processor as defined above, or what is called a process in this thesis. I make a careful distinction in this thesis between the meanings of the words process and processor to avoid confusion.

A process is the sequence of actions taken by some processor. In other words, it is the past, present, and future "history" of the states of the processor. Each processor, be it virtual or physical, has one associated process for the duration of its existence. Thus, the process associated with a physical processor is the sequence of operations that have been performed by that processor since its creation and that will be performed up until its destruction.

The act of logging in to a computer system can be viewed as creating a processor for the user. The user can then cause this processor to perform operations on his behalf. The history of these operations will be called the user's process. If there is but one physical processor in the computer system, it will carry out the operations of all of the users' processes. The process associated with the physical processor is thus a merging of the operation sequences that make up the users' processes.

Quite often, the words process and processor can be used interchangeably -- this is the source of the confusion between the words. For example, consider the modification of a particular file by a processor. This can also be said to have happened as part of the process (in the process) being executed by the processor.

The major difference between a process and a processor is that a process is a sequence of actions while a processor is an actor. A processor is an object in the computer system and subject to operations that may be executed in the system, while a process is just a view of the actions taken by the system that can be imposed in retrospect. A process results from the actions of a processor.

2.3 Processor Multiplexing

The two levels of virtual processors in the design are created by a technique called processor multiplexing. This technique originated in the first multiprogramming computer systems as a way of achieving more efficient use of scarce processor resources. Saltzer [25] has modeled the mechanisms of processor multiplexing in his Ph.D. thesis. I will recapitulate the basic issues here.

Processor multiplexing is the simulation of a number of distinct virtual processors by a smaller number of real processors. Each of the virtual

processors executes a sequence of operations in time. These sequences are actually performed by the real processors. The many processes of the virtual processors are actually merged together, creating the processes of the real processors.

The result of any one of this merging is that the operations of any one of the virtual processors are carried out in the same temporal sequence that they would have been, had the virtual processor been real. Successive operations of the same virtual processor may be separated by a gap of time during which operations of another virtual processor are being executed by the real processors. Successive operations of a virtual processor may also be

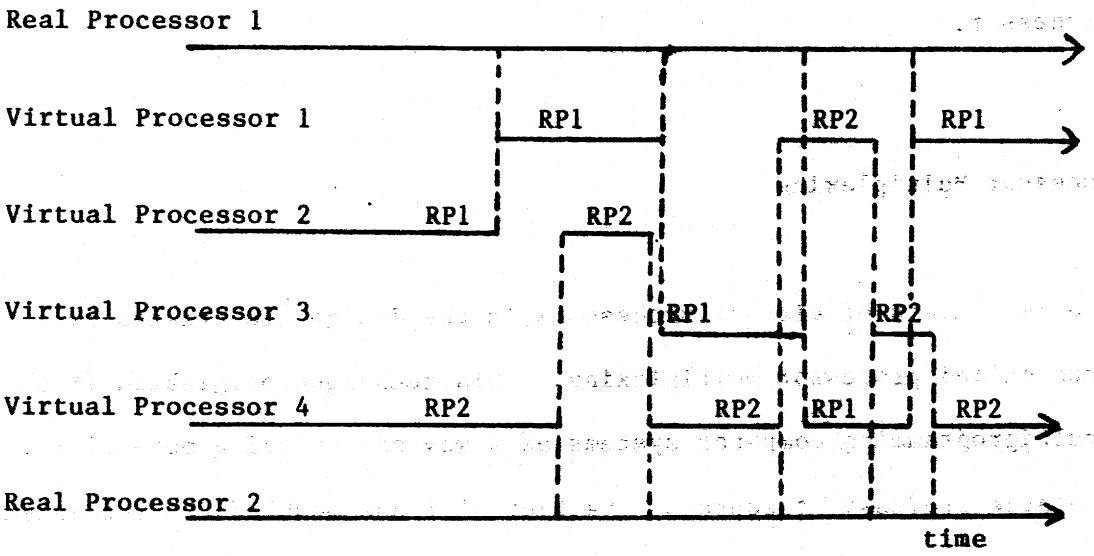


Figure 2.1
Multiplexing 2 Real Processors

executed by different real processors. Figure 2.1 shows how the operations of 4 virtual processors might be mapped into the operation sequence of 2 real processors.

To define a term used frequently in this thesis, a virtual processor being simulated by a set of multiplexed real processors is bound to one of the real processors whenever its process is being executed by a real processor. Thus virtual processor number two is bound to real processor number one during the first time interval in figure 2.1. More loosely, one can say that a process is bound to a processor when that processor is carrying out actions that are part of that process. A process is permanently bound to a processor when that processor can only execute operations of that process (the process is thus the process defined by the sequence of actions of the processor).

There are concrete aspects to this binding. When real processor A is bound to virtual processor S, processor A's internal memory contains S's current state. Similarly, processor A accesses objects through S's environment. When S is not bound to a real processor, its state is stored in a piece of memory from which it can be loaded later into an real processor's internal memory.

In addition to providing the operations of the real processors to the virtual processors, processor multiplexing can create new functionality. The virtual processors can execute an operation that causes execution of future operations to be delayed until some future event happens. They also can execute an operation that signals such an event. Such operations are called interprocess control communication. The wait operation is not an operation that requires real processor resources -- it is rather an operation that inhibits use of real processor resources by the virtual processor.

Processor multiplexing also requires a policy. Given a number of virtual processors to which an real processor may be bound, at any one time the processor can only execute one. The choice of the processor to choose is made by some algorithm, called the processor multiplexing policy algorithm. This policy algorithm receives as input the set of processors that can be run, and chooses which one is to run and for how long.

2.4 Processor Multiplexing Model

In order to discuss two levels of processor multiplexing, one needs to understand how processor multiplexing at one level is done. In this section, I will provide a model of this behavior.

I assume that the real processors are capable of executing all of the instructions that appear in virtual processors, except those that control processor multiplexing and interprocess control communication. (1) In some cases, there will be more than one real processor, although the number of virtual processors will usually exceed the number of real processors given. I also assume that a real processor can store the contents of its private state memory, and load a new set of values into this private memory from main memory. The effect of loading the private memory of the real processor is to

---

(1) In particular, the structure of the environment description in the real and virtual processors will be the same, and the addressing mechanisms will be the same. Since real processors can only directly address primary memory, the same will be true of virtual processors.

cause it to interpret a new sequence of instructions specified by the newly loaded state.

Real processors and virtual processors go through the cycle detailed in

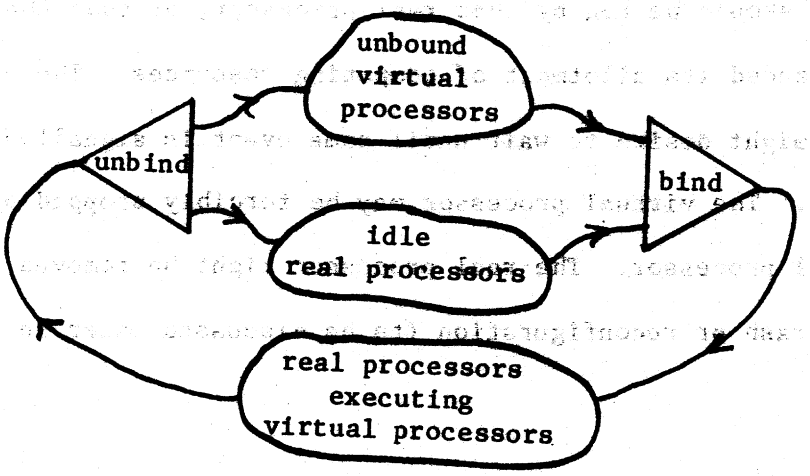


Figure 2.2
Processor Multiplexing Loop

figure 2.2. From the point of view of a real processor, it is bound to (and executes) a virtual processor until some time at which it is unbound. The box labeled "unbind" represents the unbinding of a real processor from its assigned virtual processor. Unbinding results in placing the virtual processor state in memory in a pool of virtual processor states. The real processor is then placed in a pool of available idle processors. The "bind" operation in the figure then takes a real processor from the pool of idle real processors and a runnable virtual processor from the pool of runnable virtual

processors (selected by the real processor multiplexing policy) and binds the two together.

A real processor bound to a virtual processor enters the unbind operation under several conditions. The policy algorithm may decide that another virtual processor should be run by that real processor, or that the virtual processor has exceeded its allotment of computing resources. The virtual processor itself might desire to wait until some event is signalled by another virtual processor. The virtual processor may be forcibly stopped or deleted by another virtual processor. The real processor might be removed from the system due to a crash or reconfiguration (to be discussed later in this chapter).

In this model, no indication is given that specifies the actual agent that causes the bind or unbind operations, or the agent that executes the actual processor multiplexing policy algorithm. This is intentional, since in the design I propose later in the thesis, the agent will vary from level to level. However, I would like to discuss here the alternatives that are possible.

2.4.1 Centralized Control of Processor Multiplexing

One scheme for the control of processor multiplexing is based on the idea of a central agent. This agent is responsible for the binding of virtual

processors to real processors. All binding of virtual processors to real processors is caused by the action of the central agent, while unbinding of real processors from virtual processors also may be controlled by the central agent. Of course, the virtual processors themselves have influence over the unbinding decision, since a virtual processor that chooses to wait or otherwise gives up its need for a real processor can cause real processors to stop running that virtual processor. The central agent is, however, notified if such an event occurs, so the central agent interacts on each binding and unbinding of a real processor.

Typically the central agent is a computation carried out in the computer system. Cases where the central agent is a human operator fit this model, but are not of interest here. The central agent can be viewed as a process, since it is a sequential computation that performs operations on the state of the system. The agent cannot, of course, be the process of a virtual processor, since it must make decisions about virtual processors when they are not running. If the agent unbound itself, then it could never make the decision to rebind itself. For this reason, the central agent in this scheme of processor multiplexing must be permanently executing on a dedicated real processor. (1)

---

(1) This real processor does not have to be a general purpose processor such as the ones being multiplexed. It is not multiplexed, and performs a fixed function. Consequently it could be a hard-wired processor, or a microprocessor executing a firmware algorithm. As is shown later in the thesis, the effect of a dedicated processor can be obtained by cheating a little bit.

Given this constraint, the central agent may implement any arbitrary policy for scheduling the binding of virtual processors to real processors. The implementation of such policies will usually require some kind of communication channel between the real processors and the central agent. The primary reason for such a communication channel is that the virtual processors being scheduled by the agent need to be able to wait for other virtual processors to do certain things. While the agent can reasonably bind a waiting virtual processor to a real processor, such a decision is quite wasteful, since the virtual processor will unbind itself immediately. This would reduce the economic justification for doing processor multiplexing, since real processor time would be wasted doing non-useful work.

2.4.2 Distributed Control of Processor Multiplexing

An alternative scheme for the control of processor multiplexing is one in which the functions are accomplished by a distributed algorithm executed by all real processors. In this scheme, the policy used to select a new virtual processor for a real processor in the bind operation is implemented on each real processor, as is the policy used to control which real processors to unbind. Through careful coordination, real processors unbind themselves when they choose to, send recommendations to other real processors to unbind themselves, and choose which virtual processor to next bind to.

Please note that in this scheme it is not the case that control of processor multiplexing is done in the virtual processors being implemented. If this were the case, the virtual processors could become unbound in the middle of telling the real processor which virtual processor next to bind itself to. Often an algorithm, such as that used by the current Multics, is described as being so distributed among the virtual processors. In fact the computations of such an algorithm are only executed when the real processor cannot change its execution point to another stream of instructions (inhibited mode), and so are done exactly as if they were unit operations in the real processor. I assume that the special privileges needed to control processor multiplexing in each processor are only accessible in a special domain found in each real processor's environment.

In the distributed control scheme, it is possible that each real processor can implement a different policy in assigning itself to a new virtual processor. Thus, the set of policies that can be implemented is apparently richer. As noted above, there needs to be a communication channel between the real processors and the policy-implementing algorithms. In the distributed case, each real processor must be able to send information to all other real processors.

In the distributed case, interlocking between different instances of policy algorithms becomes necessary since real processors may come unbound, or choose to bind themselves to virtual processors, simultaneously. This is just one aspect of the general need for harmonious cooperation among the policy algorithms executed by each real processor.

2.4.3 Comparison of Distributed and Centralized Control

Although no algorithm for control of processor multiplexing will match one of these extremes precisely, it is instructive nonetheless to study the advantages and disadvantages of the centralized and distributed control schemes.

The main advantage of the centralized algorithm is unity. Since the centralized scheme is executed as a process permanently bound to one real processor, it can be described by a single program that makes one decision at a time. Such a description has an obvious effect on the ease of understanding the programs of the processor multiplexing policy, by making them simply structured. Also, since in dynamic execution, one decision is made at a time, it is fairly easy to model the state transition of bindings of virtual processors being implemented, since there are no simultaneous transitions. Thus the system can be treated as a synchronous system, at least as far as the binding and unbinding of real processors to/from virtual processors is concerned.

The main advantage of the distributed scheme is autonomy. As mentioned earlier, each real processor can control its destiny relatively independently of the other real processors. The policies implemented by different real processors may vary. Also, the autonomy afforded by a distributed system can

increase the amount of parallel activity possible in determining policy. Thus the fact that a real processor is busy finding another virtual processor to execute need not prevent another real processor from doing the same. To the extent that these activities can be carried on in parallel, and to the extent that the real processors can execute in parallel, this can be an economic advantage.

The advantages of each scheme are disadvantages of the other. In the centralized case, the lack of autonomy prohibits the parallelism afforded by the distributed scheme. In the distributed case, the autonomy makes it potentially very difficult to understand the interactions of the different algorithms executed by different real processors.

It is possible, however, to incorporate parallelism into the centralized scheme to achieve more rapid execution of the central agent. The parallelism is achieved by implementing the central agent as a group of cooperating parallel processes (implemented on dedicated real processors) that take advantage of any inherent parallelism there is in the centralized policy algorithm. The sequentiality of bindings and unbindings must be preserved in this case, but the time required by the central agent to perform each action can be reduced, thus reducing the economic cost due to real processors waiting to be rescheduled by the central agent.

The distributed scheme, in general, seems to have the greater disadvantage. I am predominantly interested in simplifying the structure of

the processor multiplexing algorithms, rather than improving their performance. Performance is an issue, of course, but the main goal of this work is to understand the clearest and simplest structure that achieves the desired effects, and then to propose ways of improving performance within that structure if necessary.

2.5 Processor Reconfiguration

For many reasons, it is useful to allow real processors to be added to and deleted from the computer system while it is running. For example, real processors may be shared between two computer systems. In this example, one real processor can be moved from one system to the other in order to balance the computing resources on each system to the presented loads. Another example would be the automatic deletion of a faulty real processor when the malfunctioning is detected. The faulty processor then can be repaired and added back to the computer system while the rest of the system has continued to run. Processor reconfiguration is a required feature of any system that hopes to become a computer utility that remains up without interruption all day.

Schell [27] has developed a model of processor reconfiguration. In it the two real processor states, bound (to a virtual processor) and unbound, are each split into two states (see Figure 2.3), according to a second criterion. This criterion is whether the real processor is available for multiplexing or

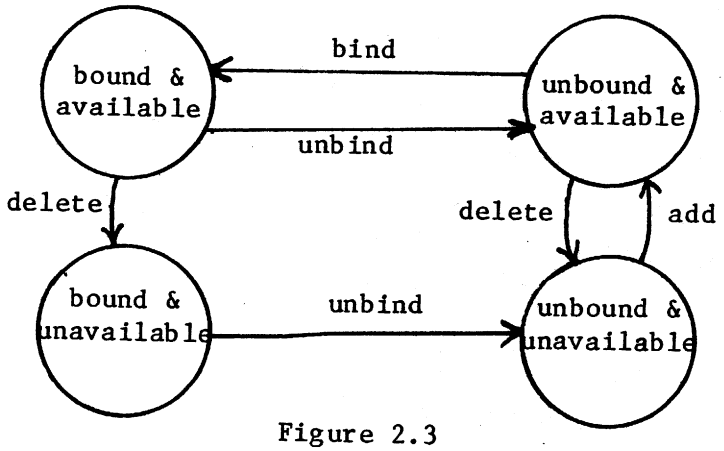


Figure 2.3
Processor Reconfiguration States

not. In figure 2.3 it is seen that deconfiguration of a real processor consists of marking it as unavailable, and then unbinding it. Adding the real processor back consists of marking the real processor available, and binding it to a virtual processor.

Processor reconfiguration fits nicely into the model of processor multiplexing. A real processor can be deleted from the system by marking it unavailable, then causing the real processor to execute unbind, which takes special action on an unavailable real processor and places it in an unavailable real processor pool. An unavailable real processor can be added to the system by causing it to enter the processor multiplexing loop as if it had just become unbound from a virtual processor, as an idle real processor. Figure 2.4 shows the revised processor multiplexing loop.

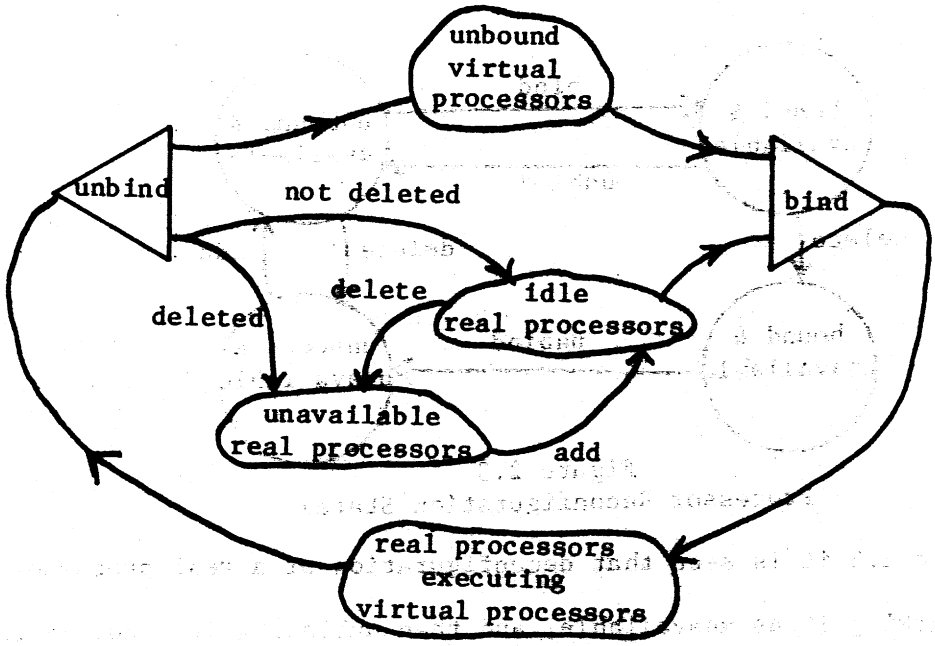


Figure 2.4
Processor Multiplexing Loop with Reconfiguration

At each processor reconfiguration, the policy algorithm must be made aware of the new state of the reconfigured processor. For example, the policy being implemented might be an assignment of static priorities to virtual processors such that the highest priority virtual processors are guaranteed to run when they are runnable. In this case, deconfiguration of a real processor that is running a virtual processor of higher priority than some other virtual processor that is assigned to a real processor will require reshuffling of the processor assignments. The policy algorithm must thus be brought into action whenever a real processor is deleted. Similarly, when a real processor is added, the policy algorithm must specify what to do with the new processor. The policy algorithm specifies this by controlling the choice made by the bind operation.

A concept closely related to processor reconfiguration is the initialization and shutdown of the computer system. Luniewski, in his master's thesis [15], has discussed how to view most of the tasks of system initialization as adding additional system resources to a minimal system.

Processor multiplexing may be initialized by starting with no real processors and a set of virtual processors to run. Obviously, this is a system at rest, with no changes being made to objects in the system. One can then add processing units, in exactly the same way that processors are added in reconfiguration, binding them to virtual processors in the processor multiplexing loop. (1) This reconfiguration proceeds until all the available processing units are added to the computer system. The system continues to execute the computations specified by the virtual processors of the system as this reconfiguration proceeds. The only effect of adding real processors will be to increase the effective speed of the system.

Processor multiplexing can be stopped and the system shut down by deconfiguring all of the real processors from the system until there are no real processors left bound to virtual processors. The system will then remain at rest until the real processors are added again. All of the state of the system will then reside in the descriptions of the virtual processors, and the state of the deconfigured real processors will be irrelevant.

---

(1) With a centralized agent, there is no difficulty in adding the first real processor (other than the agent, which is expected to always be part of the system) because the central agent performs additions. In the distributed processor multiplexing case, though, adding the first real processor is slightly more tricky than adding the later ones.

A system crash that is due to a software detected error is just another deconfiguration of processors, as far as processor multiplexing is concerned. (1) In a system crash, all real processors are deleted. This view of a system crash is important, since it defines the fact that the state of the system is completely represented in the virtual processor states, and no relevant information is left in evanescent real processor registers. For this reason, if the cause of the crash is repairable, the system state can be restarted at the point of the crash. An example of this might be a brief power-line failure, detection of a parity error in memory that can be corrected from redundant information, or other possible system states.

An important facet of processor multiplexing is that the dependence of the system on having a particular number or set of real processors can be reduced to a minimum. There is no need for virtual processors to be aware of reconfigurations of real processors, other than in terms of the total amount of processing power that can be delivered to the set of running virtual processors in a fixed period of time.

---

(1) Obviously, some system crashes cannot be viewed as deconfigurations of all processors. Most crashes in the Multics system, however, take the form of orderly shutdown of the system by software.

2.6 Interprocess Control Communication

It is the responsibility of the computer system to provide mechanisms for communication between cooperating processes. There are really two different kinds of communication that processes must be able to achieve. There must be a way for processes to exchange data in some way. This mode of communication will be called interprocess message communication (IPMC) in this thesis. There must also be a way for processes to wait for data prepared by other processes, and for processes that prepare such data to signal that it is available. This mode of communication is qualitatively different from communication of data. Since the effect of such communication is purely to reenable a waiting control point, it is called interprocess control communication (IPCC). Together, IPMC and IPCC are called interprocess communication (IPC).

In a computer system that allows sharing of virtual memory segments between processes, there is no need for a special interprocess message communication facility to be built into the processor multiplexing algorithm. Shared virtual memory segments provide an extremely high bandwidth data communication channel between the processes sharing the segments. The protection facilities provided by the computer system for shared virtual memory segments will suffice to handle interprocess message communication.

Further, shared segments are sufficiently primitive that any protocol for interprocess message communication can be built using them. For these reasons, I assume that interprocess message communication will be handled outside of the scope of this thesis.

Interprocess control communication, on the other hand, is intimately related to the structure of the processor multiplexing mechanism. The ability of a virtual processor to indicate that it does not need real processor resources until a particular event happens is basic to the economic advantage of processor multiplexing. If a dedicated real processor were actually available for each virtual processor, busy-waiting (1) would be an adequate interprocess control communication mechanism.

In order to keep processor multiplexing simple, it is desirable to have a very simple interprocess control communication mechanism. Saltzer [25] has discussed the general problem in detail in his Ph.D. thesis. The essence of the problem is to be able to communicate to a virtual processor that is waiting for an event to happen one bit of information that indicates that the event has happened. The information that the event waited for has happened is stored as a single bit in the memory of the system, known as the wakeup-waiting switch. The wakeup-waiting switch is initially off. When the event occurs, the wakeup-waiting switch is set on. In order to wait for an event, the virtual processor indicates to the processor multiplexing algorithm

---

(1) Busy-waiting is repeatedly testing the state of a shared memory word in a loop.

that it cannot run until the wakeup-waiting switch is turned on, and then unbinds itself from the real processor executing it.

In Saltzer's thesis, there is one wakeup-waiting switch per virtual processor, which represents the current event being waited for. Thus, the virtual processor wakeup-waiting switch is multiplexed to represent many different events as its process proceeds, with the requirement that when a virtual processor restarts after waiting, it must clear the wakeup-waiting switch for the next wait.

This multiplexing of the meaning of the wakeup-waiting switch of a virtual processor makes it more difficult to ensure that virtual processors are awakened at the right time. If virtual processor A can wakeup virtual processor B, there is no guarantee that the reason virtual processor B is waiting is the reason virtual processor A wakes B up. Virtual processor A's wakeup will then be misinterpreted by B, or ignored by B. In the first case, B will proceed under the false assumption that the event awaited happened, while in the second case, B will lose the wakeup (1) even though it may be meaningful to B at a later time. These problems can be serious for system security, if the wakeups are intended for a protected system operation in B's virtual processor, because a wait operation executed outside of the protected part of the system can receive IPCC signals intended for the protected part. The arrival of an IPCC signal can carry privileged system information. An

---

(1) This is the "lost wakeup" problem described by Saltzer.

unprotected receiver may either gain unauthorized access to privileged information, or prevent it from reaching its proper destination. These occurrences cannot be prevented because B is multiplexing the meaning of his wakeup-waiting switch, and so must allow A to wake him up at all times, even though B waits for A's event only sometimes.

Another interprocess control communication mechanism is the semaphore. This is quite similar to the mechanism described by Saltzer, except for the fact that the semaphore is a wakeup-waiting switch that represents a class of events independent of the events of interest to one virtual processor. It is possible to give a semaphore a semantic meaning because new semaphores can be created for each semantically different class of events. In order to implement semaphores in the model, the processor multiplexing algorithm must be informed of all V operations to semaphores, and must keep track of the set of virtual processors that are waiting for each semaphore to indicate that the event has occurred.

Unfortunately, semaphores have several disadvantages. First, they are limited to cases where the occurrence of an event will allow a fixed number of virtual processors to proceed out of the waiting state. Second, because of this limitation, the ability to proceed past a P operation on a semaphore automatically becomes a kind of scarce resource that can be used as a communication channel among processes that wait on the semaphore.

This latter point is quite important in a secure system design. Although communication of information is inherent in the IPCC mechanism between the virtual processor that causes an event and the virtual processors that await the occurrence of that event, there is no inherent requirement that virtual processors waiting for the same event to occur should have a communication path among themselves.

For these reasons, along with the need to deal with synchronization in distributed systems, Kanodia and Reed [12] have developed an IPCC mechanism that is in some sense more general than either semaphores or block-wakeup, but is still very simple. I will briefly describe the mechanism here, and indicate how it fits into the model of processor multiplexing.

An eventcount is an object in the system that represents a class of events that will eventually occur. This class of events is ordered, so that by the time event N occurs, all events numbered from 0 to N-1 will have occurred. Consequently, the set of events that have occurred at any particular time can be represented by the number of the last event to occur. This number is known as the current value of the eventcount.

There are three operations which may be performed on eventcounts. One may read an eventcount to obtain the current value. One may advance an eventcount. This will increment the current value by one, and serves to indicate that a new event in the class of events represented by the eventcount has occurred. Finally, a virtual processor may await a particular event in

the class associated with the eventcount. This last operation requires that the eventcount, and the number of the event be specified. Await will prevent the virtual processor from proceeding until the current value of the eventcount exceeds the number of the event.

The eventcount IPCC mechanism has the useful property that two virtual processors waiting for events in the same class (thus recorded in the same eventcount) do not have an inherent intercommunication path. The enabling of one virtual processor to proceed does not automatically disable any other virtual processors from proceeding, and allows broadcasting events to multiple virtual processors -- a function not easily achieved using semaphores. Consequently, this mechanism is more desirable for use in a secure system. Further, the implementation of eventcounts is not inherently more difficult than that of semaphores.

The eventcount mechanism fits into the processor multiplexing model quite simply. The processor multiplexing loop is modified to have a pool of waiting virtual processors, as well as a pool of ready-to-run virtual processors. Figure 2.5 shows this modification. The name of the eventcount and the value awaited must be stored with the virtual processor state. A special kind of unbind operation will put the virtual processor in the waiting pool instead of the ready-to-run pool if the awaited eventcount hasn't yet been advanced to the awaited value. The advance operation informs the processor multiplexing algorithm of the new value of the advanced eventcount, causing any virtual processors in the waiting pool waiting on this eventcount to be moved to the

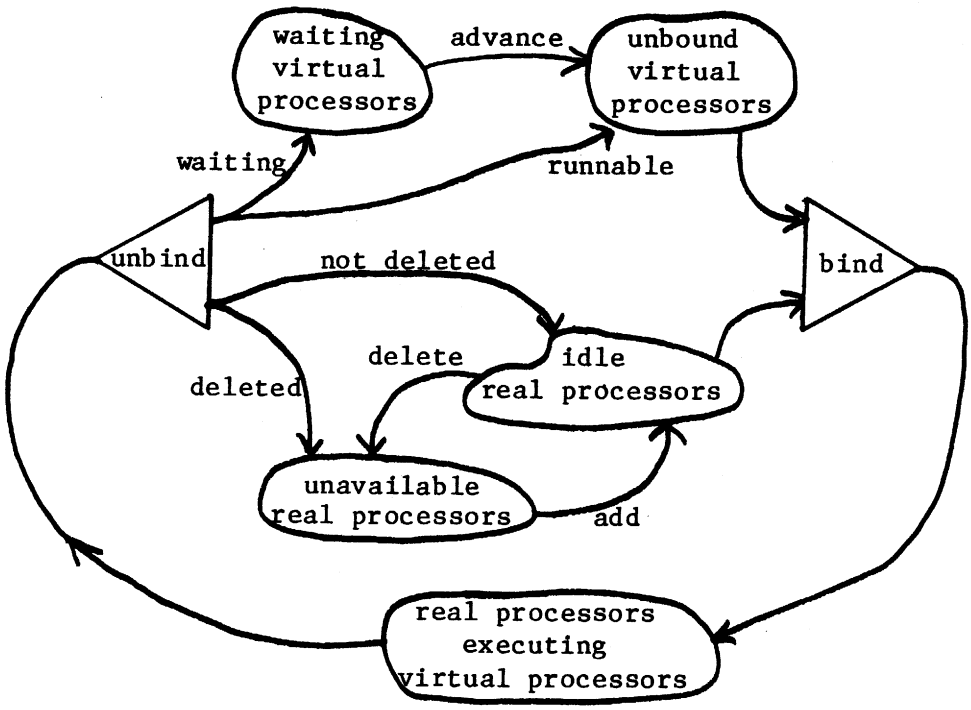


Figure 2.5
Processor Multiplexing Loop with IPCC

ready-to-run pool. In this implementation, the only storage required is the ability to remember the names and values of eventcounts that are actually being awaited by virtual processors. A way to search the waiting pool on each advance operation for virtual processors waiting on the advanced eventcount is required. (1)

---

(1) This search can be done in time proportional to the logarithm of the size of the waiting pool, at least, if a balanced tree scheme, such as AVL trees is used for searching. If hashing is used, one may be able to do better (although frequent deletions usually reduce the efficiency of a hash table).

An alternative implementation of eventcounts would include in them a list of the virtual processors waiting for changes to the eventcount. Along with the name of the waiting virtual processor would be the value waited for. The await operation would then just add the current virtual processor to the list associated with the eventcount awaited, and then unbind the process from its real processor indicating that it should not be run. When the eventcount is advanced, any virtual processors that are waiting for the new value are removed from the list, and placed in the ready-to-run pool so that they may be run.

This latter implementation can require more storage (a list pointer per eventcount, whether a virtual processor awaits it or not). The first implementation may have a certain cost due to searching the waiting pool on each advance operation for virtual processors awaiting the advanced eventcount.

The first model implementation has the nice property that if a segment were used to store the eventcount, only the advance operation would have to modify that segment. Thus, if segments have individual permissions for inspection of values and modification of values, the segment access control may be used to guarantee the security of both the IPMC mechanisms of the system (implemented in segments), and the IPCC mechanisms of the system. Using this implementation thus makes the protection mechanisms of the system more uniform and simple to understand. Stopping a virtual processor is also made simpler, because the eventcount itself need not be modified.

2.7 The Virtual Processor Stopped State

In order to multiplex virtual processors as discussed in the next chapter, a mechanism is needed to change the state of a virtual processor, just as there is a mechanism for changing the state of a real processor. In the model as so far described, the state of a virtual processor is sometimes kept in the waiting pool, sometimes in the ready-to-run pool, and sometimes in some real processor. To simplify matters, I introduce a new state of a virtual processor, called the stopped state. When a virtual processor is in this state, its private state memory can be changed and examined by other virtual processors. The stopped state is added by modifying the processor multiplexing loop to include a pool of stopped virtual processors. Figure 2.6 shows the stopped modification. A virtual processor enters the stopped pool when some virtual processor executes a stop operation specifying this processor, or when the virtual processor stops itself because it has exceeded a resource limit. A virtual processor can enter the stopped state directly from the ready-to-run pool or the waiting pool, or it can be marked as to-be-stopped and unbound from its real processor if it is running. The unbind operation puts virtual processors in the stopped pool if they are so marked.

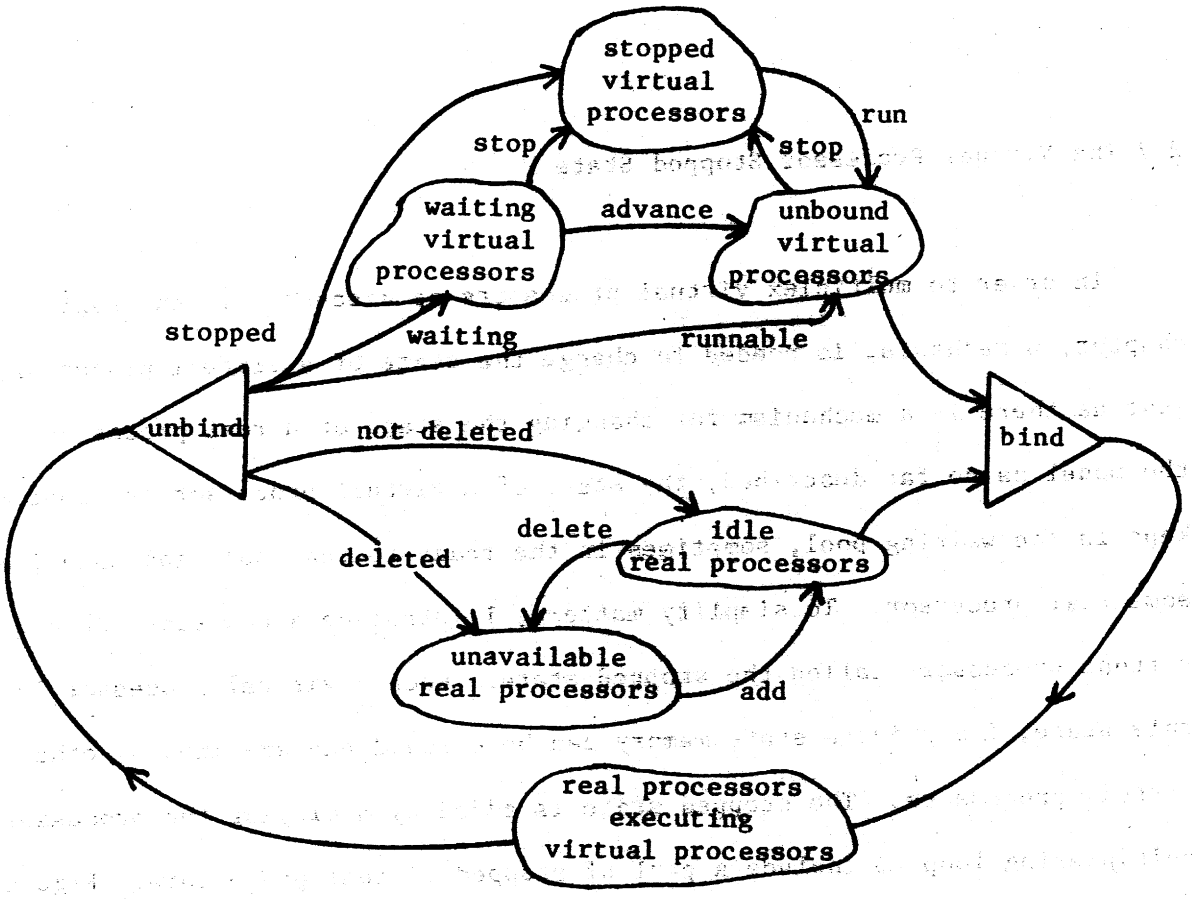


Figure 2.6
Processor Multiplexing Loop with Stopped State

A virtual processor in the stopped state can be started again when another virtual processor executes a start operation specifying the stopped virtual processor. The start operation puts the virtual processor in the ready-to-run pool.

One special point should be made here about the await operation -- the virtual processor private memory while a virtual processor is in the waiting pool looks as if the await operation has not commenced. Thus stopping a

waiting virtual processor, and restarting it later, will cause the await to be re-executed. Since the await operation is a pure predicate, with no side-effects, re-execution cannot cause any problems. Re-execution is chosen in order to avoid having to show in the state of a virtual processor that is in the stopped state which eventcounts are being awaited. The awaited eventcounts are forgotten in the transition from waiting to stopped. For consistency, the advance operation will cause re-execution of the await operation, also.

2.8 Summary

In this chapter, a number of terms are defined, and a model of processor multiplexing is developed. This model will be extended in chapter 3 to a two level processor multiplexing structure. Several important features are incorporated in the model. The model applies to:

1. Systems having multiple real processors, with small private memory for state, and a large shared memory with address mapping hardware to restrict the environment.
2. Systems where processors can share access environments.
3. Systems that allow reconfiguration of physical processors.
4. Systems that allow either centralized or distributed control of processor multiplexing.
5. Systems that allow the scheduling policy to be altered independently of the the rest of the operating system.
6. Systems in which the states of virtual processors are altered by a second level of processor multiplexing.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Chapter Three

Multiple Levels of Processor Multiplexing in a Layered System

In this chapter I explore what it means to do processor multiplexing at two levels, creating two kinds of virtual processors. To start, processor multiplexing is described in terms of a common pattern of type extension, cache management, that applies to operating systems structured according to abstract types. This pattern, and the model developed in chapter two, are then extended to handle two levels of processor multiplexing.

Having thus described the structure of the interfaces and implementations of each level of processor multiplexing, I then show how this structure helps simplify the structure of the operating system. I discuss how the mutual dependency between virtual memory implementation and virtual processor implementation is eliminated. I also indicate how the level 1 processors can be used to execute "kernel processes" that provide processing power to abstract type managers that are part of the kernel of the operating system.

To close the chapter, I discuss three problems that arise from the two level structure and appropriate methods to solve them in the context of a real computer system. The first problem is that inefficiency can be caused by multiple levels of scheduling algorithms. The second problem is that processor multiplexing can interfere with intermediate states of abstract type managers, violating the hierarchic dependency structure. The third problem is

that a mechanism for coordinating the activities of different levels of virtual processors is needed.

3.1 The Cache Management Pattern of Type Extension

Frequently the basic task performed by a higher level type manager in implementing its type out of lower level types is cache management. Janson [11] has described the basic issues of cache management in a virtual memory system based on abstract types. The cache management pattern is ubiquitous in his design.

The cache management pattern involves creating a new abstract type that is represented in terms of two existing types, the cache type and the encached type. The new type created is quite similar to the cache type in functionality. There are usually a limited supply of objects of cache type, so they are multiplexed among the objects of the new type. The encached type generally serves the function of providing a relatively large amount of storage for holding the state of objects of the new type.

For example, see figure 3.1, showing the type-managers for blocks of primary memory (coreblock), records on secondary storage (diskblock), and pages of virtual memory. Here, the major function of the page type manager is to manage the coreblocks available to it as a cache for the information in diskblocks. The only operations on diskblocks are read-block, which

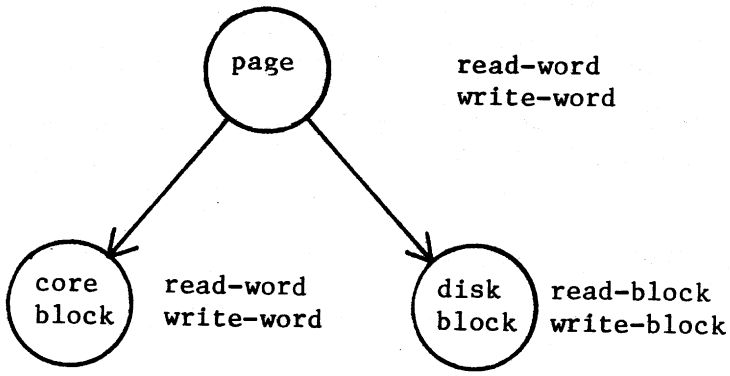


Figure 3.1
Cache Mgmt. Pattern for Page Object

reads the contents of a whole diskblock, and write-block, which replaces the contents of a whole diskblock. The coreblock has more fine-grained operations which allow selective reading and writing of words of the coreblock.

Since the page manager implements fine-grained read and write operations on the page, the most effective way to achieve these is to implement the page as a coreblock. On the other hand, there are more pages than coreblocks, so they must be permanently stored in diskblocks. The fine-grained operations can be achieved by copying the information of a page into a coreblock, where the operation is performed. At some later time, the information in the coreblock can be copied back to disk.

Processor multiplexing can be viewed as just such a cache management algorithm. Given a group of real processors and a set of memory blocks that can hold processor states, a new abstract type can be implemented, called a virtual processor. Real processors are viewed here as objects implemented by

a real processor type manager. The operations permitted on a processor consist of loading a state into it (binding) and running it, and stopping it and storing its state (unbinding). The virtual processor type manager provides four operations, bind, run, stop and unbind, that are similar in effect to the two real processor control operations. The virtual processor has the bind and run operations, and the stop and unbind operations, decoupled for simplicity. The stop and run operations affect the use of real processors in implementing the virtual processors, while the bind and unbind operations affect the processor states in storage only.

Another difference between virtual processors and real processors, however, is that virtual processors interpret the instructions encountered during the run operation somewhat differently. For example, there is an instruction recognized by the virtual processor to mean await some eventcount. No corresponding instruction exists in the real processor -- await is implemented by a sequence of instructions on the real processor that has the properties of an instruction to the virtual processor (once started, it is completed, and no intermediate states can be observed by virtual processors).

The virtual processor type manager has a very simple task -- it just treats the real processor type objects as caches for processor-states. Figure 3.2 shows this structure. The virtual processor manager's bind operation is performed by writing the state of the virtual processor in a memory block called a processor-state. The virtual processor manager unbind operation is performed by reading the value in a processor-state object. (It is an error

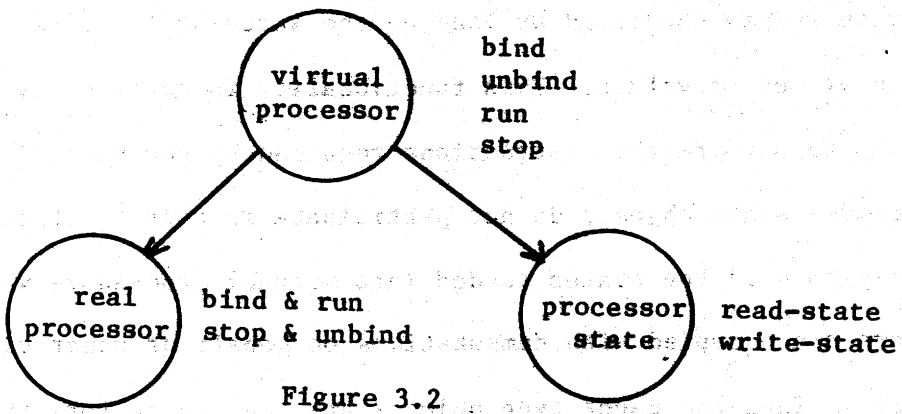


Figure 3.2
Cache Mgmt. Pattern for Virtual Processor

if unbind is attempted when the virtual processor is not stopped.) The stop operation ensures that the virtual processor state is not being interpreted by a real processor. The run operation enables the contents of a processor-state to be bound to a real processor and run, using the real processor bind-and-run operation.

The processor-state objects are very limited in the set of operations that may be performed on them. Only read and write operations are performed by the virtual processor manager. On the other hand, the virtual processor manager uses the real processor to execute the state, once the state is bound to a real processor. This situation emphasizes strongly the different roles played by the cache and encached types in a type defined by a cache manager. In the storage system example previously described, both the coreblock and diskblock are quite similar -- both are passive storage containers, with read and write operations defining their basic capabilities. The virtual processor type manager provides, as its primary function, an interpreter for an

instruction stream specified by loading the state of a virtual processor with a particular set of values. This functionality is obtained by using real processors to perform the instructions required by the virtual processors. The processor-state objects do not participate in this function; instead they serve only to hold the states loaded into virtual processors while the real processors are occupied with computations on behalf of other virtual processors. Thus the cache type objects are used to perform the primary function, and are quite similar in capability to the type implemented by the cache manager, while the encached type objects serve only as storage.

3.2 Building Two Levels of Virtual Processors

As shown in the previous section, processor multiplexing may be seen as providing a new abstract type of processor, by managing the real processor type of objects as a cache for processor states, which are stored in processor-state objects while not actually being manipulated by a processor object. The set of virtual processors produced by processor multiplexing in this way also can be multiplexed to produce yet another new abstract type of processor. (1) The solid arrows in figure 3.3 show how the resulting type hierarchy would look, for two levels of processor multiplexing. The basic algorithm performed by each level in this hierarchy is similar, with the only

---

(1) These can in turn be multiplexed, and the pattern can be carried out repeatedly, yielding a hierarchy of abstract types all of which perform a processor function.

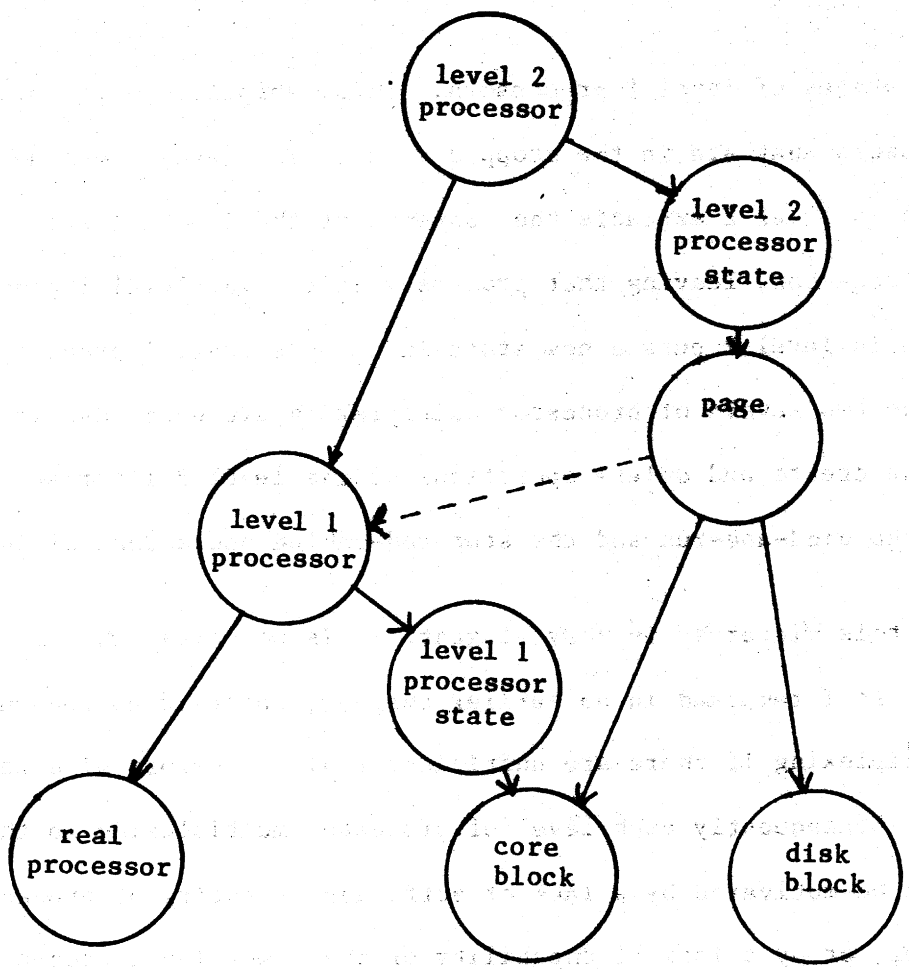


Figure 3.3
Two Level Processor Hierarchy

difference being the type of objects that play the role of cache objects and encached objects.

The model of processor multiplexing developed in the last chapter can be extended to show how the two levels of processor multiplexing fit together. Just as the bind-and-run and stop-and-unbind operations used in the first level of processor multiplexing change the internal memory of real processors, so the second level of processor multiplexing uses bind and unbind operations

to change the states of level 1 processors. This manipulation is done on level 1 processors that are in the stopped state. The level 1 unbind operation used in level 2 extracts the contents of the internal state memory of a level 1 processor, leaving that processor idle. The level 1 bind operation used in level 2 puts a new state in an idle level 1 processor. In figure 3.4, the two levels of processor multiplexing are exact duplications of the model. The create and delete operations of the level 2 interface are analogous to the bind-and-run and the stop-and-unbind operations of level 1.

Although this hierarchy is very elegant, it is not clear whether or not it is useful. As I remarked in an earlier chapter, there is no reason to use processor multiplexing if there are sufficient real processors with the right capabilities. Consequently each level of processor multiplexing in the hierarchy must be motivated by a lack of sufficient quantity of processors at the lower level, or by a lack of capability of the lower level processors. In this thesis, I propose a design that uses two levels of processor multiplexing to create a processor hierarchy of three levels: real processors, level 1 (virtual) processors, and level 2 (virtual) processors. There are several good reasons for this choice, as opposed to the single level of processor multiplexing usually found in operating systems. The reasons are:

1. It disentangles the interdependence between the implementation of virtual memory objects and virtual processor objects.
2. The utility of structuring the operating system, particularly type managers, as a set of cooperating processes.
3. The distinction between short- and long-term scheduling policy.

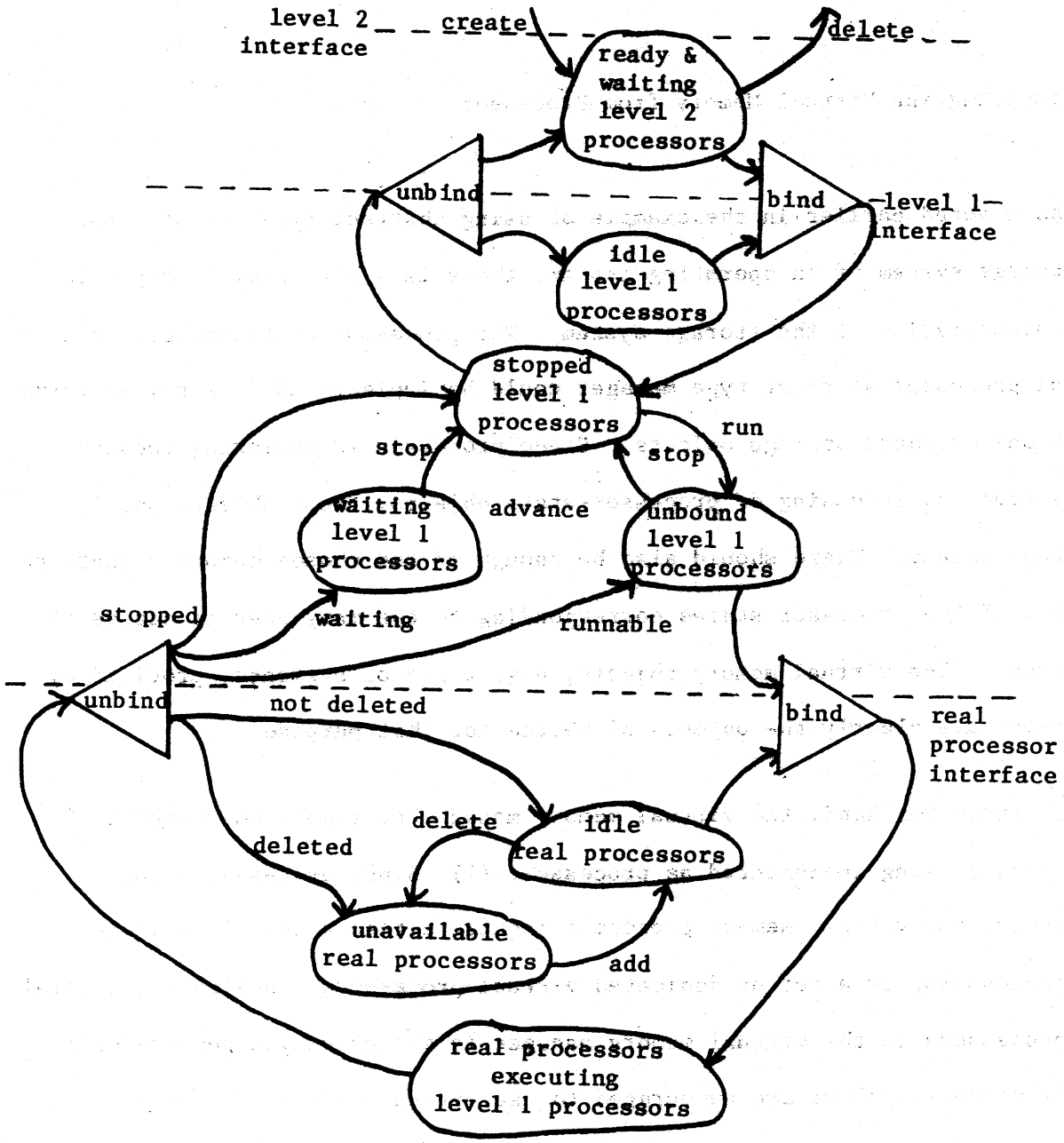


Figure 3.4
Two Level Processor Multiplexing Loop

I will discuss each of these in turn.

3.3 Disentangling Virtual Memory from Processor Multiplexing

As I noted earlier in the example of using abstract types to structure the storage system of an operating system, there is a hierarchy of types in the implementation of the storage system. The processor-state objects of a virtual processor abstract type manager could be implemented directly in terms of any one of these storage objects. Since processor multiplexing requires fairly frequent accessing of processor-state objects, these objects should have fast access. There should also be enough of the chosen memory objects to hold all of the processor states corresponding to the many user processes of the system. The virtual memory objects, e.g. pages or segments, provided by the system are clearly the objects of choice for this purpose.

On the other hand, the virtual memory management algorithms benefit greatly from being implemented as processes. (1) Since processes require processors, the virtual memory processes require either a set of dedicated real processors, or a set of dedicated virtual processors. Dedicating several real processors to the virtual memory manager is excessively expensive with today's hardware, so we are encouraged to use virtual processors implemented by processor multiplexing to achieve the virtual memory management functions.

---

(1) See Huber [10].

Using virtual memory to implement virtual processors and vice versa leads to a system with cyclic dependencies. This can be overcome by splitting the implementation of virtual processors into two stages, where the first implements virtual processors whose processor-states are represented using primary memory objects, and the second stage multiplexes the first stage virtual processors and uses virtual memory objects to hold the processor-states. The virtual memory management processes can then be implemented on first stage virtual processors. This structure has been shown before in figure 3.3. The dotted line indicates the dependency of the page type manager on the level 1 processor type manager, which provides processors to execute page manager algorithms.

3.4 Use of Processes as Abstract Type Managers

Although the common view of an abstract type manager is as a collection of closed subroutines that manipulate a data base, this view is not necessarily the best way to view the implementation of abstract types in a situation where operations can proceed in parallel. With parallel operations, there must be interlocking of some sort between the different operations on objects of the type. This interlocking is not apparent from an implementation of the operations as pure closed subroutines.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Let us consider an example in the context of the example system. There is an abstract type manager whose job it is to multiplex a connection to a message-switched communications system such as the ARPANET [16]. The abstract objects created by the type manager are connections on which operations such as create-connection, destroy-connection, send-message, and receive-message may be performed. The type manager must take the responsibility for sequentializing simultaneous requests on the same connection object. A destroy-connection cannot be allowed to proceed simultaneously with send-message, for example. Since these operations will actually be decomposed into a sequence of operations on lower level objects, such as the buffers, I/O channels, etc., there is a possibility of incorrect operation if the steps of two operations on the same object are interleaved.

One way to prevent such interleaving and achieve sequentiality is to associate a lock with each object, requiring that the lock be set by each operation before any modifications to the object are attempted, and that the lock be reset after the operation is complete. Equivalently, a process can be associated with each object to perform all of the operations on the object by accepting requests for operations that are placed in a queue. The important thing here is that two operations on an object are never performed overlapping in time. This tactic is not sufficient, however, if operations on one object can interfere with operations on other objects. An ever-present example of this kind found in operating systems is the need to manage a small set of resources that are multiplexed among different objects of a particular

abstract type. In the example system, assume that a fixed amount of memory resources is available to the connection type manager for use as I/O buffers. When a send-message is executed, a buffer must be allocated to hold the message while it is being accessed by the I/O device. Other send-message operations on different connections may be attempted simultaneously, resulting in possible interference between buffer allocation operations. In general, operations on different objects implemented by a type manager that multiplexes some lower level resource may need to be sequentialized. For this reason, viewing objects as individual sequential processes is not very useful in solving all of the problems of objects in the presence of parallelism.

Another possible view is looking at the operations performed on all objects in the class implemented by a type-manager as a sequential process, so that no two operations on objects in the class can be performed in parallel. This view actually can be realized in an implementation of an abstract type manager by building the manager as a process, with requests for operations being sent to it through a queue. (1) In the example above, the connection manager would be implemented as a process that performed the actual I/O operations and buffer management. The obvious disadvantage of this view is that it sequentializes operations on different objects even when this constraint is unnecessary. (2)

---

(1) Or alternatively, by using a single lock to protect all operations of the type manager.

(2) Unnecessary sequentialization can be especially bad if an operation on a particular object can take arbitrarily long to complete, or may never complete. In operating systems, however, operations are usually short, and must complete.

The sequentiality can be reduced, while retaining the ability to sequentialize operations on different objects, by building the type manager as a collection of cooperating processes. There will be a single process which accepts requests for operations and then causes the other processes in the type manager to carry out the operations in as parallel a fashion as possible under the constraint of correct operation. This view can be applied to the operation of the page abstract type manager as has been done by Huber [10]. In his implementation, there is one process (represented by the page table lock) which accepts requests in a sequential order. It then causes other processes to carry out operations required by the requests in a parallel fashion.

As noted above, it is possible to implement the sequential processes required to construct such an abstract type manager in two ways. A server process can always be simulated by code that is executed in each requesting process under a lock. As long as the locking convention is obeyed, there is no interference between operations performed under the lock due to parallel execution. Alternatively, the server process can actually be implemented on a dedicated processor of its own.

Use of a lock to create a process can reduce the clarity of the code and create problems that are not found in the process executing on a dedicated processor. An operation that takes place in the requesting process is not easy to protect from the peculiarities of the requesting process environment. For example, the requesting process may not have sufficient scheduling

priority to complete the operation quickly, resulting in delay to other processes waiting to perform successive operations. The meaning of the instructions and addresses in each requesting process may vary, so that the operation must be specially coded to successfully operate in environments where the handling of overflow faults may vary, for example. In addition, each operation must be examined to ensure its termination, for non-termination of one operation can cause all other operations being carried out under the lock not to terminate. If the operations are distributed through the system, it is much more difficult to bring all operations together to inspect them for termination. It is also less likely that a programmer implementing the abstract type will be able to oversee all the operations to ensure termination.

These arguments suggest it is often much simpler to construct abstract type managers as processes that execute on their own processors.

In order to use processes for implementing abstract type managers, it is necessary to have enough processors to implement all of the processes. Sufficent processors can be produced by multiplexing. At each level in the operating system type hierarchy, there must be sufficient processors available for each type manager implemented at that level. The issues of using processes in implementing the storage system generalize to the case of other type managers in the system. There must be a low-level type of processor to implement processes for low level type managers. Higher level type managers will benefit from the additional quantity and capabilities of higher level processors.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Many abstract type managers should be implemented on lower-level processor abstractions in order to guarantee more complete control over the hardware. In the example system, the connection type manager may need to be scheduled rapidly when a message arrives, in order to get that message to the receiving process promptly if necessary. If such a process were implemented on too high a level, it would be delayed in its response by the cost of several levels of scheduling by different processor multiplexing algorithms. Consequently, it should be implemented on a relatively low level processor.

In a system with two levels of processor multiplexing, most of the abstract type managers for system objects will be built out of the first level of virtual processors for this reason.

The type manager processes inside the operating system must always be capable of servicing requests, if it is required that the system not deny service to users. For this reason, it should be impossible for the type manager processes to be put into a state that will ignore requests for service forever. Thus, the abstract type manager process must always have a processor. Further, such abstract type manager processors must always have priority for physical processor resources over all user computations.

Consider the example system. If the processors on which the page abstract type manager is implemented had lower priority than user computations, user processes that did not require service by the page manager could effectively deny service to user processes that did require service by

the page manager. By saturating the real physical processor resources, user computations could prevent the page manager from running for arbitrary periods of time.

Abstract type managers implemented on virtual processors provided by the first level of processor multiplexing should not be affected by the second level of processor multiplexing that implements user virtual processors. There are two reasons for this. First, the second level of processor multiplexing, which depends on abstract type managers implemented on virtual processors, cannot be allowed to manipulate the virtual processors of those type managers. This would lead to a cyclic dependency where the type manager process depended on the second level processor multiplexing algorithm that depends on the type manager. Second, the type managers of the operating system must be guaranteed service ahead of the user computations scheduled by the second level processor manager.

A mechanism whereby a process executing on a virtual processor can attach itself firmly to its virtual processor is required, so that it cannot be removed from the virtual processor by the second level processor multiplexing manager. In addition, virtual processors executing abstract type managers inside the operating system must have priority for computational resources over the virtual processors executing user computations.

Looking back to figure 3.3, let me emphasize these points. The level 1 processors implemented by the level 1 processor type manager are used in two

ways. Some of them are multiplexed by the level 2 processor type manager to make level 2 processors. Some others are used to used in implementing the system type managers, such as the page type manager, the connection type manager, and the level 2 processor manager itself, to perform various management functions, isolating and sequentializing the system type manager operations. These latter level 1 processors are permanently bound to the processes of the page manager. They also have scheduling priority over those level 1 processors used to implement level 2 processors. The resulting

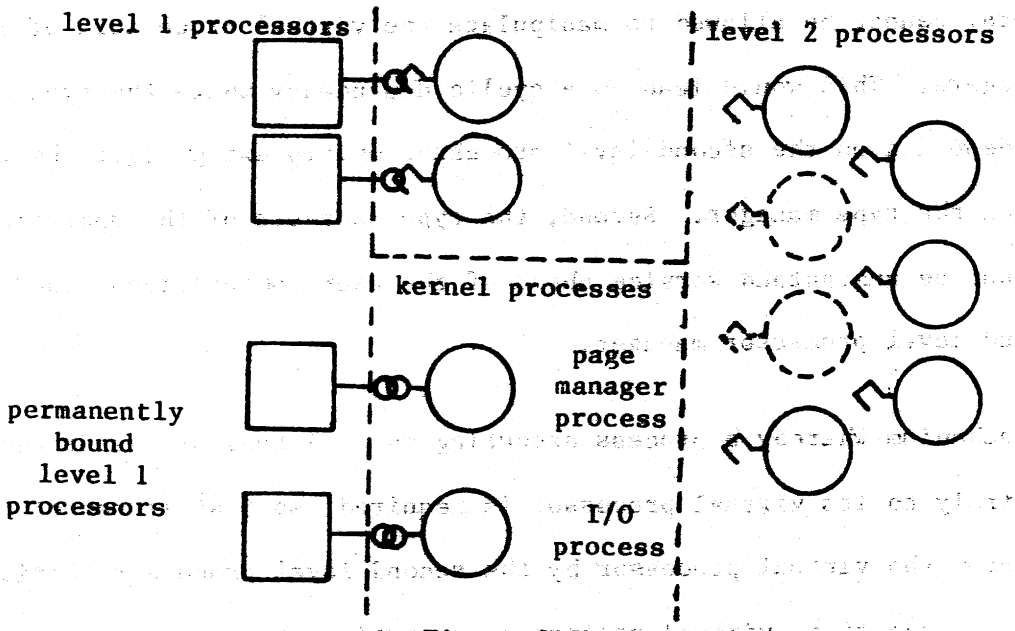


Figure 3.5
Permanently Bound Type Manager Processes

structure is shown in figure 3.5.

3.5 Two Levels of Scheduling

There is a natural hierarchy in scheduling policy that is found in many operating systems. In Multics, for example, there is a short-term multiprogramming policy that multiplexes processors among a small number of user computations. The goal of this algorithm is to achieve maximum use of the processors, and thus maximum throughput in the short-term. Multics also incorporates a long-term scheduling policy that controls the set of user computations that participate in short-term multiprogramming. The goal of the long term policy is to achieve control of the responsiveness of the system.

The scheduling hierarchy is easily incorporated into the two level virtual processor hierarchy. The first level of processor multiplexing provides level 1 processors that have a built-in short-term scheduling policy that is designed to maximize throughput. The second level then provides level 2 processors that have an administratively variable scheduling policy that is designed to control the responsiveness of the system for each class of users.

3.6 Problems of a Processor Hierarchy

Having mentioned the advantages of a processor hierarchy, I will now describe the potential disadvantages of the hierarchy. There are three such problems. They are inefficiency due to multiple levels of processor multiplexing, potential interference by the level 2 scheduler in the internal workings of a type manager at a lower level, and the need for IPCC between processes implemented at different levels in the hierarchy.

3.6.1 Efficiency of Multiple Levels of Scheduling

Having two levels of scheduling going on at one time can be very costly in terms of scheduling overhead. For example, if the frequency of scheduling decisions at the second level were the same as the frequency of scheduling decisions at the first level, and each scheduling decision had a fixed overhead cost in processor time, then the total amount of processor time wasted in scheduling decisions would be twice that of a single level scheduler.

Extra scheduler overhead is not a problem with the two level scheduler, however. The reason is that the scheduling policy implemented at the second level makes long-term decisions. Thus the second level decisions are made far

less frequently than the short-term multiprogramming decisions made at the first level. Consequently, the overhead of scheduling at the second level will be insignificant compared to the overhead of the scheduling decisions at the first level, assuming that decisions at the second level cost the same or less than decisions at the first level. Furthermore, most of the work done by each level would have to be done in a single level, anyway. Extra overhead only arises if the second level duplicates the effort of the first, so that the same work is done twice, or if the interface through which the second level controls the first is more costly than that which can be achieved in a single level design. The short- versus long-term distinction eliminates duplication of effort. The interface overhead problem is mitigated by the low frequency of interactions between the first and second levels relative to the frequency of interaction between the first level and the real processor level.

Although the second level of scheduling does increase the time overhead of processor multiplexing slightly, another cost is actually reduced by introducing the second level. This cost is the cost of memory to hold processor states. At the first level, primary memory must be used. (1) At the second level, cheaper virtual storage can be used instead of primary memory.

---

(1) The major use of primary memory in the level 1 implementation is to hold environment descriptions. Only level 1 processors that are in use (i.e., not stopped) need have their environment descriptions in primary memory. Level 2 is responsible for ensuring that the environment descriptions are in primary memory.

3.6.2 Protection of Low-level Type Managers from Level 2

Consider the operations of the page type manager, whose position in the system type hierarchy is shown in figure 3.3. Operations provided by the page manager are used by both the level 2 processor implementation and the level 2 processors that execute user computations, since both use pages for holding their data bases. Some of the operations on pages manipulated by level 2 processors can be implemented as subroutines or in-line code (1) that can be executed by level 2 processors while bound to level 1 processors. If the designer of the system is not careful, it may be possible for a level 2 processor to become unbound from its level 1 processor in the middle of executing the sequence of instructions that implement a page operation.

Having started executing an operation of a level below the level 2 processor implementation, the process must be allowed to finish that operation before it can be unbound from the level 1 processor. If it were prevented from finishing, two problems might occur. First, the level 2 processor manager could modify the private memory (e.g., the instruction pointer) of the

---

(1) The expansion into subroutines or in-line code of the type manager operations should, of course, be transparent to the user of the system -- he should not know that type manager operations are actually sequences of lower-level instructions. Presumably, the user will be prevented from actually writing code to manipulate the type manager data bases by a run-time or compile-time protection mechanism.

level 2 processor, and then rebind the level 2 processor to a level 1 processor. This modification would interfere with the subsequent correct operation of the type manager. Second, the level 2 processor manager could prevent the operation from ever completing, thus leaving the data bases of the manager in a possibly inconsistent state (e.g., it might have a lock set in it). Both of these problems violate the hierarchic structure of the system, since they can cause type managers at lower levels to depend on the level 2 processor manager for correctness.

Allowing the level 2 processor manager to unbind a level 2 processor in the middle of a lower level operation can lead to deadlock of the level 2 processor manager, as well. The deadlock can arise because the data bases being manipulated by the interrupted abstract operation are used in the implementation of the level 2 processor manager. For example, the interrupted page manager operation may have set a lock on some part of its internal data bases to prevent parallel manipulation of those data bases by other processes. The level 2 processor manager, when it handles the unbinding of the level 2 processor that is stopped, may call upon the page manager to obtain information about the level 2 processor for rescheduling. The request of the level 2 processor manager will be forced to wait until the level 2 process being rescheduled finishes the current operation, since the lock is set by the level 2 process. The process cannot finish its operation until it is rescheduled, therefore there is deadlock.

To prevent violations of the type hierarchy and deadlocks, operations of type managers at lower levels than the level 2 processor manager should appear to be indivisible to the level 2 processor manager. The level 2 processor manager will only be able to unbind a process from the level 1 processor before or after, but not during an abstract type operation.

In the design, this indivisibility is achieved by having abstract type managers inform the level 1 processor manager when they start and finish indivisible operations. Between the start and finish of indivisible operations, the level 1 processor manager will not allow the level 1 processor to enter the stopped state. Since level 2 can only inspect and alter the states of stopped level 1 processors, the desired indivisibility is achieved.

A very simple method for deciding when an operation should be indivisible at level 1 arises from the hierarchy. All operations of type managers below the level 2 interface in the type hierarchy should be indivisible. If a type manager is below level 2, level 2 uses it and depends on its correctness. It is a violation of the abstract type model for level 2 to be able to interfere with the operations of types that it depends on.

3.6.3 Cross-level Interprocess Control Communication

Each level of processor provides its own mechanism for communicating between computations running on those processors. It will occasionally be

necessary to design the system so that a computation expressed in terms of level 1 processor operations (such as the page type manager) can signal a computation expressed in terms of level 2 processor operations, or vice versa.

Consider the example system of figure 3.3. If the page manager were implemented as a process permanently bound to a level 1 processor, then level 2 processors requesting the services of the page manager would have to signal the page manager somehow, and be signalled back when the request is finished. The level 1 page manager processor cannot use the IPCC primitives implemented in the level 2 processor type manager, because the level 2 processor manager depends on the page manager for various services, such as implementing its tables and moving the environment descriptions of level 2 processors in and out of primary memory. A cyclic dependency would result if the page manager processor attempted to use the level 2 processor IPCC primitives. On the other hand, the level 2 processor requesting service must be able to await at level 2 if the level 2 scheduler is to retain control over the resource usage by level 1 processors. In this case, then, a level 2 advance by the level 2 requesting processor needs to awaken the page manager processor that awaits at level 1 (an inward signal), and later a level 1 advance by the page manager processor needs to awaken the requesting level 2 processor that awaits at level 2 (an outward signal).

What is required in general is a way to perform an advance operation at one level that causes await operations in progress at the other level to proceed, just as if the advance were done at that level. I now present the

algorithm for level 2 advance and await, and then discuss how inward and outward signalling are implemented.

3.6.3.1 Level 2 Advance and Await Algorithms

The algorithm for await at level 2, in terms of level 1 await, is:

1. mark current level 2 processor as awaiting the named events.
2. do a level 1 processor await on the same eventcounts. (1)

The algorithm for level 2 advance is:

1. do a level 1 advance on the specified eventcount.
2. mark as not waiting, any level 2 processors whose eventcounts included the one advanced. This will cause them to become assigned to level 1 processors (if they are not already so bound), where they will discover that the current await immediately proceeds.

It is absolutely necessary to have the computation re-execute the await instruction at level 1 whenever a level 2 processor that was awaiting at level 1 is reassigned to a new level 1 processor by the level 2 processor abstract type manager. Re-executing the await guarantees that step 2 of the advance algorithm works properly.

---

(1) In chapter six, I will show that the level 1 await here need not be on the same eventcounts. I have simplified the algorithm here because the added complexity discussed in chapter six is irrelevant to the outward signalling mechanism.

3.6.3.2 Inward Signalling

Inward signalling, an advance at level 2 starting processors that are awaiting at level 1, works correctly in the level 2 advance algorithm above. Level 2 eventcounts are implemented in terms of level 1 eventcounts, so that an advance at level 2 is performed by an advance at level 1 plus some bookkeeping to handle processors awaiting at level 2.

3.6.3.3 Outward Signalling

Outward signalling, an advance at level 1 starting a processor that is awaiting at level 2, is more difficult than inward signalling. While an await at level 2 is performed by invoking await at level 1, it is possible for the processor awaiting at level 1 to become unbound from its level 1 processor, so that it is now waiting only at level 2.

Unbinding from level 1 is <u>possible</u> for await operations that need not be a part of a level 2 atomic operation. For example, when a level 2 processor is waiting for a page to be brought into primary memory it can be unbound from level 1 since the correct operation of the system does not depend on the level 2 processor to actually reference the page after it is brought in.

Unbinding a level 2 processor while it is awaiting at level 1 is desirable for an economic reason. The real processors of the system may not be used to full capacity if level 1 processors are all awaiting events. Since there will be relatively few level 1 processors (since level 1 processors take up large amounts of expensive primary memory), if it is possible to unbind waiting level 2 processors, it is economically advantageous to do so. Short waits are not as much of a problem as long waits.

Basically, the difficulty of outward signalling is that the level 1 processor advance primitive cannot know all of the processors awaiting at level 2 that are to be awakened when an eventcount is advanced. If the full economic advantage of unbinding level 2 processors awaiting level 1 advances is to be obtained, the level 2 processor manager should not rebind a waiting level 2 processor to level 1 before it will be able to proceed through the await. Thus, the level 2 processor manager must be aware of advances to eventcounts that are done at level 1 with the intention of signalling processors at level 2.

Detection is not easy, since all eventcounts are potential channels for outward signalling. The task may be restricted since in any particular system only a few eventcounts will be used for outward signalling. In the example system, there will be a fixed set of eventcounts that are signalled by each kernel type manager -- the page manager will have a small set of events that it signals, and so will each other type manager in the operating system. By structuring the system so that the level 2 processor manager knows this set,

and can efficiently search it for modified eventcounts, we can solve the outward signalling problem.

The level 2 await primitive recognizes eventcounts that can be outward signalled because they are all stored in the same segment. This is a simple way to design the system so that the level 2 manager need not be changed every time the set of eventcounts signalled outward by lower level type managers is changed. Eventcounts in this segment will be treated specially by the level 2 processor await primitive -- the level 2 processor manager will periodically poll the value of these eventcounts to see if they have changed.

How frequently the level 2 processor manager checks will determine the responsiveness of the user processes to outward signalled events. The checking can be triggered by a real-time clock ticking at a certain rate (chosen for the desired responsiveness). Alternatively, the checking can be done every time an eventcount in the outward signalling eventcount segment is advanced in order to ensure maximum responsiveness. This latter strategy requires a small amount of help from the level 1 processor manager, in the form of a special eventcount that is advanced by level 1 every time any outward signalling eventcount is advanced by the level 1 advance operation. The level 2 processor manager (which is permanently bound to a level 1 processor) can then await this special eventcount.

3.7 Summary

In this chapter, I have shown how two levels of processor multiplexing can work together. The model developed in this chapter, and the solutions to the three problems discussed, will be used in chapters five and six as a basis for a detailed design of a system where two level processor multiplexing is used.

Chapter Four

Level 1 Virtual Processor Interfaces

In this chapter, we begin discussion of a proposed operating system design that incorporates two levels of processor multiplexing, as in our model. Here we discuss the interface of level 1 virtual processors.

The description of level 1 is divided into two chapters. This chapter describes and motivates the interface of the level 1 processor. Incorporated into this interface are many features that are important in a real system such as Multics. Examples from the Multics system are used to motivate the design. Chapter five describes an implementation of the level 1 virtual processor manager.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

4.1 Level 1 Virtual Processor Interface

Level 1 processors are quite similar to real physical processors. They execute instructions in basically the same way, have similar internal states, and have the same address mapping to address primary memory. There are some differences from hardware processors, though. They can execute several new operations that are implemented by the level 1 processor manager. Their rate of execution is controlled by the level 1 processor manager. They cannot be added to or deleted from the system. We describe here those differences.

The operations that the level 1 processor can perform that cannot be performed by real processors serve four different purposes. Some of the operations allow level 1 processors to do interprocess control communication. Some of the operations allow level 1 processors to control the bindings of level 2 processors to other level 1 processors. These operations are structured so that the level 2 processor manager may be built as a central agent out of several dedicated level 1 processors. Some of the operations are concerned with virtualizing the hardware facilities of real processors, such as fault handling. Finally, there are operations to change the hardware resources being used by level 1, to allow for reconfiguration.

To facilitate description, operations of the level 1 processor are described as if they were subroutine calls. The names of each operation will consist of the prefix "VP1$" to indicate that it is an operation of the level 1 virtual processor manager. The data input and output from the operation are specified by parameters to the call. Parameters that represent input values appear normally, output parameters are underscored. In the actual implementation, these operations all act as if they are non-decomposable machine instructions. It is not possible to stop a level 1 processor during the execution of one of these operations. Also, the level 1 operations must not be interrupted in the middle by a fault. Consequently, each level 1 operation ensures that all of its parameters are in primary memory and accessible to the level 1 processor before performing the required operations. If the parameters are not in primary memory, a fault will be reflected to the level 1 processor. The level 1 processor can then handle the fault, and restart the operation from the beginning. Accessing of parameters is discussed more fully later in the chapter.

There are certain operations that are used only by the second level processor multiplexor. These operations are specially protected, so that only the level 1 processors that are used to implement the level 2 processor manager may execute them. Protected operations will be marked in the text by an asterisk following the parameter list when their calling sequence is described.

In any case, the level 1 operations are all internal to the kernel of the operating system, and can be used only by programs written as part of the kernel of the operating system.

4.2 Limited Supply of Level 1 Processors

The level 1 processor manager creates virtual processors that perform computations for higher levels in the system. There is a fixed, small number of level 1 processors in the system. The limitation on the number of processors arises because level 1 processors are implemented at the lowest level of the system. The level 1 processor states and environments are stored in primary memory. Since primary memory is expensive and of limited supply, the number of distinct level 1 processors that can be implemented is limited. The actual number of level 1 processors created in an implementation will depend on the available memory, and the need for level 1 processors at higher levels of the system. For a Multics configuration such as the one installed at M.I.T., with two processors and 384K words of primary memory, I estimate that about fifteen or twenty level 1 processors will be sufficient. This estimate is based on two facts. The number of processes actually participating in multiprogramming at any one time in the M.I.T. Multics never exceeds six. Six level 1 processors can thus be allocated to the second level processor multiplexor to implement user processes. The remaining nine to fourteen are allocated to executing kernel processes that manage various kernel resources such as virtual memory, multiplexed I/O devices, etc.

4.3 Multiprogramming of Real Processors Among Level 1 Processors

Unlike physical processors, level 1 processors do not execute instructions at a constant rate (due to the fact that they are implemented by processor multiplexing). In order to provide kernel processes with quick response to events, level 1 processors have fixed priorities for computing resources. Kernel processes that need fast response, such as I/O device service processes, will be bound to high priority level 1 processors. User processes will always be bound to level 1 processors of the lowest priority.

The simplest way to discuss the effect of priorities is to describe the effect of the priority mechanism on the assignment of real processors to level 1 processors. Real processors will always be assigned to the highest priority runnable (1) level 1 processors. If two level 1 processors have equal priority values, the one that has been computing the longest will have priority. This implies that scheduling of processors of equal priority will be approximately FIFO. It has been the experience in Multics that FIFO scheduling during short-term multiprogramming was the most effective means of achieving good throughput and avoiding thrashing. This choice of policy implements that experience.

---

(1) By runnable, we mean non-waiting and non-stopped.

4.4 Execution States of Level 1 Processors

From outside the level 1 processor implementation, a level 1 processor is either executing (running or waiting) or stopped. Without observing the side effects of execution, such as changes to shared memory, it is not possible to tell whether an executing level 1 processor is actually executing on a real processor or not. As we have shown in chapters two and three, the stopped state of a level 1 processor exists to allow changing the binding of the processor safely.

The level 2 processor manager must change the execution state of level 1 processors in order to multiplex them. Since the level 2 processor manager will be constructed out of level 1 processors, the level 1 processor manager must provide operations that allow one level 1 processor to change the execution state of another. There are two such operations.

VP1$run (l1proc) *

changes the state of the level 1 processor named l1proc from stopped to executing. If l1proc is already executing, the operation has no effect.

VP1$stop (l1proc) *

causes the level 1 processor named l1proc to stop as soon as possible. If the level 1 processor is already stopped, the operation has no effect.

The binding of a level 1 processor may only be changed when it is in the stopped state. A level 1 processor only enters the stopped state in between atomic operations. So that operations on system objects can be implemented on level 1 processors as atomic operations, a facility is provided that allows a sequence of instructions to be treated as an atomic operation. Executing the operation

VP1$begin_atomic_operation ()

indicates that an atomic operation is to be begun. Once VP1$begin_atomic_operation is executed, the level 1 processor cannot enter the stopped state. The operation

VP1$end_atomic_operation ()

ends the current atomic operation. Atomic operations may be nested in time; the level 1 processor can only be stopped after the final call on

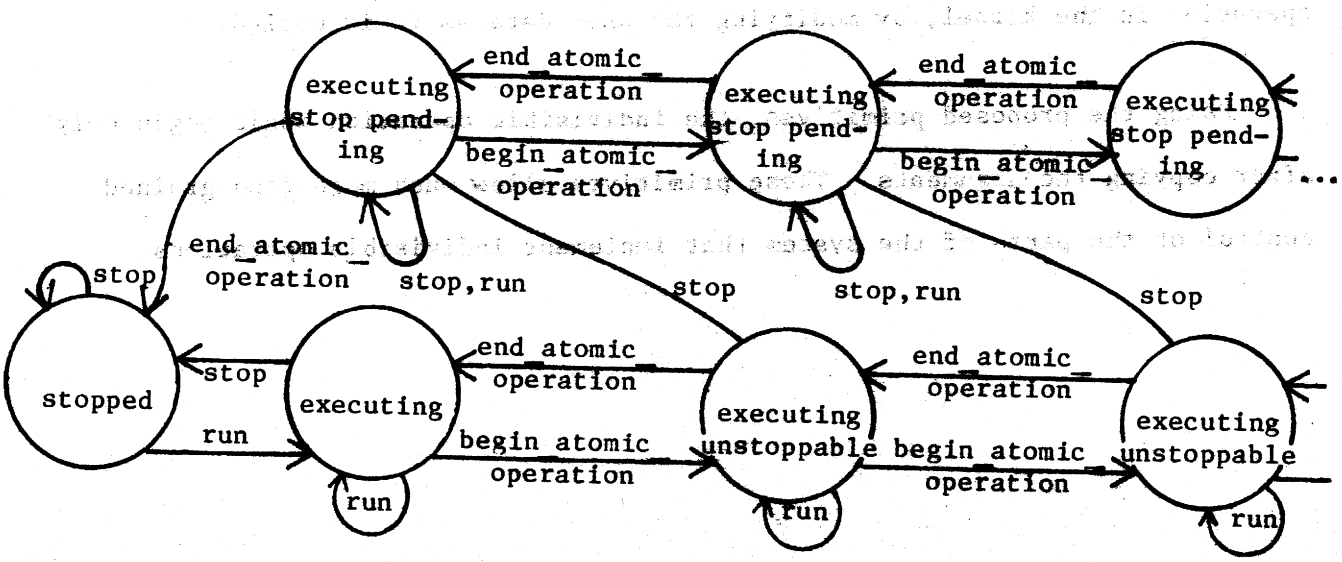


Figure 4.1
States of Level 1 Processor

VP1$end_atomic_operation. Figure 4.1 shows how the actual execution state changes in response to state changing operations.

The operations VP1$begin_atomic_operation and VP1$end_atomic_operation are similar to a facility already existing in the Multics operating system. The Multics mechanism for assuring that virtual processors executing system code do not get pre-empted in the middle of a system operation is to mask the physical processor from getting timer runouts or pre-empt interrupts while executing in the supervisor domain. The Multics mechanism is flawed, however, because some code executed in the system domain is not part of any kernel abstract operation. A particular example is the copying of argument values into the kernel domain from the user domain. The copying is done by code executing in the kernel domain, but accessing user data structures. It is possible to put the processor into a loop while executing an (indivisible) operation in the kernel, by modifying the user data as it is copied.

Using the proposed primitives, the indivisible operation would begin only after copying the arguments. These primitives allow much more fine-grained control of the parts of the system that implement indivisible operations.

4.5 Scheduling Controls

The level 2 manager will be implemented on level 1 processors. In order to control the amount of real processor time used by the level 1 processors it multiplexes, the level 2 processor manager must be able to stop level 1 processors after they use up a short-term allocation of processor time. This function must be provided by level 1, since level 1 controls the allocation of real processor resources to level 1 processors. Level 1 thus associates with each level 1 processor the accumulated processor time used since VP1$run was called, and a limit on this usage called the quantum. When a level 1 processor exceeds its quantum of processor time, the level 1 processor manager effectively calls VP1$stop on that processor, causing it to stop after the current atomic operation is completed.

Since level 1 processors exceed their quanta independently of the execution of the level 2 processor manager, the level 2 implementation needs some help to know when level 1 processors stop, and which level 2 processors have stopped. Each time a level 1 processor stops, a special eventcount managed by level 1, called the stop eventcount, is advanced. The level 2 processor manager can then await this eventcount to discover when level 1 processors stop. To let the level 2 processor find the stopped level 1 processors easily, the level 1 processor manager maintains a queue of stopped

level 1 processors. When a level 1 processor stops, it enters the queue. A level 1 processor operation,

VP1$next_stopped (llproc) *

returns the name of the next stopped level 1 processor in the queue, deleting it from the queue. The level 2 processor manager can use this operation to find all of the stopped processors.

4.6 Changing the Bindings of Level 1 Processors

The second level processor manager needs to be able to change the bindings of level 1 processors it multiplexes. To provide this function, there are two operations that allow the internal state of stopped level 1 processors to be extracted and loaded. The state description used in these

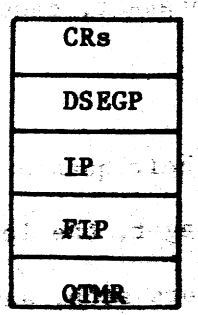


Figure 4.2
Level 1 State Data

interfaces is shown in figure 4.2. The state consists of the values of the computational registers (CRs), the address of an environment specification (DSEGP), the current value of the instruction pointer in the environment (IP),

the address in the environment to which the IP will be set when a fault occurs (FIP), and the amount of resources remaining until the level 1 processor is automatically stopped for exceeding its quantum (QTMR).

The operation

VP1$bind (llproc, state, error) *

sets the state of level 1 processor llproc from its state argument. The operation succeeds, and error is set to false if llproc is stopped, otherwise, the operation fails and error is set to true. A level 1 processor may be unbound by the operation

VP1$unbind (llproc, state, error) *

that returns the new state of the level 1 processor in the variable state. If llproc is stopped, error is set to false and the operation succeeds, else error is set to true, and no data is copied into state.

4.7 Interprocess Control Communication

The level 1 processor manager provides operations to perform interprocess control communication using eventcounts. At this level, eventcounts are implemented simply as primary memory words. In order to allow these eventcounts to be shared among several virtual processors, each of which has a different local name for it in its environment, we need a global name for each memory word. It is possible to use the absolute primary memory address for this purpose. Using the primary memory address would not allow these eventcounts to be managed by the virtual memory manager, though, because the virtual memory manager can move the eventcount from one address to another, or to disk. To allow the virtual memory manager to move the pages containing eventcounts in and out of primary memory freely, the environment description for each level 1 processor contains an additional value for each page of primary memory. This value is the unique name of the page in the virtual memory as a whole. Given the name of a page within the environment of a level 1 processor, the level 1 implementation can determine both its current primary memory address (if in primary memory) and its unique name. Level 1 can use this unique name to name eventcounts in the page, no matter how they move about in primary and secondary memory.

The level 1 processor manager implements the two operations,

VP1$await (ec1, value1, ec2, value2, ec3, value3)

and

VP1$advance (ec).

VP1$await actually allows up to three eventcounts to be awaited simultaneously. It thus takes from 1 to 3 pairs of arguments (3 pairs are shown in the calling sequence). The ec arguments are passed by reference, using pointers in the environment of the caller. The level 1 implementation performs the translation to unique system-wide name. The operation VP1$await only returns to the caller after one of the eventcounts ec1, ec2, or ec3, exceeds the corresponding value specified as value1, value2, or value3. A level 1 processor could simulate the effect of waiting on multiple eventcounts by spawning three separate level 1 processors to wait on each eventcount separately, then waiting for one of them to advance a shared eventcount. Spawning processors this way is cumbersome, so it is useful to allow multiple eventcounts to be awaited simultaneously. The number of eventcounts that can be awaited is limited to three because the level 1 processor implementation can use only a fixed amount of storage to remember the eventcounts being awaited. Three is not a magic number, but seems sufficient for all purposes I have investigated.

Outward signalling eventcounts are supported specially by the VP1$advance operation. Whenever an outward signalling eventcount is advanced, a special eventcount called the outward_signals eventcount is also advanced implicitly.

Outward signalling eventcounts are recognized by the advance operation because they are all implemented in the same virtual memory segment. Thus, by simply checking the unique name of the eventcount, outward signalling eventcounts can be recognized.

4.8 Special Eventcounts

We have already described two special eventcounts that are advanced by the level 1 processor manager itself: the stopped and outward_signals eventcounts. There are two other kinds of special eventcounts that are provided by the level 1 processor interface.

In order to have processes that synchronize themselves in real time, we provide a special eventcount that is advanced proportionally to real time. The clock eventcount is advanced once every delta microseconds, where delta is a reasonably large value, like 50,000. This allows reasonably fine-grained scheduling of processes that have to deal with real time events, such as timeouts on communications channels, etc.

In order to provide for processes that control I/O devices, we need some mechanism for I/O devices to signal processes about interesting events, such as completion of an operation, errors, etc. Messages from I/O devices are stored in special regions of memory called mailboxes, but a mechanism for scheduling processes when interesting events happen is still needed. A very

natural mechanism is to associate with each device mailbox an eventcount that is advanced by the I/O device (with the help of the level 1 processor manager) each time a message is put in the mailbox. A device control process can then simply wait on the eventcount until this advance occurs, then inspect the message.

4.9 Fault Interface

Certain hardware operations signal errors by causing faults. On typical hardware processors, a fault is handled by saving the instruction pointer at the time of the fault and transferring to a special address. In creating level 1 processors, we virtualize fault handling to allow each level 1 processor to specify its own private fault handlers. As part of the state of each level 1 processor, there is a pointer called the fault transfer pointer. Upon encountering a hardware fault, the level 1 processor will save the processor state at the time of the fault, and transfer control to the fault transfer pointer. An operation provided by the level 1 processor manager is used to obtain the processor state at the time of the last fault. This operation is:

VP1$get_fault_data (processor state)

It gets the processor state of the most recent fault. The processor state returned by this operation is shown in figure 4.3. The data of the processor state contains the values of the computational registers at the time of the fault (CRs), the instruction pointer at the time of the fault (IP), and the

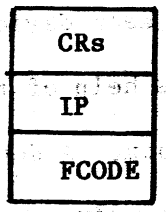


Figure 4.3
Level 1 Fault Data

type of fault (FCODE). The other data of the level 1 processor state, such as DSEGP, QTMR, and FIP, are not kept for faults because the data is constant in the level 1 processor.

Faulting instructions may be restarted by restoring the processor state data using a level 1 processor operation:

VP1$restore_processor_state (processor_state)

If a level 1 processor takes a second fault before extracting the fault data of the first, the level 1 processor manager will crash the system by deconfiguring all of the real processors, so that the problem can be debugged.

In order to allow extending existing processor instructions in type managers above level 1 by providing special fault handlers to increase the effective functionality of instructions, there must be a way for the fault handler to appear to be part of the same atomic operation that caused the fault. For this reason, taking a fault in a level 1 processor implicitly causes a VP1$begin_atomic_operation to be executed. So that it is possible to protect the whole sequence, from faulting instruction to restart of the fault,

the VP1$restore_processor_state operation implicitly executes a VP1$end_atomic_operation. The fault handler need not, of course, remain unstoppable throughout its execution. It can execute VP1$end_atomic_operation in the middle of its execution, as long as it executes VP1$begin_atomic_operation before restoring the state. Such an action must be performed if the fault taken is to be reflected to a program at a level above the second level processor implementation. The fault handler that is specified by FIP in the level 1 processor state must be a program in the kernel of the system below the level 2 processor manager.

4.10 Processor Interrupt

In Multics, there is a mechanism whereby one virtual processor can cause another to take a special fault, called a "process interrupt". This mechanism is used to implement the function of interrupting a computation by hitting the attention key, for example. In order to implement this in level 2, we need a mechanism whereby the level 2 procesor manager can cause a level 1 processor to take a special fault, called the "processor interrupt". We don't wish this interrupt to happen during an atomic operation, or in a kernel process. Consequently, we introduce a mechanism that allows this fault to be set only in a stopped virtual processor. The primitive

VP1$set_processor_interrupt (l1proc, error) *

will cause l1proc to take a special fault when the level 1 processor is next

run. If llproc is not stopped, the operation does not proceed and the error argument is set true, otherwise error is set to false.

To cause a level 1 processor interrupt to occur in a level 1 processor that is not stopped, it must first be stopped, then the processor interrupt must be set, and then the processor must be run. This is a somewhat clumsy interface, since VP1$stop does not take effect immediately. Since the VP1$set_processor_interrupt operation is used only in the level 2 manager, the clumsiness is not a real serious problem. I have chosen this particular interface because it simplifies the design of the level 1 implementation, even though it makes level 2 somewhat more complex.

4.11 Processor Reconfiguration

Level 1 has to deal with reconfiguration of physical processors. It provides three operations for this purpose. The operation

```
VP1$add_cpu (cpu_id)
```

adds the physical processor named cpu_id to the system. The operation

```
VP1$del_cpu (cpu_id)
```

deletes the physical processor named cpu_id from the system. The operation

```
VP1$crash_system ()
```

eliminates all physical processors from the level 1 multiplexor, and forces one of the processors to execute a special debugging program. The other processors are made to stand by idle.

The reconfiguration primitives are accessible to all parts of the system kernel. Outside the system kernel, these operations are not directly usable, in order to prevent user-written programs from denying service to other programs.

4.12 Parameter Passing To Level 1 Processor Operations

All data operated on by level 1 processor operations must be in primary memory. If an object is not in primary memory, the real processor will generate a missing-page or missing segment fault, indicating that the instruction cannot be performed. The software operations of the level 1 processor behave exactly the same. The data provided as parameters to the level 1 processor implementation must be in primary memory. If the data is not in primary memory, the level 1 processor implementation reflects this condition as a software-generated missing page or missing segment fault.

Two other alternatives to generating software "faults" could have been used in the level 1 interface. First, the level 1 manager could crash the system if its parameters were not found in primary memory. With this alternative the level 1 processor invoking the operation would be required to insure that its parameters were in primary memory. For frequently executed level 1 operations, having to wire-down parameters to primary memory by calling the wire-down primitives of virtual memory can be quite expensive. The second alternative would be to reflect an error to the level 1 processor

in some manner other than a fault. Reflecting the error requires some way to transfer information back to the level 1 processor that an error has occurred. The fault mechanism is such a way, inventing another mechanism serves no useful purpose.

The implementation of the level 1 primitives must be able to access the parameters. Since the level 1 processor itself accesses data in memory through a map, the level 1 processor implementation must be able to interpret the map to find the parameters. The map can be modified asynchronously by the processors of the virtual memory manager, so there must be some way to insure that such modifications do not interfere with the correct operation of the level 1 processor manager.

The level 1 processor operations operate logically by first determining whether the parameters are in primary memory. If not, a fault is reflected to the appropriate fault handler, which presumably will handle the fault by moving the parameters into primary memory. The test will be repeated until the parameters are all in primary memory. (1) Then, the parameters are accessed to perform the required operation. The data cannot be moved from primary memory during this accessing. There must be a special mechanism for

---

(1) Note that the method of accessing parameters used by the level 1 implementation does <u>not</u> generate an upward dependency on the virtual memory mechanism. The specification of the level 1 interface is that it reflects an error and does not do the operation if its parameters are not in primary memory. No matter what the virtual memory manager does, it cannot cause a level 1 operation to fail to meet its specification either by doing the operation or reflecting an error status.

handling the asynchronous modification of the map during an operation of a level 1 processor.

It is instructive to investigate the similar problem found in the physical processor instructions. The physical processor operates by converting the addresses found in instructions through the map into real addresses, then accessing the real addresses directly during the instruction. The modification to the map is thus not reflected immediately in the processor's accessing, but must wait until the processor stops using the converted address. The processor converts all addresses to real addresses before actually accessing the data operated on by the instruction. Discovering a fault is thus done before the instruction has taken irreversible steps, so the instruction can be restarted from the beginning.

There is, however, a problem in the physical processor accessing of memory. The main reason for changing the map is that a page or segment is moved from primary to secondary memory or vice versa. When the page is moved to secondary memory, it must be guaranteed that no processor has outstanding references to it. This guarantee is provided by marking all maps that refer to the page so that a fault will be generated when the page is referenced. However, for a short period of time the physical processor may have a translated real address that refers to the page. The moving of a page from primary to secondary memory proceeds as follows: first, flag all maps referring to the page, then, wait until all physical processors stop using the translated real addresses they were using at the time the flags were set in

the maps. These two steps together guarantee that the page can be moved safely.

For the software parts of the level 1 processor manager, similar mechanism must be provided. The software parts will first translate the addresses of parameters using the map into the address space of the level 1 manager. The level 1 manager address space cannot be modified by higher levels in the system. Any faults in accessing parameters are discovered and reflected during the translation, so that after translation is complete the parameters are guaranteed to be accessible. Then, the level 1 manager will use the translated addresses to reference primary memory. Before the page manager can move anything in primary memory, it must first flag the map, then wait until any translated addresses being used in level 1 operations are done with. The level 1 processor must have a special mechanism to achieve this waiting. This mechanism is a level 1 instruction,

VP1$propagate_map_change (),

that causes the invoking level 1 processor to stop executing further instructions until all other processors having translated copies of addresses finish their current level 1 processor operation. (1)

---

(1) In many real processors, translated primary memory addresses are held between operations in an associative memory built into the processor. In this case, finishing the current level 1 processor operation is insufficient to guarantee that no translated addresses are being held by the processor. Consequently, the operation VP1$propagate_map_change also has to cause all associative memories on all processors to be cleared.

Chapter Five

Level 1 Processor Implementation

(The reader who is not interested in the details of an implementation of level 1 processors may choose to skip this chapter, without much loss of continuity.)

In this chapter, two implementations of level 1 processors on a multiprocessor, shared primary memory computer system are described. The two implementations are actually closely related. The first version of the implementation relies on a slightly non-traditional hardware that uses a specialized processor as a central agent to control the multiplexing of the other processors of the system. Within this architecture, the implementation of level 1 processors is quite simple to describe. The second implementation shows how, with extra complexity and a small loss of efficiency, the specialized processor can be simulated on general-purpose processors such as those of Multics.

The first implementation is not intended just as a basis for developing the second, however. Adding a microprocessor to the architecture of a system such as the Honeywell Level 68 to implement level 1 processor multiplexing would not be at all difficult or expensive. The changes that must be made to the general purpose processors to implement the binding and unbinding functions in hardware amount to simplifications of structure; they would, however, be relatively expensive to retrofit into current processors.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

The proposed hardware architecture is relatively simple to incorporate into newly designed multiprocessor systems. Incorporating the ideas about architecture described here should be worthwhile in terms of simplifying the design of multiprogramming operating systems.

5.1 Overall Structure of the Implementation

The level 1 processor implementation follows the model of processor multiplexing presented in chapter two, using a central agent to control processor multiplexing. The central agent is implemented on a dedicated processor called the Processor Control Processor. It controls the general-purpose processors (GPPs) of the system by controlling their binding to level 1 processors. Within the implementation, level 1 processors are represented by level 1 processor states stored in primary memory. The central agent is also responsible for implementing the IPCC mechanisms, coordination of level 1 processors with events in I/O processors, and reconfiguration of the GPPs, since IPCC, I/O events, and reconfiguration may indirectly require reassignment of GPPs to a different set of level 1 processors.

Figure 5.1 shows the pattern of communication among the processors in the system. Level 1 processors are executed on the GPPs. The PCP communicates with each GPP to control its assignments to level 1 processors. The operations described in chapter four that allow level 1 processors to affect other level 1 processors are all implemented in the PCP. When a level 1

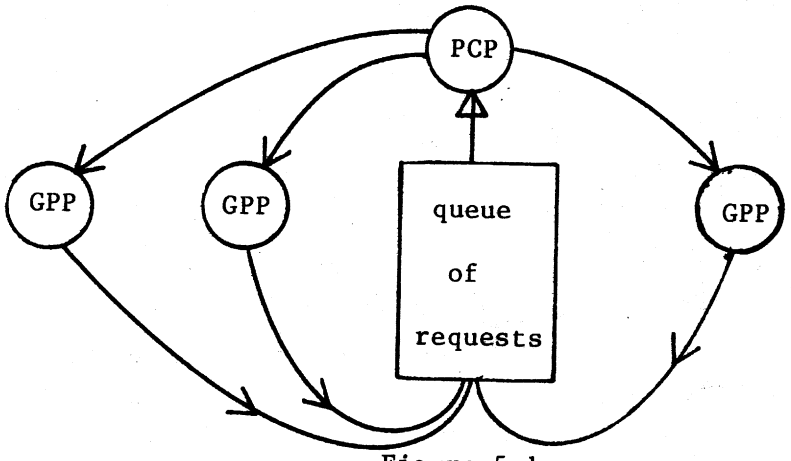


Figure 5.1
Processor Communication in Level 1 Implementation

processor executes one of these operations, its GPP actually communicates a request to the PCP, which performs the operation.

The PCP actually handles one request from a GPP at a time. Successive requests are queued. In order to keep the GPPs as busy as possible, once a GPP has queued a request, it can proceed to execute, without waiting for the request to be processed by the PCP. In the case of operations like VP1$run, VP1$stop, and VP1$advance, the GPP proceeds to execute the level 1 processor that executed the operation. Other operations, like VP1$await, require that the GPP not continue executing the level 1 processor executing the operation.

To prevent the GPP from being excessively idle during periods when a burst of requests are sent to the PCP, the function of choosing the next level 1 processor to run on a GPP is distributed among the GPPs. There is a shared priority queue that all GPPs can access containing all runnable level 1 processors in priority order. Figure 5.2 shows this queue. When a GPP

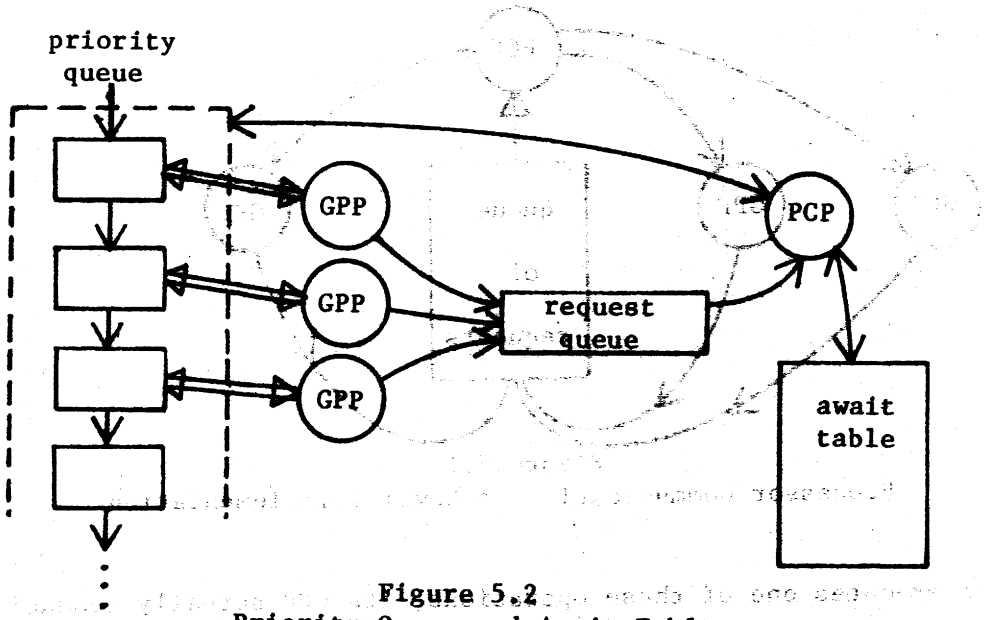


Figure 5.2
Priority Queue and Await Table

determines that it cannot continue running its current level 1 processor, it will take the highest priority runnable level 1 processor from this queue, and run it.

The PCP controls the bindings of level 1 processors to GPPs indirectly. The queue of runnable level 1 processors is altered by the PCP to reflect any changes in the runnability of the level 1 processors. After such a change has been made, the GPPs must be reassigned. The PCP accomplishes the reassignment by determining the GPPs that are improperly assigned, and forcing them to unbind themselves from the current level 1 processor, and reassign themselves based on the newly altered queue of runnable level 1 processors.

Also distributed in each GPP is the handling of the quantum for each level 1 processor. Each GPP keeps track of the time it spends executing each level 1 processor, so that when the level 1 processor quantum is exceeded, the GPP informs the PCP and reassigns itself to a runnable level 1 processor.

Interprocess Control Communication is centralized in the PCP. The PCP maintains a table, called the await table (see figure 5.2), that keeps track of the level 1 processors that are awaiting along with the eventcount names and values awaited. An advance operation proceeds by having the GPP executing the advance increment the value of the eventcount, then transmit to the PCP the name of the eventcount and its new value. The PCP then processes this information by finding all of the level 1 processors that should be awakened, and awakening them. The special eventcounts (stopped, clock, I/O eventcounts, outward_signals) are not advanced by GPPs, but are handled within the PCP. The clock and I/O processor eventcounts are handled by periodic polling of their values in the PCP. The stopped and outward_signals eventcounts are advanced by the PCP, and reflected to the level 1 processors.

5.2 Hardware Architecture

Although the hardware architecture is slightly different than that of a traditional multiprocessor computer system, I have tried to make the number of differences as few as possible. The GPPs of the system look very much like the physical processors of traditional computer systems. Most of the implementation of level 1 processor manager is in software. I have chosen a minimal set of hardware facilities needed to implement the level 1 processor manager. These facilities are:

1. A mechanism that allows the PCP to interrupt the GPPs.
2. Shared primary memory to be used for communication of data between PCP and GPPs.
3. A special mode of execution in the GPP used to allow the implementation of the GPP part of level 1 operations in software on the GPPs.
4. A special instruction that translates addresses within the level 1 processor environment into a version that is unaffected by changes made to the environment specification.
5. A special instruction that allows the GPP to change its binding to a new level 1 processor.

These features are discussed in detail below.

5.2.1 The Processor Control Processor

The processor control processor (PCP) is a highly specialized processor that controls the multiplexing of the general-purpose processors of the system. It need not be a high-speed processor, nor must it have any of the facilities needed for handling general purpose computations, such as interrupts, faults, powerful instruction set, large memory, etc. It is probably best implemented as a microprocessor.

The PCP communicates with the general-purpose processors of the system through the system's primary memory. The PCP can read and write primary memory, although it need not store either its program, or most of its data in primary memory.

The PCP can also send a special signal, called UNBIND, on lines that connect the PCP to each individual general-purpose processor. Figure 5.3 shows the communication paths of the system. The UNBIND signal is used by the PCP to cause a processor to stop doing what it is doing, and find a new level 1 processor to run.

The UNBIND signal is the only interrupt-like operation in the system. There are no interrupt signals for the PCP, since it operates by repeatedly polling the primary memory cells of interest to it. The I/O processors will

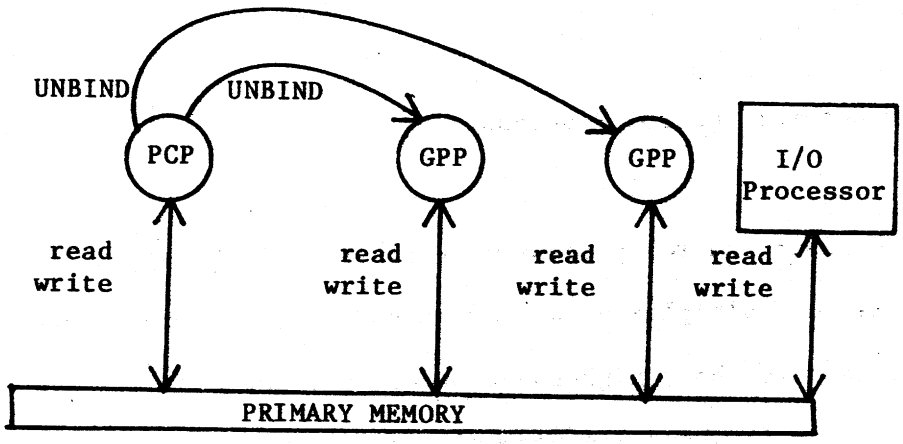


Figure 5.3
Hardware Communication Paths

communicate with level 1 processors purely through memory. If an I/O processor needs to send a signal to a particular level 1 processor, it will increment a memory location treated by the PCP as a special eventcount, and the eventcount will be observed by the PCP and reflected to the level 1 processor. Each GPP is able to send a control signal to each I/O processor to start it executing, by advancing an eventcount (actually a counter, since it is not handled by the normal eventcount mechanisms) that is polled by the I/O processor while the I/O processor is stopped.

5.2.2 General-Purpose Processors

The general purpose processors (GPPs) of the system are much like the general purpose processors of Multics, the IBM System/370, etc. They all access primary memory through address translation hardware that is controlled

by a data base in primary memory called a descriptor segment. Each GPP has a set of internal registers, some of which are used to perform computational operations of the level 1 processor, and some of which are used in the level 1 processor multiplexing implementation. The structure of the internal memory

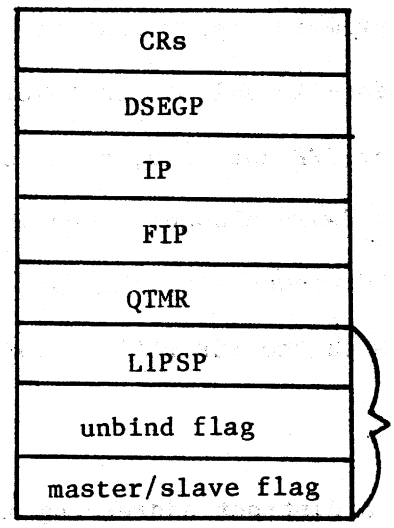


Figure 5.4
GPP Internal Memory

of a GPP is indicated in figure 5.4. Most items are familiar from chapter four. The bracketed items are explained shortly.

The GPP operates in one of two modes, master mode and slave mode. In slave mode, the GPP is running a level 1 processor. Its instruction pointer, computational registers, descriptor segment pointer, and fault handler pointer are all used in slave mode. The slave mode instructions allow the processor to access memory through the descriptor segment, perform operations on its computational registers, transfer, and so forth. One additional slave mode

operation, INVOKE-LEVEL1, allows the GPP to enter master mode for the purpose of communicating with the PCP.

Master mode in the GPP exists so that the level 1 processor operations that need to communicate with the PCP can do so. In master mode, the GPP has access to the data bases in primary memory that are shared with the PCP. Master mode would be unnecessary if all of the level 1 processor management operations were built into the GPP hardware, but I have attempted in this design to make the minimal hardware changes necessary for a clean design of the level 1 implementation. Consequently, the operations that allow the level 1 processors to communicate with the PCP will be software operations run in master mode.

Master mode executes in a distinct addressing mode from the level 1 processor environment accessed in slave mode. The separate environment protects the code executing in master mode from errors in the level 1 processor environment. Since the level 1 processor environment is controlled at a level higher than the level 1 implementation, level 1 cannot depend on the correctness of the environment in any level 1 processor without causing a cyclic dependency.

In the master mode environment, it must still be possible for the GPP to access parameters to level 1 operations that are stored in the level 1 environment. The simplest choice is to have the master mode environment able to access absolute core addresses directly. An alternative would be to have

master mode use a different map, but the difficulty of converting addresses in terms of the level 1 processor map to the equivalent addresses in a distinct master mode map make this alternative unattractive. When in master mode, addresses in code executed by the GPP are interpreted as absolute core addresses.

The special functionality of the GPP must now be discussed. The level 1 processor state pointer in the GPP is a pointer (actually an absolute core address) to the level 1 processor state in primary memory that corresponds to the level 1 processor currently bound to the GPP. The GPP uses this pointer to store the state of the level 1 processor when the GPP enters master mode. This pointer is also used to store the fault data when a level 1 processor takes a fault.

The format of a level 1 processor state block in memory is shown in figure 5.5. The level 1 processor state block contains information that is available at the level 1 interface, and some that is not. The current state, containing computational register values (CRs), a instruction pointer (IP), a fault handler pointer (FIP), a quantum timer register value (QTMR), and an environment descriptor pointer (DSEGP), corresponds to the state information presented at the level 1 interface by the bind and unbind operations. It also corresponds to the state of a GPP. This is the state that is loaded into a GPP when the GPP is bound to the level 1 processor. The fault data, containing computational registers (CRs), instruction pointer, and fault code (FCODE), is kept here so that the VP1$get_fault_state operation can access it.

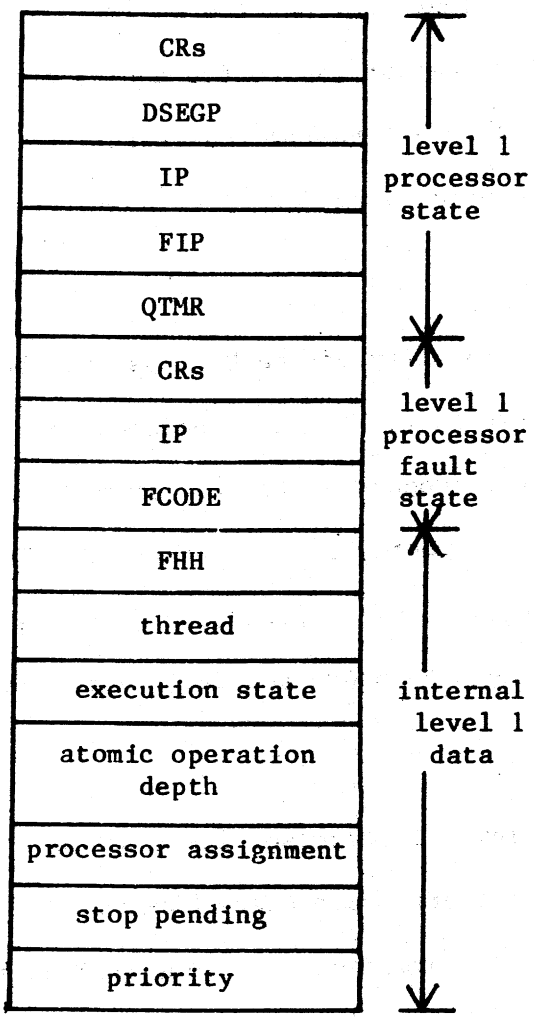


Figure 5.5
Level 1 Processor State Block

The GPP sets the fault state when a fault occurs, and also sets the flag that indicates that a fault has happened (FHH). If the FHH flag is already on when a fault occurs, the GPP unbinds itself as if the level 1 processor had executed VP1$crash_system. The rest of the data in the state block is not interpreted by the hardware and will be described in detail later.

In master mode, there are two special instructions that cannot be used in slave mode. The first, ACCESS, allows the GPP in master mode to interpret an address relative to a specifed descriptor segment. This instruction will be used to allow the GPP to translate data addresses from the address space of a level 1 processor into the master mod (that is, absolute core addresses). address space. If the ACCESS instruction encounters a missing-page or missing-segment fault, it will set a condition code indicating the fault that occurred, and proceed to the next instruction. The ACCESS instruction loads a register of the GPP with the address in the master mode address space that corresponds to the specified address in the specified descriptor segment. It also loads into another register the system-wide unique address, from the map, of the word.

The other special master mode instruction is LOADSTATE. The LOADSTATE instruction allows the GPP to load a particular level 1 processor state from an address in the GPPs master mode environment into the GPP's registers. The master mode flag is then turned off, and the GPP begins executing the level 1 processor. The level 1 processor state pointer of the GPP is loaded with the address of the level 1 processor state block named in the LOADSTATE instruction.

Two other special registers are present in the GPP. The quantum timer register is a register loaded from the level 1 processor state that contains a value that is decremented once every microsecond. When the register reaches zero, it stops decrementing.

The unbind flag is set by the PCP UNBIND signal. The unbind flag is checked after executing each instruction when the GPP is in slave mode. A set flag causes the GPP to unbind itself from the level 1 processor it is currently executing. The GPP also unbinds itself from the current level 1 processor when the INVOKE-LEVEL1 operation is executed. The basic cycle of the GPP is shown in figure 5.6.

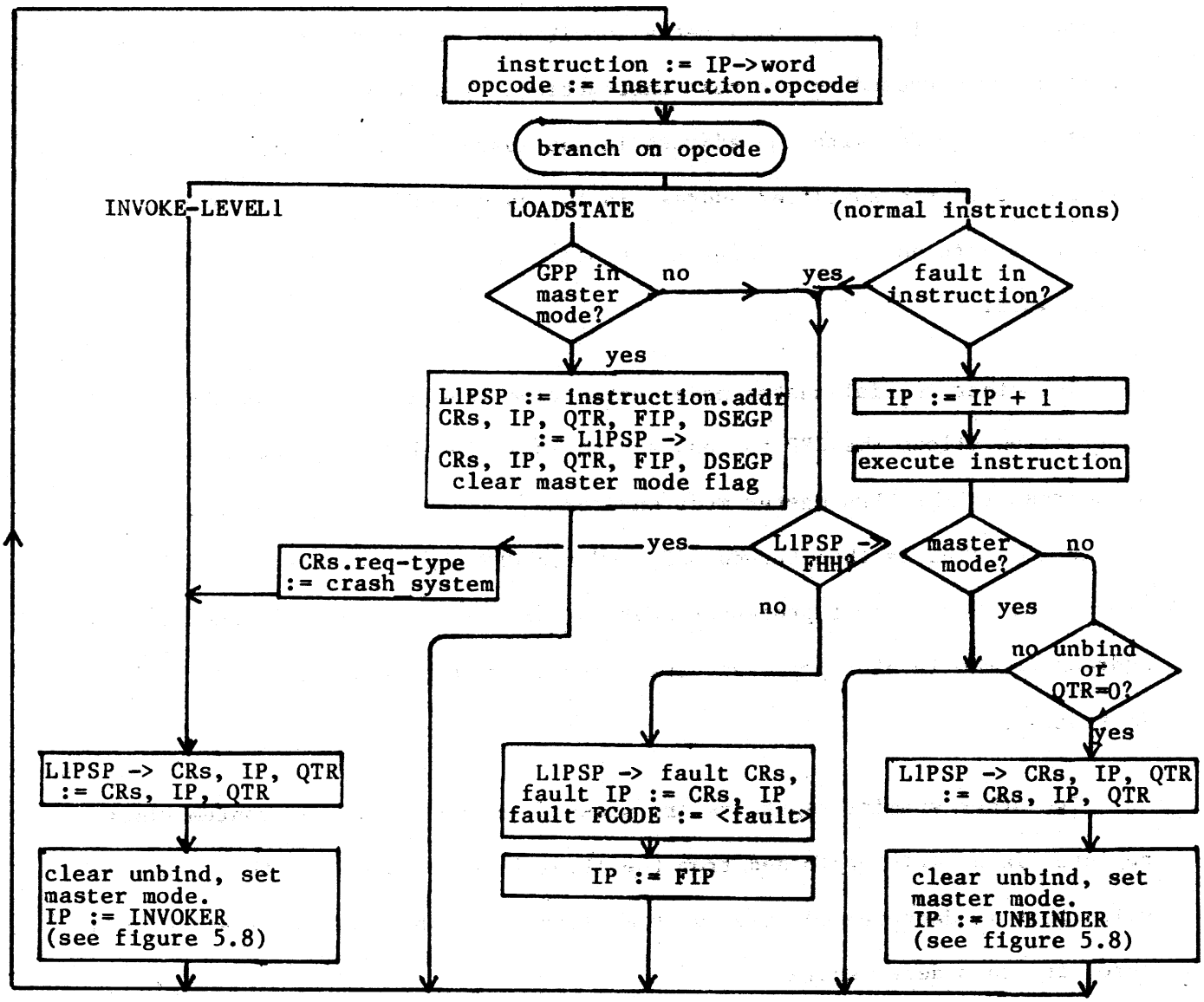


Figure 5.6
Basic GPP Cycle

5.3 Data Bases

There are four data bases used in the level 1 processor implementation. They are the level 1 processor state table (L1PST), the PCP request queue (PCPRQ), the await table (AT), and the GPP control table (GCT). The first two data bases are accessed both by GPPs and the PCP, so there is a locking mechanism required for each; the AT, however, is private to the PCP, so no locking is required. The GPP data items are each only written in by one processor so there is no need for a lock.

The level 1 processor state table consists of an array of level 1 processor state blocks. The format of a level 1 processor state block has been shown in figure 5.5. Each level 1 processor state block stores all of the state information about a level 1 processor, along with certain information used to schedule the assignments of physical processors to level 1 processors. All of the non-stopped level 1 processors are threaded into a list in order of decreasing priority. The stopped level 1 processors are either unthreaded, or threaded into a list called the next-stopped queue used to implement the VP1$next_stopped operation. Each level 1 processor state block has stored in it the state of execution of the level 1 processor; it may either be running, runnable, awaiting, stopping (a transient state on the way to stopped), or stopped.

The information not yet described in the level 1 processor state block is used as follows (see figure 5.5). The thread value is used to thread the block onto the priority queue or the next_stopped queue. The execution state is stored in the execution_state value. If the level 1 processor is running on a GPP, the name of the GPP is stored in the state block. The atomic operation depth contains the number of times a VP1$begin_atom_operation has been executed without a matching VP1$end_atomic_operation. The stop_pending flag is used to remember that the level 1 processor must be stopped after its atomic_operation depth reaches zero. The priority is permanently associated with a level 1 processor, and is used to find the right place to thread the level 1 processor into the priority queue.

The data in the level 1 processor state table is protected by a lock called the L1PST lock. The data in the L1PST will not change while the L1PST lock is set, with one exception. A level 1 processor state block that is marked in the running state can undergo certain modifications at any time. The stored registers, instruction counter, quantum timer register, fault information, and PCP request type fields may be modified by the GPP running the level 1 processor at any time while the level 1 processor state block is marked as running; none of the remaining data may be modified except by locking the L1PST lock.

The PCP request queue is a FIFO queue used to send messages to the PCP. It is a fixed size block of storage, probably best managed as a ring buffer. A lock called the PCP request lock prevents more than one GPP from placing

messages in the queue at the same time. Its size should be chosen to minimize the amount of time spent waiting for the PCP to free up enough space for the next message, which waiting is done by busy-waiting in the GPP. The queue must be at least as large as the largest message placed in it.

The await table is kept internally to the PCP and keeps track of the mappings from eventcounts awaited by level 1 processors to the level 1 processors awaiting, and vice versa. Its format is unimportant to the current discussion, as long as it is possible to convert an eventcount name and current value into a list of the level 1 processors to awaken, and it is possible to delete the entries from the table that correspond to a particular level 1 processor. A simple form of the table might be a list of three-tuples: eventcount name, awaited value, and level 1 processor name. However, there are much more effective ways of obtaining the desired functionality than such a list.

The GPP control table contains entries for each GPP. There are two data items in each entry. The first is a flag that indicates whether the GPP is available for use by level 1 or not, for reconfiguration. It is modified only by the PCP. The second entry is a counter incremented each time the GPP finishes executing an unbind operation, either due to an UNBIND signal from the PCP, or due to timer runout or INVOKE-LEVEL1 in the GPP. It is used in the implementation of VP1$propagate_map_change; this use is described later with the implementation of VP1$propagate_map_change.

5.4 Operation of the Processor Control Processor

The PCP has three functions to perform. First, it must manage the bindings of GPPs to level 1 processors. Second, it must do the work of the requests in the PCP request queue, calling for the PCP to run and stop level 1 processors, add and delete GPPs, enter level 1 processors into the await table, and awaken the level 1 processors awaiting a particular advance. Third, it must implement the special eventcounts -- the outward_signals eventcount, the stopped eventcount, the clock eventcount, and the eventcounts associated with I/O processors.

The PCP does all of these things by periodically polling the relevant data bases, and then performing the necessary actions. Basically, the PCP executes in a loop, first checking the PCP request queue for requests and doing the ones found in the queue, then checking the special eventcounts against the entries in the await table to see if any level 1 processors should be awakened, then checking the level 1 processor assignment table to make sure that all GPPs are properly assigned and issuing the appropriate UNBIND signals to correct any discrepancies.

There are nine kinds of requests that are sent from GPPs to the PCP through the PCP request queue. Here the data associated with the requests and the processing done by the PCP are described. A flow chart of the operational

cycle of the PCP appears in figure 5.7.

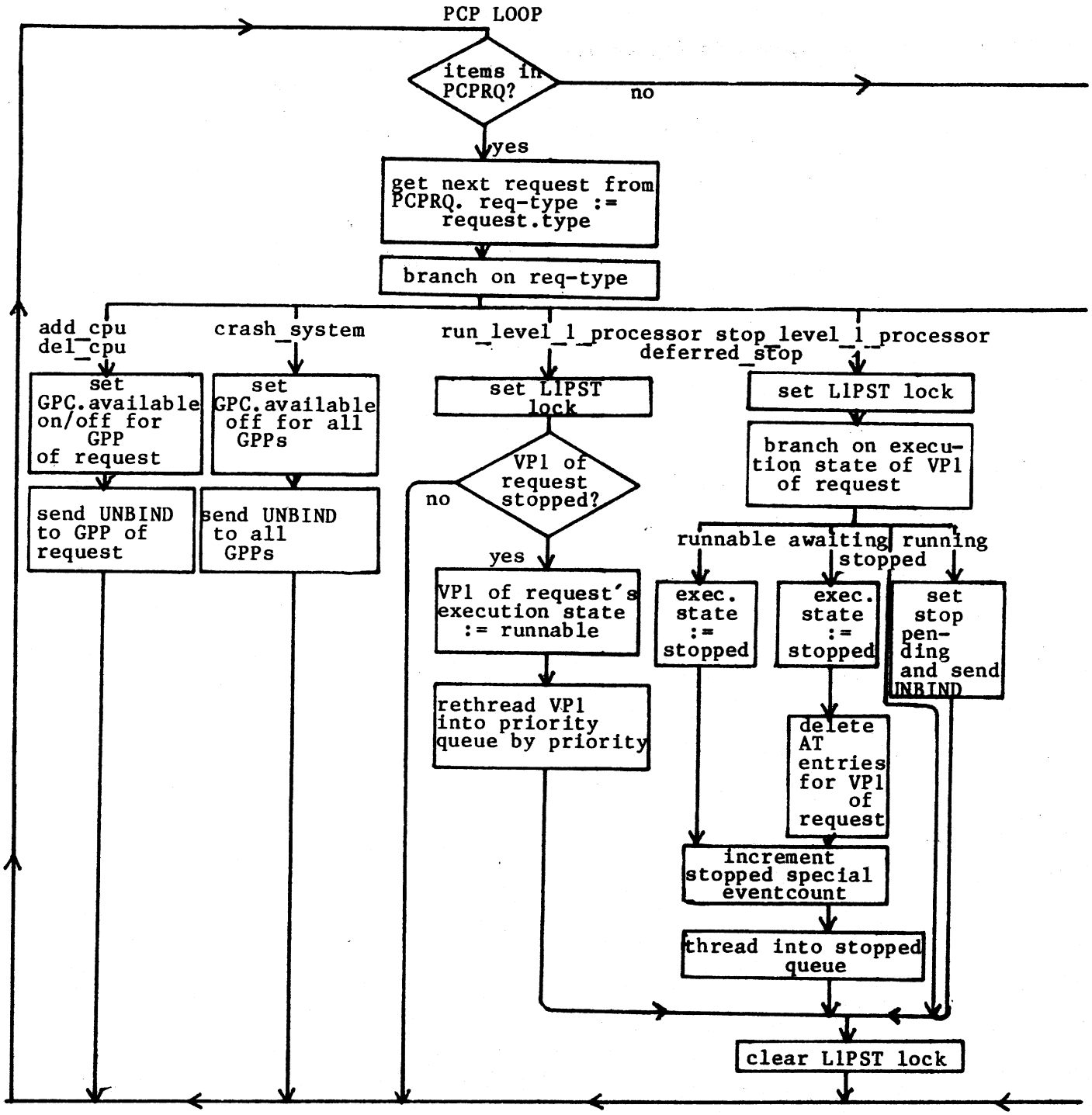
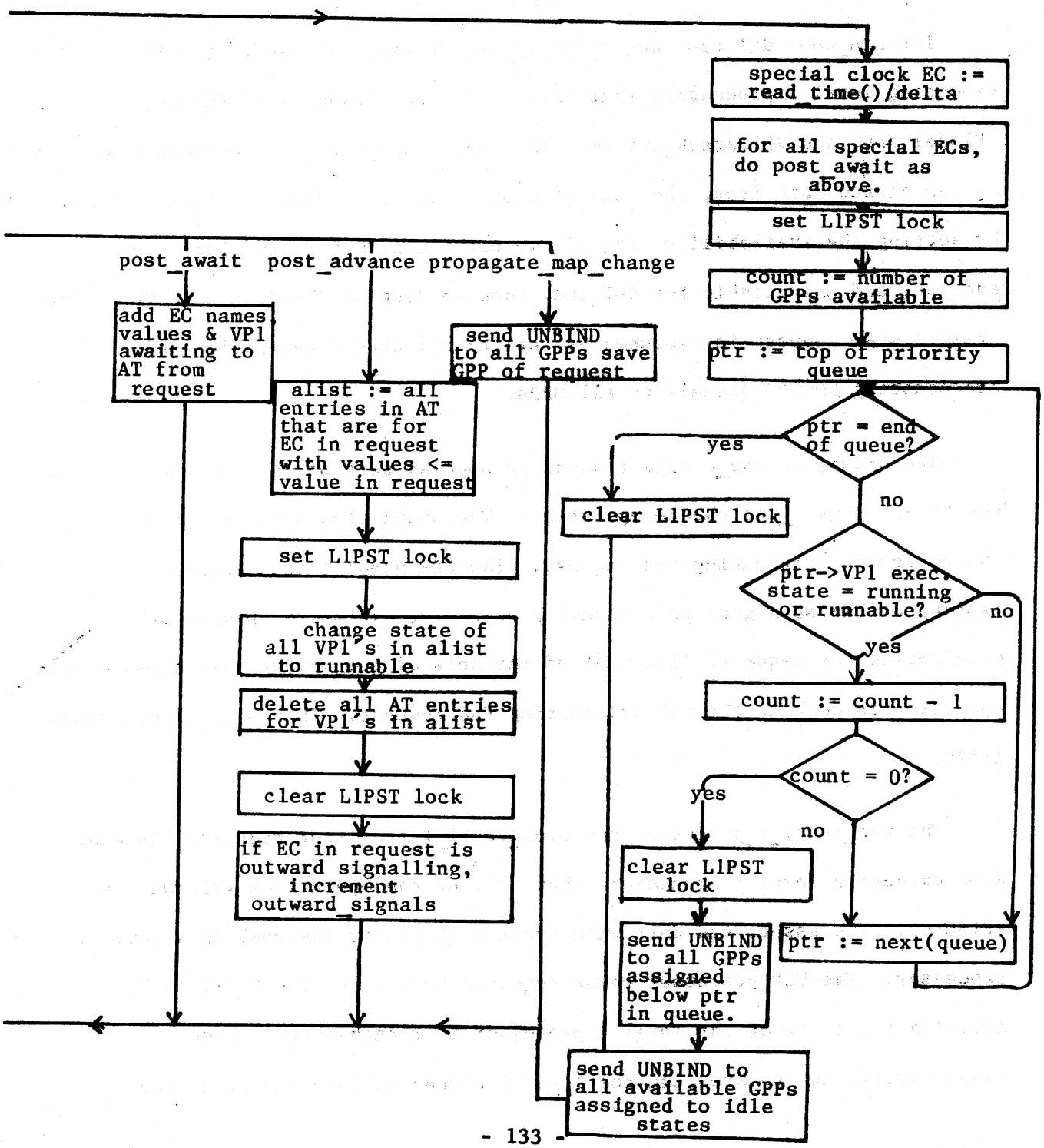


Figure 5.7
PCP Algorithm Flow Chart


special clock EC := read time()/delta
for all special ECs, do post_await as above.
set L1PST lock
count := number of GPPs available
ptr := top of priority queue
ptr = end of queue?
yes
no
clear L1PST lock
ptr->VP1 exec. state = running or runnable?
no
yes
count := count - 1
count = 0?
yes
no
clear L1PST lock
send UNBIND to all GPPs assigned below ptr in queue.
ptr := next(queue)
send UNBIND to all available GPPs assigned to idle states
post_await
post_advance
propagate_map_change
add EC names values & VP1 awaiting to AT from request
alist := all entries in AT that are for EC in request with values <= value in request
send UNBIND to all GPPs save GPP of request
set L1PST lock
change state of all VP1's in alist to runnable
delete all AT entries for VP1's in alist
clear L1PST lock
if EC in request is outward signalling, increment outward_signals

The add_cpu, del_cpu, and crash_system requests are sent by GPPs executing level 1 processors that call on the operations VP1$add_cpu, VP1$del_cpu, and VP1$crash_system. The add_cpu and del_cpu requests also have an associated data item, the name of a GPP. The PCP processes these requests by setting the availability flag of the particular GPP to available for add_cpu, and unavailable for del_cpu, then sending an UNBIND to the GPP. The crash_system request is executed by marking all GPPs unavailable, and broadcasting UNBIND signals to all GPPs.

The propagate_map_change request is used as part of the implementation of the VP1$propagate_map_change operation. The associated data is the name of the processor originating the request. The PCP handles this request by issuing an UNBIND signal to all real processors, except the processor originating the request. The rest of the work of the VP1$propagate_map_change operation is done in the GPP originating the request. This will be discussed later.

The run_level_1_processor and stop_level_1_processor requests are sent by GPPs executing level 1 processors that call on the operations VP1$run and VP1$stop. The associated data with these requests is the name of a level 1 processor. The PCP processes these requests by locking the L1PST lock, altering the state of the level 1 processor to runnable or stopped, respectively, and rethreading the level 1 processor into the processor

priority list or the next-stopped list. (1) If the level 1 processor is being stopped, it also must have all associated entries removed from the PCP await table, so that the space can be reused. (2) The L1PST lock is then unlocked.

The processing of the stop_level_1_processor request is not actually quite this simple. If the level 1 processor is either running or is in the middle of an atomic operation (its atomic operation depth is non-zero), the level 1 processor cannot be stopped immediately. In this case, instead of changing its state to stopped, a flag will be set in the level 1 processor state block to indicate that a stop is pending. If the level 1 processor is running, it will be sent an UNBIND signal to ensure its speedy stopping. The pending stopped flag is interpreted by the GPP at the time of an unbind, and will cause the GPP to put the level 1 processor in the special stopping state, and then send a deferred_stop message in the PCP request queue.

The deferred_stop message is sent to the PCP under three conditions. In an unbind operation on the GPP, if the pending stop flag is found on in the current level 1 processor state block, and the level 1 processor atomic operation depth is zero, then a deferred_stop is sent to the PCP. If the quantum timer runs out, and the atomic operation depth is zero, then a

---

(1) Whenever the next-stopped list has a new level 1 processor added to it, the PCP increments the special stopped eventcount, The increment is observed later by the PCP when checking the special eventcounts, and reflected then to the awaiting level 1 processors.

(2) Please recall that executing VP1$run on a stopped level 1 processor will cause the VP1$await instruction to be re-executed, so that the information in the PCP await table will be regenerated at that time.

deferred_stop is sent to the PCP. If the level 1 processor executes a VP1$end_atomic_operation instruction that decrements the atomic operation depth to zero, and the stop pending flag is on, or the quantum timer has run out, a deferred_stop is sent to the PCP.

The level 1 processor sending the deferred_stop message is put into the special stopping state by the GPP. The data contained in a deferred_stop message is the name of the level 1 processor being stopped. The PCP processes a deferred_stop message in the same way it processes a stop_level_1_processor request, except that it need not check to see if the level 1 processor is stoppable.

The post_advance PCP request is sent by the GPP executing an advance operation to cause the level 1 processors awaiting the advance to be awakened. The actual incrementing of the eventcount is done by the GPP; the PCP need only search its await table for the level 1 processors to awaken, and perform the awakening. The data sent with the post_advance request is the system-wide unique address of the eventcount and the value of the eventcount after incrementation. The PCP performs this request by finding all entries in the await table that have the same system-wide unique address with awaited values less than or equal to the value sent in the post_advance request. It then locks the L1PST lock, finding all of the level 1 processors that are named in the above-mentioned await-table entries. The state of each of these level 1 processors is changed from awaiting to runnable. When the level 1 processor is next run, it will re-execute and find that one of the eventcounts has been advanced, so it will proceed.

The PCP also checks each post_advance request to see if the advance was on an outward signalled eventcount. If so, it increments the special outward_signals eventcount (the posting of the outward_signals eventcount occurs later).

The last PCP request is post_await. It is sent by a GPP to the PCP after checking the eventcounts awaited in a VP1$await operation, if none of the eventcounts is greater than or equal to the values awaited. The data sent to the PCP are the name of the level 1 processor awaiting, and pairs of system-wide unique addresses of eventcounts and awaited values. (1) The PCP responds to these requests by adding entries to the PCP await queue for each of the eventcounts.

After processing the PCP request queue, the PCP handles the special eventcounts. The system's calendar clock is read by the PCP and it decides whether to increment the clock eventcount. The PCP then reads each special eventcount, getting its current value. It then acts as if it received a post_advance for each special eventcount, searching the await table for awaiting level 1 processors, and awakening them. The PCP can always directly access the special eventcounts. There are only a few such eventcounts. They are the stopped eventcount, the clock eventcount, the outward_signals

---

(1) Please note that the limit on the number of eventcounts in a VP1$await operation is associated both with the maximum size message that is sent through the PCP request queue, and with the maximum number of entries that can be placed in the PCP await table. The more eventcounts that a level 1 processor can await, the larger these tables.

eventcount, and the I/O device eventcounts. These eventcounts are handled specially in the PCP because the agents that increment the eventcounts do not use the PCP request queue, and so do not use post_advance requests to reflect the incrementing to level 1 processors.

The final step of the PCP is to update the assignments of GPPs to reflect the changes in the level 1 processor states and bindings. This step is done by locking the L1PST lock, and inspecting the assignments of GPPs reflected in the level 1 processor states. The PCP then issues UNBIND signals to a set of GPPs so that the GPPs will reassign themselves to the correct set of level 1 processors, based on the priority ordering of the level 1 processors.

The algorithm used to choose the GPPs to unbind is very simple. The PCP knows how many GPPs are on the system. By starting at the top of the priority queue in the level 1 processor state table, and counting running and runnable level 1 processors as the queue is traversed until as many are found as there are GPPs, the PCP can find the set of level 1 processors that should be running. If any GPPs are running lower priority level 1 processors, they should be preempted by sending an UNBIND signal. The PCP thus traverses the rest of the priority queue, sending UNBIND signals to GPPs running any lower priority level 1 processors.

5.5 GPP operation

The way that level 1 processor operations are implemented on GPPs is by using the INVOKE-LEVEL1 instruction. The INVOKE-LEVEL1 instruction causes the GPP to enter master mode, and to transfer to the unbind handler. A flag is set in the level 1 processor state by the INVOKE-LEVEL1 instruction to indicate that a INVOKE-LEVEL1 has been executed. The type of level 1 processor operation to be performed is transmitted in a register, and the addresses of any data, such as eventcounts, etc., required by the operation are transmitted through registers.

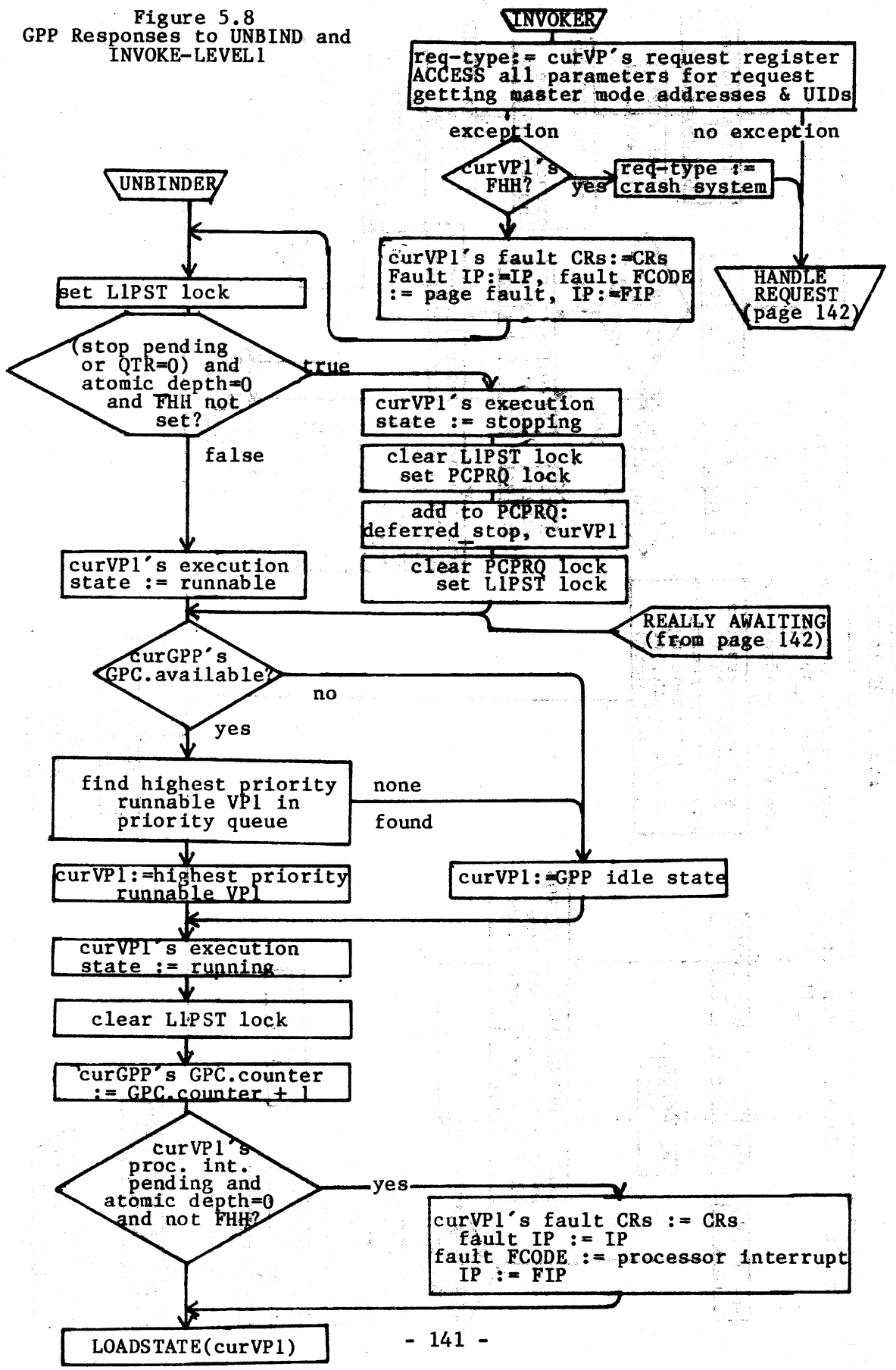
To simplify the discussion of the unbind operation, we must first discuss the handling of exceptions, such as missing page exceptions, in accessing the data associated with a particular operation. The data will be accessed by first using the ACCESS master mode instruction to convert the address of the data in the address space of the level 1 processor into an address that is reachable in the master mode address space. If the ACCESS instruction encounters a missing-page exception, it reflects this in the condition code, rather than faulting. If a missing page condition occurs, the code in the unbind sequence will abort the current operation, and update the level 1 processor state to simulate a missing-page fault, moving the current copies of the computational registers to the fault data, along with the instruction

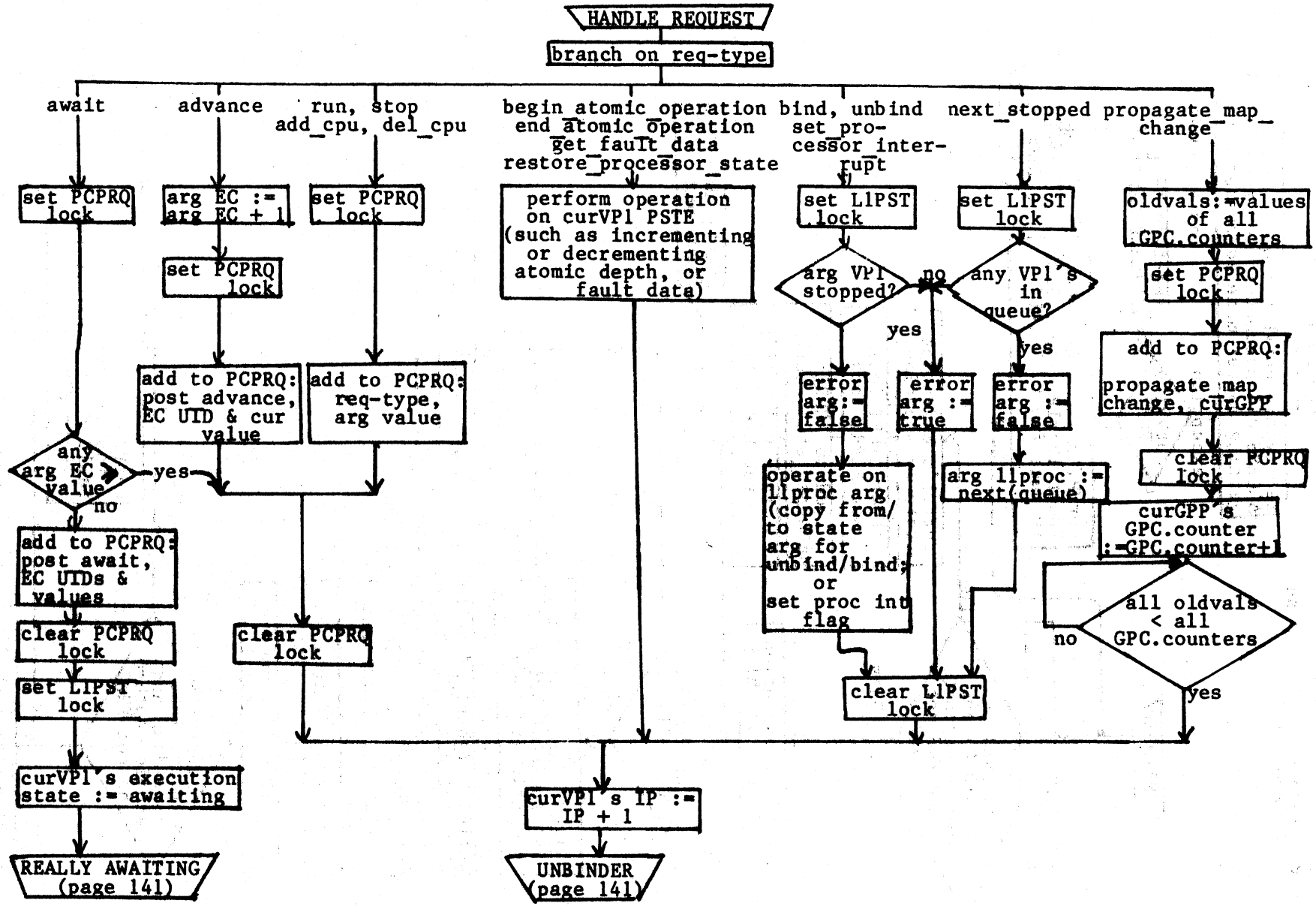
counter, and setting the fault code to indicate the type of fault encountered. The current instruction counter of the level 1 processor will then be set to the fault handler address. The GPP will then proceed with finding a level 1 processor to execute.

If no fault is detected by the ACCESS instruction, then the GPP can perform the rest of the operation correctly. Having determined the address of data in the master mode environment, the GPP can then proceed to access these objects, without fear of encountering faults.

The unbinder that executes in master mode in all GPPs is described in the flowchart in figure 5.8.

Figure 5.8
GPP Responses to UNBIND and INVOKE-LEVEL1




HANDLE_REQUEST
branch on req-type
await
advance
run, stop
add_cpu, del_cpu
begin_atomic_operation
end_atomic_operation
get_fault_data
restore_processor_state
bind, unbind
set_pro-
cessor_inter-
rupt
next_stopped
propagate_map_
change
set PCPRQ lock
arg EC := arg EC + 1
set PCPRQ lock
perform operation on curVP1 PSTE (such as incrementing or decrementing atomic depth, or fault data)
set L1PST lock
set L1PST lock
oldvals:=values of all GPC.counters
set PCPRQ lock
arg VP1 stopped?
no
any VP1's in queue?
yes
yes
add to PCPRQ: post advance, EC UID & cur value
add to PCPRQ: req-type, arg value
add to PCPRQ: propagate map change, curGPP
error arg:= false
error arg := true
error arg := false
any arg EC value
yes
no
operate on l1proc arg (copy from/ to state arg for unbind/bind; or set proc int flag
arg l1proc := next(queue)
clear PCPRQ lock
curGPP's GPC.counter := GPC.counter+1
add to PCPRQ: post await, EC UIDs & values
clear PCPRQ lock
clear PCPRQ lock
all oldvals < all GPC.counters
no
yes
clear L1PST lock
set L1PST lock
curVP1's execution state := awaiting
curVP1's IP := IP + 1
REALLY AWAITING (page 141)
UNBINDER (page 141)

The basic flow of the unbinder is quite simple. If the unbind is due to an INVOKE-LEVEL1 instruction, the request is handled. Then, the L1PST lock is locked, and the level 1 processor is checked to see if it should be stopped. If it should be stopped, the level 1 processor is placed in the stopping state and a request is sent to the PCP. If not, it is marked as runnable. The GPP then searches the priority queue for the highest priority runnable level 1 processor. It is marked as running, the L1PST lock is unlocked, and the GPP uses the LOADSTATE instruction to run the level 1 processor, having set up a simulated fault if a processor interrupt is to be sent to the level 1 processor.

The only exception to this basic flow is the handling of the PCP request associated with the VP1$await instruction. In order to ensure that an advance operation does not happen and get inserted into the PCP request queue between the time the eventcounts are tested and the time the post_await message is entered in the PCP request queue, the eventcounts are tested while the PCP request queue lock is locked. The GPP then decides whether to enter the post_await message into the PCP request queue or not, and unlocks the PCP request queue. (1) If the post_await message is entered, the level 1 processor is marked as awaiting, otherwise, the instruction counter is advanced passed the INVOKE-LEVEL1 instruction, and the unbind proceeds as before.

---

(1) The problem I am solving here is the same critical race Saltzer [25] describes, which in his case necessitates a wakeup-waiting switch that is tested under a lock. The eventcounts themselves serve the same purpose as the wakeup-waiting switch in this implementation.

The advance operation is very simple. It simply increments the memory word of the eventcount, and transmits the new value, and system-wide unique address (obtained in the ACCESS instruction) through the PCP request queue, in a post_advance request.

The propagate_map_change operation is fairly subtle in its operation. The implementation works by causing all GPPs other than the current one to unbind themselves, then waiting until they complete their next unbind operation. To know when each GPP finishes its next unbind operation, there is a table of counters, one for each GPP on the system. Each time a GPP completes an unbind operation, it increments its counter. The propagate_map_change operation is done in three steps. First, the GPP reads the current values of the counters associated with each other GPP. Second, it sends a propagate_map_change PCP request. Third, it busy-waits until each other GPP's counter is greater than the value of the counter obtained in the first step. By the time the third step is completed, all GPPs will have completed at least one unbind operation after the VP1$propagate_map_change operation started. Consequently, there will be no copies of absolute addresses obtained from the maps retained in the processors that were generated before the VP1$propagate_map_change started.

The add_cpu, del_cpu, crash_system, run, and stop operations all consist of transmitting PCP requests of the associated type, with the arguments to the operations as data.

Several of the operations, however, are handled without the PCP's help. The VP1$get_fault_data operation is done by copying the data from the level 1 processor state block. VP1$restore_fault_data copies its argument into the current state in the fault state block. VP1$begin_atomic_operation increments the atomic operation depth in the level 1 processor state, and VP1$end_atomic_operation decrements that value. After doing the work of any of these operations, the GPP proceeds to finish the unbinding operation normally, finding the next level 1 processor to execute.

The VP1$bind, VP1$unbind, and VP1$set_processor_interrupt operations operate similarly. They all require that the level 1 processor they operate on be stopped. Consequently, they lock the L1PST lock, then test to see if the level 1 processor to be operated on is stopped. If so, the operation is performed. If not, an error status is stored in the status code of the operations. The L1PST lock is then unlocked.

The final operation to be discussed is the VP1$next_stopped operation. This operation just locks the L1PST lock, gets the next level 1 processor on the next-stopped queue, and stores its name in the return value. The L1PST lock is then unlocked.

With the exception of the await operation when it decides to send a post_await request, the instruction counter is always incremented by 1 after handling a INVOKE-LEVEL1 instruction, before finishing the unbind. This causes the instruction counter to skip over the INVOKE-LEVEL1 instruction just executed.

5.6 Implementing Level 1 Processors on Traditional Hardware

If it is not possible to have a dedicated processor to run the PCP, it is still possible to adapt this design to work. This adaptation is done by simulating the PCP on the general purpose processors that are available. Similarly, mapping the interrupts sent by I/O devices into increments on special eventcounts is not difficult. Both these ideas are discussed in the rest of the chapter, to show that the design can be easily adapted to architectures similar to the Honeywell 68/80 system that currently supports the Multics system.

5.7 Simulating the Processor Control Processor

The necessary qualities of the PCP for implementing the level 1 processor design given in this chapter are that it must have its own environment and state, and that it always must be ready when there are tasks for it to do. It must also be able to send an UNBIND signal to any other processor.

While these characteristics are true of a dedicated hardware processor, it is also possible to obtain them by other schemes. The scheme used here will be to recognize that the PCP need not always be executing. When it is not executing, its state can be represented in primary memory. The same

techniques that make processor multiplexing possible will enable simulating the PCP on a multiprocessor architecture.

The PCP's state (computational registers, descriptor segment pointer) will be stored in primary memory in a block called the PCP state block. In addition, the PCP state block will contain a lock called the PCP lock, and a flag, called the PCP-has-work flag.

Basically, we simulate the PCP by attempting to have the currently executing physical processor load the PCP state and run the PCP whenever the PCP is given more work to do, such as, for example, when a new request is entered into the PCP request queue. Some other processor may be executing in the PCP, however, so the PCP lock is used to prevent two processors from simultaneously entering the PCP. In order to enable any processor to run the PCP, each processor must be able to send UNBIND signals to all other processors. Further, when running the PCP, there must be some mechanism that prevents UNBIND signals sent to the current processor from taking effect until the processor stops executing the PCP.

The detailed algorithm executed every time something is entered into the PCP request queue is as follows. The PCP-has-work flag is set. The processor attempts to set the PCP lock. If the lock is already set, the processor continues with what it was doing; presumably it is executing some version of the unbind operation shown in the previous design, so it continues to unbind itself. If the processor succeeds in setting the lock, it then clears the

PCP-has-work flag, and loads the state from the PCP state block. When the PCP processes all of the work currently queued for it, it gives up the processor by storing its state in the PCP state block, unlocking the PCP lock, and then checking the PCP-has-work flag. If the PCP-has-work flag is on, some other processor has given more work to the PCP since the current processor started running the PCP. Consequently, the current processor tries to run the PCP, and gives up only if it finds the PCP lock already set. (1)

In order for this simulation to work, it is necessary to run the PCP in this way whenever it must do some processing. As we have seen there are three kinds of processing that the PCP does. They are handling the PCP request queue, noticing changes in special eventcounts and handling the clock, and making sure that the assignments of processors to level 1 processors is correct with respect to priority assignments. Handling the PCP request queue is simple in the simulation. We just change the algorithm for sending PCP requests to always try to run the PCP after placing a request.

Handling special eventcounts is not so simple. We would like the PCP to run relatively quickly after a special eventcount is incremented. There are three kinds of special eventcounts. The stopped eventcount is simple to handle, since it is incremented only by the PCP itself, so the PCP is always running after incrementing the stopped eventcount. The clock eventcount is less simple. If there is a way to set an alarmclock in the system that will

---

(1) The PCP-has-work flag is really a wakeup-waiting switch for the PCP, if you imagine giving up the processor by the PCP as a block.

send an UNBIND signal to some processor periodically, then the GPP can always check the current clock value at the start of the UNBIND handler to see if the PCP should be run. This solution can also handle the checking of the other special eventcounts incremented by I/O devices, since the alarmclock can be set to go off with a frequency that gives an optimal rate of polling of the special eventcounts. The major cost of simulating the PCP on the other processors of the system arises from the need to unbind processors more frequently to handle the clock.

5.8 I/O Devices That Send Interrupts

Traditionally, I/O devices send interrupts to the system to signal the completion of I/O operations. Up to this point, we have been assuming that I/O devices signalled the completion of I/O operations, or other events requiring immediate attention of a level 1 processor, by incrementing memory words that the PCP then handled as eventcounts. The PCP then reflected these changes as advances, detecting them by periodic polling.

If the more traditional method of having the I/O devices send interrupt signals to the GPPs is used, the incrementing of eventcounts can be simulated by having the interrupt handlers of the system do nothing but increment the appropriate memory words. The PCP will periodically poll these memory words, and reflect changes to them by awakening level 1 processors that await changes to those words.

Responsiveness is a question here. If the polling frequency of the PCP is controlled by a clock, as above, in order to get very fast response to I/O device signals, the polling frequency must be very high. This has a cost, in that most times the clock forces the PCP to run, there will be nothing for it to do. Consequently, the best choice is to run the clock so that it interrupts the processors only as frequently as necessary to cause the clock eventcount to work. The interrupt handlers, in addition to incrementing the eventcount associated with the device causing the interrupt, will attempt to run the PCP. This choice guarantees that when the PCP is run, it has something to do.

5.9 Summary

In this chapter I have shown how to implement level 1 processors using a structure based on a central agent. The first implementation is developed using a dedicated processor for the central agent. Then, for an implementation more suitable for traditional multiprocessor architectures, I showed how the dedicated processor can be simulated without a dedicated processor on the general-purpose processors of the system.

The simplicity of the implementation in either case derives primarily from the centralized structure. It is clear in this structure how the assignments of level 1 processors to GPPs is controlled.

Chapter Six

Level 2 Processor Interface and Implementation

The second level virtual processors are used to run user computations in the computer system. In this chapter, the interface and implementation of level 2 processors are described. The level 2 interface is quite similar to the level 1 interface, with a smaller number of operations.

There are three major differences between level 1 and level 2, however. First, since level 2 primitives are visible at the perimeter of the security kernel, protection mechanisms are very important to prevent unauthorized interference between level 2 processors. The level 2 interface is designed so that privileged information is not accessible at the interface. The authorization to use particular level 2 operations is provided by the ordinary access control mechanisms used to protect stored information.

Second, the level 2 implementation is partitioned into two parts: a fixed mechanism for multiplexing level 1 processors, and a policy mechanism that controls the rate of resource usage by the level 2 processors. The policy mechanism is designed to be modifiable by an administrator at an individual computer installation without the need to re-verify the security of data in the system.

Third, the IPCC mechanism provided at level 2 is more flexible than that of level 1. The await operation can await a larger number of eventcounts. A process interrupt facility is provided that is really just a special case of the await operation. The await operation also takes care of outward signalling eventcounts. The IPCC mechanisms are completely protected by the access control mechanisms that apply to segments containing eventcounts; there is no need for a special protection mechanism to prevent unauthorized interprocess control communication.

In this chapter, the interfaces to level 2 are discussed first. The overall structure of the implementation then is discussed, and the isolation of scheduling policy from mechanism is explained.

6.1 Level 2 Processor Interfaces

At level 2 there are two sets of operations that allow control of level 2 processors. The creation and deletion operations manage the set of level 2 processors that are in existence at any time. The IPCC operations allow communication between level 2 processors. These two sets are the only operations that are provided at the level 2 interface for the control of level 2 processors.

Some internal interfaces are important because they form the interface between the scheduling policy and the scheduling mechanism in the level 2

implementation. These interfaces are discussed later in the description of the implementation.

6.1.1 Creation and Deletion of Processors

Unlike the first level processor manager, which implements a fixed set of processors, the second level processor manager allows for creation and deletion of second level processors. This facility makes the assignment of processors to user computations much simpler -- whenever a user wants to start some process (as when he logs in to the computer system) he can just have a new processor created on which to run that process.

Initiation of a process running on a level 2 processor requires fabricating an environment for the processor to execute in, creating a level 2 processor to perform the process, and starting the level 2 processor running at a particular point in the environment. In this thesis, I assume that the environment is created and maintained outside the level 2 processor implementation, by an environment type manager. Authorization to initiate a process in a particular environment, with a particular initial execution point, is handled at a higher level in the system. Montgomery [18] has discussed a mechanism for protection of process initiation. His mechanism should be used in conjunction with my design.

The process initiation operation starts by first verifying the right of the level 2 processor invoking the kernel process initiation operation to create a process that starts with the particular initial execution point in the specified environment. This verification is done within Montgomery's model. Then, it creates an environment description (such as a Multics descriptor segment) for the specified environment, by calling on the environment description manager. Inside the security kernel, it then passes the environment description and initial execution point to the level 2 operation that creates the level 2 processor and starts it running at the initial execution point.

The level 2 operation that creates and starts a level 2 processor running in a particular environment with a particular execution point is the operation

VP2$create_processor (envptr, startptr, schedclass, procname)

This operation takes a name of an environment (envptr), a point within the environment to start executing (startptr), and a scheduling class (schedclass). It creates a level 2 processor that is named procname, and starts it running at the initial execution point. The schedclass parameter is information passed to the scheduling policy mechanism of the level 2 processor manager to control the rate of resource usage of the created processor.

Protection of level 2 processors from destruction is also at a higher level in the security kernel of the system than level 2. The level 2 operation used to destroy a level 2 processor is

VP2$destroy_processor (procname, envptr).

This operation destroys the level 2 processor named procname. The level 2 processor is not destroyed until it becomes stopped at level 1, so that any kernel operations in progress will complete. VP2$destroy_processor does not return until the processor named procname is destroyed. The environment of the processor is not destroyed by this operation. The environment ptr (envptr) is returned so that the higher level process termination operation can destroy the environment.

6.1.2 IPCC Interfaces

IPCC among level 2 processors, like IPCC among level 1 processors, is done using eventcounts. Eventcounts are implemented as words in virtual memory segments. Protection of eventcounts is accomplished by using the virtual memory protection mechanisms. An advance operation requires that the level 2 processor executing the advance have both read- and write-permission to the eventcount, while an await operation requires only read-permission.

Since segment protection is used to prevent unauthorized release of and interference with (modification of) information sent through the interprocess control communication mechanism, ensuring various security policies is simplified. To confine a level 2 processor from transmitting information to unauthorized receivers through both eventcounts and segments, one only has to restrict the set of segments it has write-permission to. If the set of segments it can write cannot be read by unauthorized receivers, then the

confinement is assured. IPCC using eventcounts does not introduce a new information channel from the confined processor, since sending information via eventcount IPCC requires advancing eventcounts, and thus modifying segments.

Similarly, a level 2 processor can be protected from unauthorized interference with its IPCC, by preventing unauthorized level 2 processors from having modify-permission to eventcounts that it awaits.

The await operation at level 2 has new functionality over the level 1 await operation. First of all, it allows waiting on outward-signalling eventcounts. Thus, the eventcounts that can be awaited by level 2 await operations are those that are advanced at level 2, and those that are in the set of specially handled outward-signalling eventcounts (advanced at level 1). Second, the number of eventcounts that can be simultaneously awaited is not restricted to a small number in level 2. A level 2 processor can await a large number of eventcounts simultaneously. The difference in the number of eventcounts that can be awaited reflects the cost of storage used in the level 1 and level 2 implementations.

The operations on eventcounts at level 2 are:

VP2$await (ec1, value1, ec2, value2, ...)

and

VP2$advance (ec).

VP2$await waits until ec_n is greater than or equal to value_n, for some pair of arguments _n_. VP2$advance advances the eventcount specified. VP2$await

requires read permission on all of its parameters. VP2$advance requires both read- and write-permission.

6.1.3 Processor Interrupts

A common feature of many operating systems is to allow a process is to receive a pseudo-interrupt when certain external things happen. For example, a user of Multics can, by hitting the attention key on his terminal, interrupt the program he is currently running. The handler for this interrupt reads commands from the terminal, allowing the user to inspect the state of the program, modify its environment, and debug the program. The user can thus stop a runaway program, which might be executing in an infinite loop, and debug it.

One way to model this processor interrupt mechanism would be to associate two level 2 processors with the user's computation. See figure 6.1. One of the level 2 processors, called the slave processor, runs the user's program, while another, called the control processor, waits for the attention key to be struck. The attention key being struck advances an eventcount associated with the attention key. The control processor then proceeds past the await, and causes the slave processor to stop (assume, hypothetically, that a level 2 processor stop operation exists). Then the control processor can read commands from the teletype and execute them, to debug the stopped slave processor. The slave processor can then be restarted (using a hypothetical

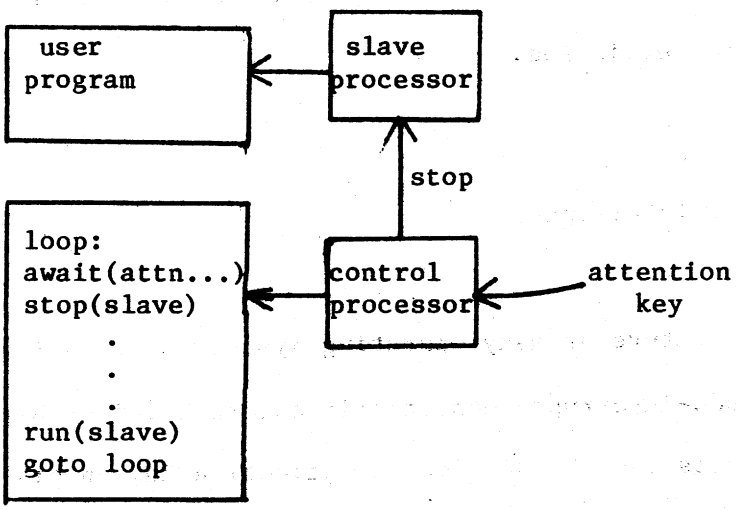


Figure 6.1
Processor Interrupt Model

level 2 run primitive), and the control process can go back to waiting for the attention key to be struck.

Directly implementing this model of processor interrupts is quite costly, since at any one time half of the level 2 processors are either awaiting an attention key to be struck, or stopped. Further, some mechanism would be needed to insure that the control processor is bound to a level 1 processor whenever its slave processor is. Otherwise, when the control processor needs to run, to stop the slave processor quickly, it can be held up if there is not a free level 1 processor to run the control processor. However, this model is useful in inventing a simple processor interrupt facility at level 2.

Instead of stopping one processor and starting another to read commands, the processor interrupt facility simply forces a fault to occur in the slave processor. The fault handler in the processor, upon determining that the

fault was a processor interrupt, will transfer to a processor interrupt handler. This processor interrupt handler can be thought of as a potential control processor that is awaiting some condition to occur. When the condition occurs the control processor is created, the slave processor is stopped, and the processor interrupt handler is executed in the control processor.

The conditions under which the processor interrupt handler will be entered are specified as if the processor interrupt handler were actually executing an await operation on a set of eventcounts. Thus, there is an operation that a level 2 processor can perform, called

VP2$set_processor_interrupt (ec1, value1, ec2, value2, ...)

The effect of this operation is as if a level 2 processor were created in the same environment, that begins by executing a VP2$await operation on the eventcount-value pairs specified, and after the await returns, calls the processor interrupt handler. (1) When the handler returns, the stopped level 2 processor will be restarted at the point where it was stopped by the interrupt. While the interrupt handler is executing, the stopped level 2 processor cannot run.

---

(1) The processor interrupt is initially received by the fault handler set up in the level 1 processor. I assume that this fault handler determines the fault type and reflects it to a set of higher level fault handlers. The fault handler for each type of fault can be changed through an interface that controls the level 1 fault handler called the fault manager. The program to be called upon a processor interrupt is specified through the fault manager interface.

Once the handler is entered, the interrupt conditions are reset, so there are no interrupts during the time the handler is deciding what to do to handle the interrupt. The handler reenables interrupts by calling VP2$set_processor_interrupt again. At any particular point in time, either no handler is set, or one has been set. Attempting to use VP2$set_processor_interrupt to set up two handlers that are invoked under different conditions causes the new handler to completely supersede the old one.

In order to interrupt a process, then, one need merely advance one of the eventcounts specified in the call to VP2$set_processor_interrupt. Having the level 2 processor itself specify the conditions under which it is to be interrupted allows protection by the access control on eventcounts against malicious attempts to send interrupts. Further, programs running on the processor can be quite flexible in choosing the set of conditions that cause processor interrupts. The clock eventcount, I/O eventcounts, or any level 2 eventcount can be made to cause an interrupt.

6.2 Structure of the Second Level Processor Manager

The level 2 processor implementation is based on a relatively centralized processor multiplexing algorithm. The multiplexing of level 1 processors among level 2 processors is done by two dedicated level 1 processors, called the unbinder and the binder/scheduler. A third dedicated level 1 processor handles outward signalling of eventcounts. Not all of the work is done by the dedicated level 1 processors, however. The creation and deletion operations are distributed in the processors that do the initiation and termination of processes. The IPCC operations are distributed among the level 2 processors, to some extent.

There are four data bases shared among the parts of the level 2 processor implementation. They are the level 2 processor table, which contains the state of each level 2 processor, the level 2 await table, which keeps track of all of the eventcounts being awaited by level 2 processors, the level 2 reschedule queue, which is a list of level 2 processors that are candidates for rescheduling, and the free level 1 processor list, that contains a list of level 1 processors that can be bound to level 2 processors.

The processors and data bases of the level 2 implementation are shown in figure 6.2. The binder/scheduler processor executes in two domains. In the binder domain, the mechanisms for binding level 2 processors to level 1

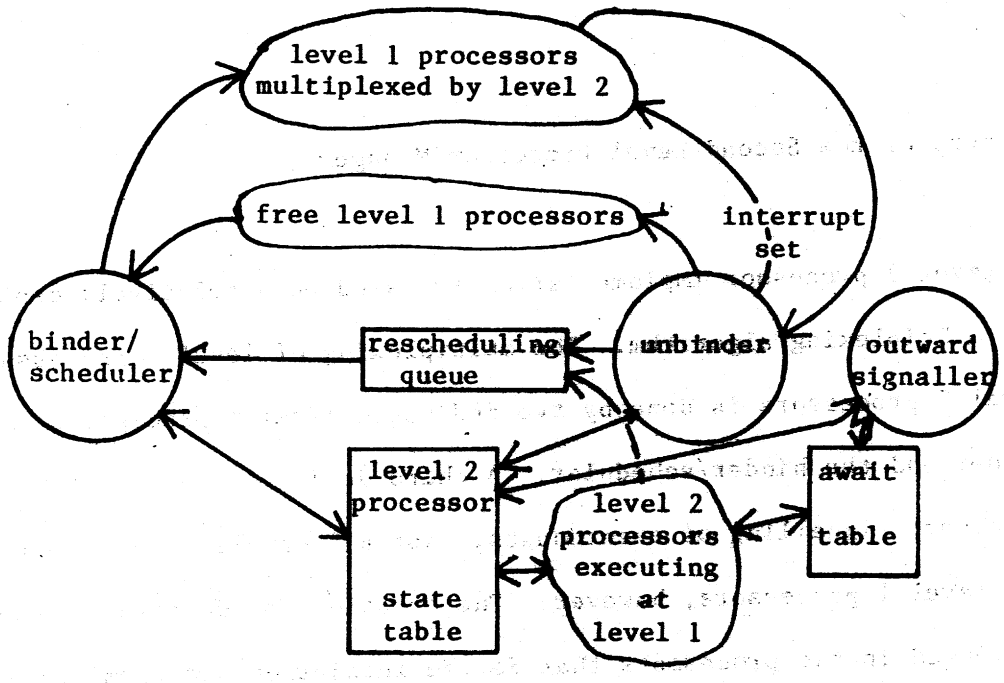


Figure 6.2
Processors and Data Bases of Level 2

processors are found. The scheduler domain is a less privileged domain that implements the particular scheduling policy for the level 2 processors. The scheduler domain can call on a small set of primitives to control the actions of the binder domain. These primitives are discussed later in this chapter. They are designed so that the scheduling policy may be written without compromising the security of the system.

6.2.1 Level 2 Data Bases

Before describing the actions of the level 1 processors that make up the level 2 implementation, I describe in more detail the four level 2 data bases. All of these data bases are protected by a single lock, called the level 2 processor lock. Waiting for the level 2 processor lock to be unlocked is done by awaiting the level 2 lock eventcount that is advanced (using VP1$advance) each time the lock is unlocked. To ensure that the level 2 operations operating under the level 2 processor lock do not deadlock, level 2 processors accessing these data bases must do so while unstoppable at level 1.

The level 2 processor table is a table containing one entry for each level 2 processor that exists. Its function is similar to the function of the level 1 processor state table. The data of the level 2 processor table is stored in a virtual memory segment.

Figure 6.3 illustrates the format of a level 2 processor table entry. Each entry of the level 2 processor table contains a state description of the level 2 processor in a format suitable for calling the VP1$bind operation. Some of the data in this description is in a different form, however. The pointer to the environment description is not a primary memory address at this level, but a name that can be presented to the environment description manager operation that places the environment description in primary memory. In

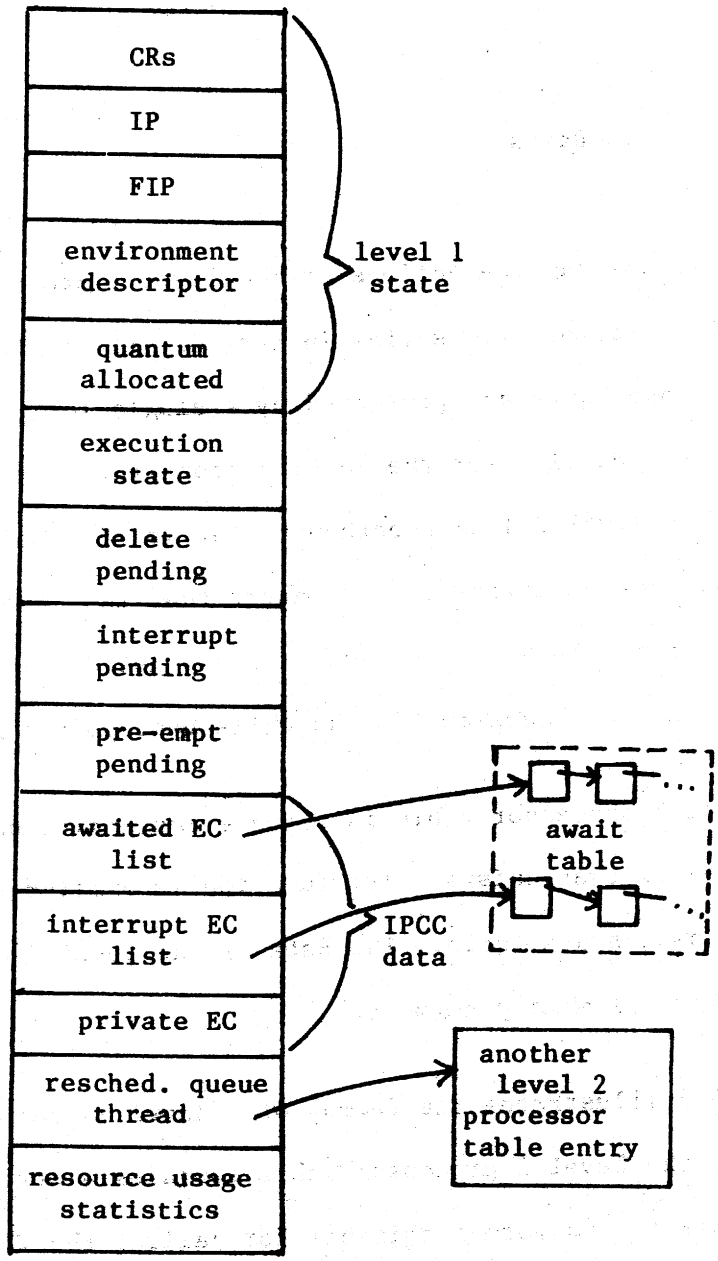


Figure 6.3
Level 2 Processor Table Entry

addition to the state description, there is a value that represents the execution state of the level 2 processor -- running on a level 1 processor,

runnable, awaiting some eventcounts (and not bound to a level 1 processor), or queued for rescheduling. Also in each entry are three flags that control the action taken by the unbinder -- delete pending, processor interrupt pending, and pre-empt pending. The level 2 processor table also has two pointers to lists in the await table, one for awaited eventcounts, and one for processor interrupt eventcounts. A private eventcount is stored in each processor table entry to be used in the await operation described shortly. Associated with each entry is a set of resource usage statistics maintained for use by the scheduling policy in making decisions.

The await table is primarily a mapping from eventcount names to level 2 processors awaiting those eventcounts. Given an eventcount name, and a value, one can inspect the await table and find all level 2 processors that should be awakened when the eventcount is advanced to the specified value. A suitable representation for the await table is shown in figure 6.4. The await table consists of an eventcount map that converts an eventcount name into a list of await table entries. Each entry on the list contains a value awaited. Entries on the list are sorted in increasing order of value awaited, so that the set of entries less than or equal to the current value of the eventcount can be found efficiently. Each entry also contains a pointer to a level 2 processor table entry that indicates the processor that is interested in this particular value of the eventcount. A flag in the entry indicates whether the entry corresponds to an eventcount being awaited by the level 2 processor, or to an eventcount used in VP2$set_processor_interrupt. Finally, all of the

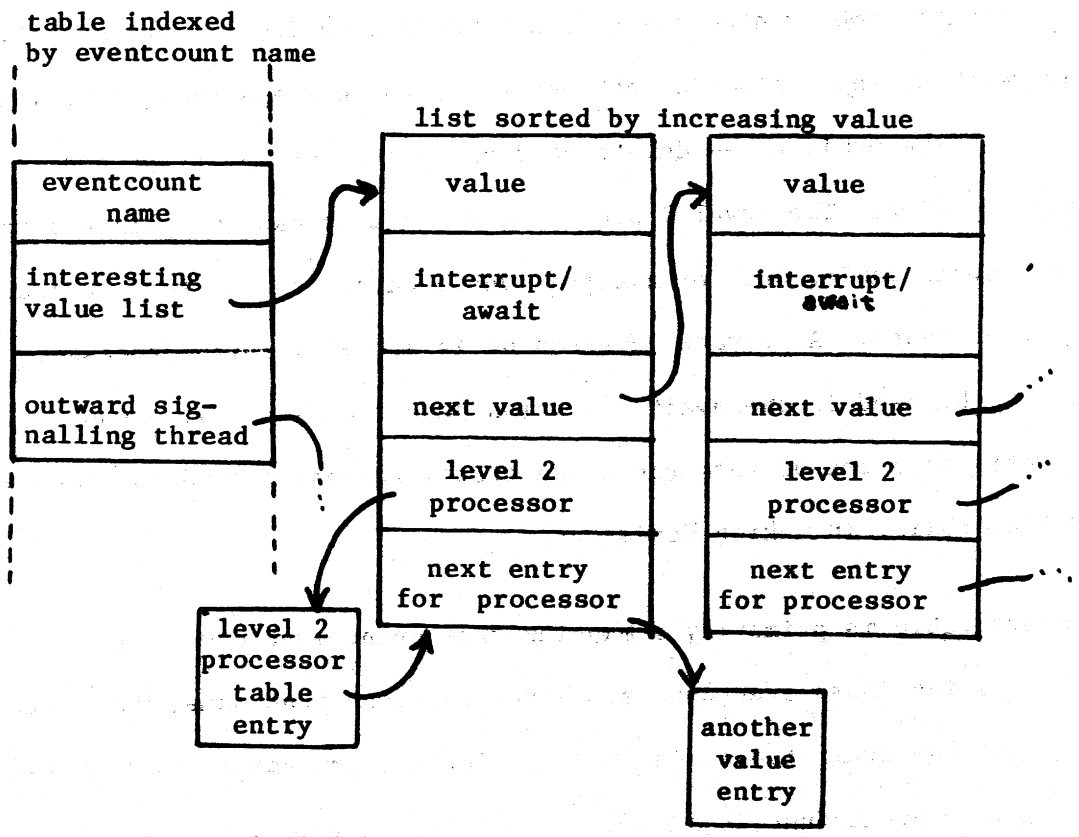


Figure 6.4
Await Table Structure

entries for a particular processor are threaded into two lists, one for awaited eventcounts, and one for processor interrupt eventcounts. All of the outward signalling eventcounts are also listed together in a special list, used by the level 2 processor that handles outward signals. The await table is stored in a virtual memory segment.

The rescheduling queue is a list of level 2 processors that are candidates for rescheduling. The level 2 processor table entries each have a thread pointer that allows level 2 processors to be threaded onto this list.

Associated with the rescheduling queue is an eventcount that is advanced each time a level 2 processor is added to the queue.

The free level 1 processor list is just a list of the level 1 processors that are free for the binder to bind level 2 processors to. Level 1 processors are added to the list each time level 2 processors are unbound from them. Binding a level 2 processor to a level 1 processor is done by selecting one of the free level 1 processors on the list, and binding to that level 1 processor. An eventcount is associated with the free level 1 processor queue. It is advanced each time a level 1 processor is placed in the free queue.

One other data base is used in the implementation, but is completely private to the scheduler domain of the binder/scheduler processor. It is called the scheduler queue, and is discussed in the description of the scheduler.

6.2.2 Processes of the Second Level Manager

The three processes that are part of the level 2 manager run on dedicated level 1 processors. Each of these processes performs one particular class of operations, waiting for a particular event to happen, then interacting with the level 1 implementation and the level 2 data bases to perform its function. They are implemented on distinct processors for two reasons -- their operation is only loosely coupled, so it would add complexity to try to specify the

order of their operations, and the tasks performed by each of these processors can proceed in parallel to a reasonable degree.

The binder/scheduler and the unbinder processors implement the bind and unbind operations of the model of processor multiplexing described in chapter

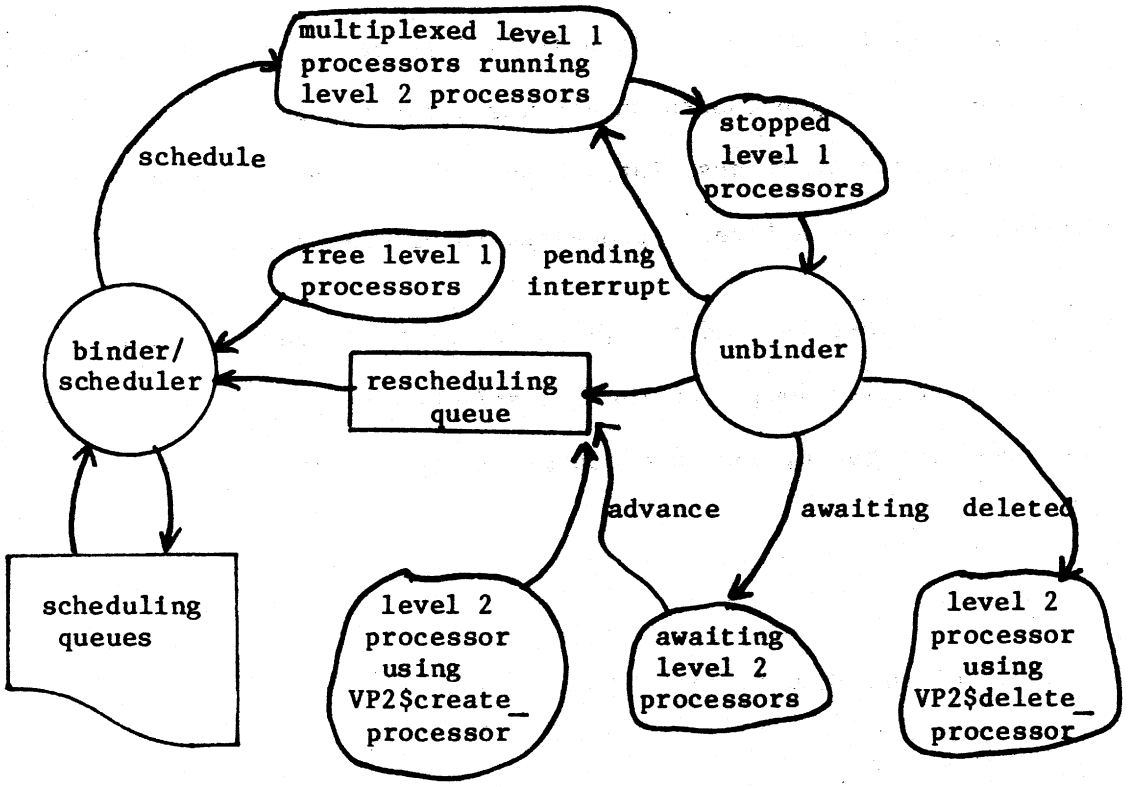


Figure 6.5
Actions of Binder/Scheduler and Unbinder

two. Figure 6.5 illustrates the actions of the binder/scheduler and the unbinder. When a level 2 processor is stopped at level 1, due to exceeding its quantum or an explicit VP1$stop operation, the unbinder processor awakens and determines what to do with the level 2 processor. It uses the

VP1$next_stopped operation to get the name of the level 1 processor, and translates this into the name of the level 2 processor that is stopped. If the level 2 processor table entry for the stopped processor indicates that a delete is pending, the unbinder performs the deletion. If a processor interrupt is pending, and rescheduling has not been explicitly requested by the scheduler, the unbinder uses VP1$set_processor_interrupt and VP1$run to cause the processor interrupt to happen. Otherwise, the level 2 processor is unbound from the level 1 processor, and placed in the rescheduling queue if it is not waiting, and marked as queued for rescheduling. If the level 2 processor is waiting, it is marked as awaiting.

The rescheduling queue is the means by which the binder/scheduler is informed of processors to be rescheduled for level 1 processors. The binder/scheduler is driven by two conditions -- the availability of free level 1 processors noted in the free level 1 processor list, and the arrival of new level 2 processors to be rescheduled. These conditions are signalled by advances of eventcounts associated with each queue. It takes each new level 2 processor that arrives in the rescheduling queue, and enters this processor into an internal data base called the scheduling queue. As level 1 processors become free, the binder/scheduler chooses the best candidates from the scheduling queue, and binds them to the free level 1 processors.

The binder/scheduler can also enforce scheduling policies that require pre-emption of level 2 processors from level 1 processors before their quantum is exceeded. Pre-emption of level 2 processors bound to level 1 processors is

achieved by marking the level 2 processor table entry as having a rescheduling requested, then using VP1$stop to stop the level 1 processor. When the level 1 processor stops, the level 2 processor will be placed in the rescheduling queue by the unbinder.

The binder/scheduler does not see level 2 processors that are awaiting eventcounts. As part of doing the corresponding advance, the level 2 processor is queued for rescheduling, from which queue the binder/scheduler can extract it. If the binder/scheduler pre-empts a level 2 processor that is awaiting, it will be unbound from the level 1 processor it is running on, but will not be placed in the rescheduling queue until the corresponding eventcount is advanced.

The third processor of the level 2 processor manager is the outward signaller. The outward signaller's job is to periodically poll the outward signalling eventcounts that are being awaited by level 2 processors. It uses the list of outward signalling eventcounts in the await table to find out the names of all the outward signalling eventcounts being awaited. It uses the outward_signals eventcount to control the frequency of its polling, as I noted in chapter three. When the polling of outward signalling eventcounts indicates that a level 2 processor should be awakened, the outward signaller awakens the level 2 processor, just as if the outward signaller had incremented the eventcount itself.

6.2.3 Eventcount Implementation

6.2.3.1 Advance

The level 2 advance operation increments the eventcount by calling on the level 1 advance operation. By using level 1 advance, level 2 solves the inward signalling problem. Any level 1 processor that is waiting on the advanced eventcount is awakened by level 1. After using level 1 advance, the level 2 advance operation determines the level 2 processors that must be awakened (if awaiting) or sent a processor interrupt (if the advanced eventcount is part of the processor's processor interrupt condition).

Finding the level 2 processors affected by an advance and performing the required awakening and setting interrupts is done by an operation that is internal to the level 2 implementation, called WAKEN. The WAKEN operation takes the name of the eventcount and its current value as input. WAKEN then uses the await table to find all level 2 processors that are to be awakened and interrupted. The WAKEN primitive is also used by the outward signaller processor to reflect all of the outward signalled eventcounts.

The level 2 await operation actually waits by using the level 1 await operation. Since level 2 can await a large number of eventcounts simultaneously, some method must be used to reduce the number of eventcounts awaited at level 1. The reduction is accomplished by associating with each level 2 processor a private eventcount that is advanced by the level 2 WAKEN operation to actually awaken the associated level 2 processor. The level 2 await operation actually waits at level 1 by awaiting a change to the private eventcount of the waiting level 2 processor.

The WAKEN primitive actually awakens a level 2 processor in three steps. First, all of the await table entries on the awaited eventcount list for the level 2 processor are deleted from the await table. Further advances on the private eventcount are prevented, since no await table entry for the processor will be found. Second, it advances the private eventcount. If the level 2 processor is bound to level 1, this will cause it to run. Third, if the level 2 processor is not bound to a level 1 processor, its state is changed to queued for rescheduling, and it is threaded onto the rescheduling queue so that the binder/scheduler sees it.

The WAKEN operation also causes processor interrupts to happen. Await table entries that are to cause processor interrupts are specially flagged. The WAKEN operation causes the interrupt to occur in three steps. First, the list of await table entries associated with the level 2 processor interrupt is deleted from the await table. This prevents further interrupts from being set. Second, the level 2 processor table entry is flagged as having a pending

processor interrupt. Third, if the level 2 processor is currently bound to a level 1 processor, the level 1 processor is stopped, using VP1$stop, and otherwise, the level 2 processor is marked as queued for rescheduling and is placed on the rescheduling queue. If the processor is running at level 1, when it stops the processor interrupt will be set by the unbinder processor. Otherwise, when the binder/scheduler binds the processor to level 1, it will use VP1$set_processor_interrupt to set the interrupt.

6.2.3.2 Await

The level 2 await operation works by locking the level 2 processor state lock, then checking the eventcounts and obtaining their system-wide unique names. If any of the eventcounts is greater than or equal to the corresponding value, the processor state table is unlocked, and the await operation returns. (1) Entries are made in the await table for each eventcount-value pair, and the current value of the level 2 processor's private eventcount is obtained. Then the state table lock is unlocked, and the level 2 processor executes a VP1$await on the private eventcount, for the next higher value of the eventcount.

---

(1) If a fault (other than a fault handled transparently below level 2, such as a missing page fault) occurs while accessing any eventcount (such as no access to read the eventcount), the state table lock is unlocked and the fault is reflected. When the fault is restarted, the lock will be relocked, and the await operation starts from the beginning again.

A processor interrupt can occur during the await operation at level 1. It is desirable to allow processor interrupts to occur during level 2 awaits, so that a user can interrupt his program if by mistake an await is executed that never will finish. The interrupt handler can await also. Because the interrupt handler shares the same awaited eventcount list and private eventcount at level 2, there must be some way that the interrupt handler can be allowed to use level 2 await, while ensuring that when the interrupted await is restarted it works correctly.

To solve the problem of the interrupted await, I modify the basic level 2 advance and await algorithms slightly. Essentially, the effect of my modification is that restarting an interrupted await causes the await to be re-executed from the beginning.

The WAKEN primitive, in interrupting a level 2 processor that is awaiting (it has an associated await list) does two extra things. First, the await table entries for all eventcounts on the interrupted processor's awated event list are deleted from the await table. Second, the private eventcount of the interrupted processor is advanced. Advancing the private eventcount ensures that the level 1 await operation in the level 2 await will return.

The level 2 await operation must check the eventcount and value parameters a second time after the level 1 await returns, because the level 1 await can return for one of two reasons now. One reason, of course, is that the level 2 await is over -- in this case, one of the eventcounts will be

greater than or equal to the awaited value, and the level 2 await operation will return to its caller. The other reason is that the await was interrupted by a processor interrupt. If none of the eventcounts is greater than or equal to the awaited value, the await must be restarted by re-entering the events in the await table, getting the private eventcount value, and awating the private eventcount at level 1.

6.2.3.3 Set_processor_interrupt

The VP2$set_processor_interrupt operation works similarly to await. The state table is locked, and each eventcount is checked and its system-wide name is obtained. If any eventcount exceeds its corresponding value, the state table lock is unlocked, and the processor interrupt pending flag is set. The level 2 processor then executes a VP1$stop operation on itself. (1) If every eventcount is less than the corresponding value, then the processor state table lock is unlocked and the set_processor_interrupt operation returns.

6.2.3.4 Outward Signalling

As noted briefly above, the outward signaller handles outward signalling eventcounts. Whenever a level 2 processor awaits or sets an interrupt

---

(1) Rather than simulating the fault, the mechanism in the unbinder is used to cause the processor interrupt for simplicity.

condition that involves an outward signalling eventcount, that eventcount is threaded onto a special list in the await table, called the outward signalling list. The outward signaller periodically takes this list of eventcounts and obtains the values of all outward signalling eventcounts on the list. Then, it uses the WAKEN interface to cause the level 2 processors interested in the outward signalling eventcounts to wake up or be interrupted.

6.2.4 Scheduling Policy

In a real computer system installation, there are many requirements on the the allocation of resources to individual user computations over time that cannot be predicted in advance by the system builder. Consequently, the system builder would like to provide for some flexibility in the resource allocation policies he builds into the system.

For this reason, the second level processor manager would like to provide an interface by which the administrator can control its resource allocation policies. The most general mechanism is to allow the administrator to write the program that makes the scheduling decisions for the second level processor manager. In the second level processor manager, this mechanism is provided for in a clean manner.

We would like the policy mechanism to be modified by the system administrator only in such ways that are safe. It would be unreasonable if by introducing a slight bug in the resource allocation policy, the system's data integrity and security could be compromised. Consequently, it is necessary to encapsulate the administrator's policy control program in an environment of the least privilege necessary to do the tasks required.

Obviously, the resource allocation policy mechanism can, if malicious or incorrect, deny resources to computations that can legitimately proceed. By allowing the administrator to write such a program, then, we place the capability for denial of service in his hands.

Through denial of service, or slowdown of service, of course, the resource allocation policy has a subtle channel of communication with all of the processes it controls. This can lead to unauthorized release of information. However, to use these subtle channels requires much more than a simple mistake on the administrator's part. So assuming the administrator is not malicious, we can provide a degree of protection against unauthorized release of information through this path.

The mechanism provided is implemented as a domain in the binder/scheduler processor, called the scheduler domain. Encapsulated in the scheduler domain, which only has access rights to call certain level 2 processor management primitives will be the scheduling policy algorithm. The scheduling policy algorithm will await an event of interest, such as the availability of a free

level 1 processor or the arrival of a new level 2 processor in the rescheduling queue. The policy algorithm will then incorporate the new knowledge into its policy and make scheduling decisions that it will accomplish by calling on an interface that causes selected level 2 processors to be bound to free level 1 processors.

There are three basic primitives available to the resource allocation policy process. The first one, schedule, allows the process to name a level 2 processor to be bound to a free level 1 processor and to specify a quantum of resources. The level 2 processor will be assigned to a level 1 processor if there is a free one, and the quantum for the level 1 processor will be set from the specified value. The second primitive, next-rescheduling, extracts the next level 2 processor from the rescheduling queue. It returns the name of the level 2 processor, and a summary of its resource usage information on which a scheduling decision can be based. The third primitive, pre-empt, allows the scheduling policy to pre-empt a level 2 processor already bound to a level 1 processor. The pre-empt primitive marks the level 2 processor as having a pending pre-emption, and if the level 1 processor is bound to level 1 it uses VP1$stop to stop it from running. The unbinder processor notices this flag, and puts such a processor in the rescheduling queue. The flag is reset when the processor is placed in the rescheduling queue.

Very simple checking ensures that the policy algorithm does not make incorrect use of the level 1 and level 2 processor resources. The schedule primitive makes sure that a level 2 processor of the specified name exists and

is not currently assigned to a level 1 processor. It ensures that the important data bases associated with the level 2 processor environment description (e.g., descriptor segment) are in core to make sure that the level 2 processor addresses memory correctly. It also ensures that the process is runnable and not waiting for some eventcount implemented at level 2. Similarly, the unbinding of a level 2 processor and deallocation of in-core resources, etc. is carried out outside of the domain of the scheduling policy algorithm, in the unbinder processor.

With the 3 operations that the scheduler domain uses to control scheduling, it can implement almost any policy, without the possibility of a bug in the policy algorithm interfering with the operations of the level 2 processors being controlled by the policy (except by denying service). This is accomplished primarily by storing the sensitive data about processes being scheduled outside the domain of the scheduler. The sensitive data contained in the level 2 processor state, etc. cannot be read or modified by the schedule, next-reschedule, and pre-empt primitives.

It should be noted that the resource allocation policy process runs in a level 1 processor, rather than a level 2 processor. This is necessary, in order to prevent the resource allocation policy from having to schedule itself.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Chapter Seven

Using Level 1 Processors in the Operating System

The level 1 processors provided by the level 1 processor manager are very useful tools for structuring the kernel of an operating system. They can be used wherever a scarce resource is multiplexed among a group of users of the system to control the multiplexing. Level 1 processors can be used to manage multiplexed I/O devices, the virtual memory, and even scarce resources being managed by the abstract type managers of the kernel.

The isolation of environment and control point that level 1 processors provide can be very useful in ensuring that parts of the system execute with the least privileges necessary to accomplish the task. Putting I/O device management in level 1 processors rather than interrupt handlers that execute in any level 1 processor environment is an example where using level 1 processors can reduce the privileges needed by parts of the kernel.

Using concurrently executing level 1 processors to implement uncoupled or loosely coupled algorithms also simplifies specification of the kernel. There is no need to specify a particular order of operations where that order is irrelevant to the tasks of distinct modules. Overspecification of the system can lead to extra complexity, possible deadlocks, and more difficult proof.

Finally, using level 1 processors to perform a particular task in the kernel assures that there is always an agent capable of performing a task when it needs to be done. For example, a virtual processor dedicated to handling missing page faults generated in I/O processors will allow the I/O processors to deal with virtual rather than real memory, and thus simplify the task of interfacing user computations to I/O devices.

7.1 Permanently Bound Processes

Processes that implement parts of the kernel algorithms are best implemented as computations that run on dedicated level 1 processors. There are a fixed, relatively small number of such processes. These processes manage shared resources, and can cause bottlenecks in the system resulting in denial of service to users if they are not scheduled properly. Most such processes provide functions that must be correct in order for the second level of processor multiplexing to work. For these reasons, the processes used in the kernel of an operating system with two levels of processor multiplexing will permanently bound to level 1 processors.

7.2 I/O Device Management

In traditional operating systems such as Multics, the operations of asynchronously running I/O channels are controlled by interrupt handlers. Such interrupt handlers are invoked on the real processor, and execute in the environment of whatever process was executing on the processor at the time. This has two bad effects from the point of view of containing the effect of bugs in the system. First of all, the interrupt handler, which may be quite lengthy, has access to manipulate anything in the environment of the interrupted process. If the interrupt handler has a bug, it may inadvertently read or modify data that is not relevant to the reason for the interrupt. The interrupt handler thus has more privilege than needed for its task, and violates the principle of least privilege [26]. Just as the interrupt handler has access to the data of the process, it also has control of the execution point, and may arbitrarily delay the interrupted process, although the process may perfectly reasonably execute on another processor.

The other problem is that the existence of interrupt handlers forces complex structures in the non-interrupt code of the system. First of all, all processes must execute in environments that have sufficient access privileges for all of the interrupt handlers of the system. This is the other side of the violation of the principle of least privilege mentioned above. All

processes thus possess privileges to access a large number of shared data bases that they normally would have no need to access. This large amount of shared data is potentially a shared information channel between processes, at least, and may contain information, such as typed passwords in I/O buffers that can contribute to sabotage of the system if misused.

The parasitic nature of interrupt handler control points also forces processes to use unnatural control structures. Since the interrupt handler has no state of its own, it cannot wait for another process to complete its action. Waiting could cause a deadlock if the process waited for is the one that the interrupt handler is executing in. For this reason, all processes that interact with data shared with interrupt handlers must never lock such shared data unless provision is made to make sure the interrupt handler does not interrupt the process doing the locking. This requirement makes handling of I/O require unreasonably complex algorithms.

For these reasons, it is quite useful to associate kernel processes with each I/O device. A device's kernel processor can await the eventcount advanced by the device to determine when the device needs service. Only the kernel process associated with a device need have privileges to manipulate that device's buffers, mailboxes, or other device specific control data. This reduces the privileges available to ordinary processes running user computations. Further, the kernel device process need only have privileges to resources that are needed to do the job of handling the device. The kernel device process need not access any user data; its interface to the user can be

through a single shared queue object. Thus both the ordinary process, and the computations associated with handling a device have reduced privileges if the I/O device management is implemented in a process.

The control structure of the device manager and user can also be much simplified. The simplification results from the fact that the communication is now symmetric; both the user and the device manager are running on different processors, and each can communicate with and wait for the other in the same way. No process is held up from executing because it handled the interrupt even though there are free processors. Further, independent device manager processes can be executing simultaneously, whereas in the interrupt scheme, this is hard to achieve without increasing the complexity of the interrupt structure of the system. Using level 1 processors for device management can succeed in smoothing the load of device management over all processing units available to the system.

The performance implications of running I/O management algorithms in level 1 processors are likely to be good. The difference between running a computation at interrupt level in a real processor, and scheduling a level 1 processor that has a higher priority than some currently executing level 1 processor, is that in the interrupt scheme, the state of the running process is stored and reloaded once per interrupt. In the process oriented scheme, in order to get the device manager to run, the process state must be stored, and the device manager's state loaded; when the device manger reaches a waiting point, its state will be stored, and the old process's state reloaded. Thus

there will be twice as much saving and loading of states in the process scheme.

If this were the only effect, there would obviously be a performance degradation. However, there are other effects that very likely will balance or overcome this defect. First of all, the device manager process now has a state that the interrupt handler had to encode in some way in its associated data bases. This state specifies what the handler is to do next, so it is not necessary to program the device manager to interpretively determine the meaning of the most recent I/O signal. If taken advantage of, the state information can replace the information used by the device manager to keep track of what it is doing. Another improvement is that complicated, expensive locking and masking algorithms need not be used in the process scheme for communication between the device manager and the user computation. Such algorithms require both computation time, and memory resources in the kernel. Consequently removing the need for such algorithms can improve performance.

In sum, then, if the cost of saving and restoring a process state is comparable to the cost of maintaining the state of the I/O connection between interrupts, then there probably will be a net performance gain resulting from removing complexity from kernel algorithms.

7.3 Kernel Type Managers as Processes

There are a similar set of problems associated with the implementation of kernel type managers as subroutines callable by user processes. We have discussed these in chapter three, but I will mention them briefly again.

First of all, without a domain mechanism that allows the user computation and kernel to be mutually protected, a kernel type manager executing in a user's process will have access to all of the user's data. It thus operates with more privilege than necessary. If the type managers of the kernel are all protected from the user but there is no domain mechanism within the kernel, the kernel domain in any user processor must have access to all data needed by kernel type managers available to that process. While it is possible with domains to restrict the accessibility of such data, and to restrict the access rights of abstract type managers to user data, having the kernel type managers execute in each user process still requires that each user's address space contain all of the domains in the kernel. If the address space is maintained in a per-process object such as a descriptor segment in Multics, then many copies of the same data will exist and must be kept up to date.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

By structuring the abstract type managers in separate processes, each abstract type manager need only have in its environment those objects with which the manager must transact. This both simplifies the structure of each abstract type manager's environment, and eliminates the need for a separate domain construct, with its additional complexity of implementation.

Implementing the kernel type managers in separate processes can lead to simplification of the part of the kernel that manages the environment descriptions of processes. When kernel type managers are implemented in a distinct domain of a process that executes user algorithms, the operations that the user code uses to manipulate its environment description must ensure that the manipulations done do not interfere with the part of the environment used by the kernel type managers. Thus the kernel algorithms depend on the environment manager, so the environment manager must be at a very low level in the kernel. By separating out the kernel type managers into separate processes, they may be executed in fixed environments that are not manipulated by the environment manager. The environment manager can then be implemented at a higher level in the kernel.

Implementing an kernel type manager in a separate process also protects the execution point of the kernel type manager from the resource controls on the user processes. In chapter 3, we have discussed how this can help guarantee that the kernel type manager never stops executing in the middle of an operation. The proportion of the time during which an ordinary user process cannot be interrupted can thus be reduced.

A reason that we have not yet discussed for putting kernel type managers in separate processes is to provide the facilities of the type manager to computations executing on dissimilar processors. Suppose we have several kinds of specialized processors on the system for various functions such as handling special I/O channels, or performing specialized computations such as Fast Fourier transforms or associative searches. A simple way to pass data to such processors is through shared data objects in the virtual memory. To have a very specialized processor perform the virtual memory operations itself upon encountering a missing page or missing segment fault is probably impossible or unnecessarily complex. The part of the kernel type manager that actually handles a missing page can be easily invoked by such a specialized processor if the page fault handling is implemented in an independent, dedicated virtual processor. If it is normally done by code in each ordinary process, then some special case mechanism must be used to handle page faults in a specialized processor, with the result that the special case mechanism may not interface correctly to the normal mechanism. Having two mechanisms to perform the same action is probably always a bad idea in designing a system.

7.4 Explicit Recognition of Parallelism in the System Design

In an operating system like Multics, there are many operations that are carried out in the security kernel of the system that do not require a particular order of execution. An example of this is the page replacement algorithm in the virtual memory. The page replacement algorithm operates by choosing candidate pages in primary memory to move from primary to secondary memory. The pages are then removed from primary memory. The removal of pages from memory must anticipate the demand for space in primary memory for new pages, because removal of pages that have been modified while stored in primary memory requires an operation to write the data in the page to secondary memory. This operation can proceed in parallel with the use of other pages in memory. In order to efficiently free up pages in primary memory, a process that is only loosely coupled to the executing user computations must constantly keep ahead of the user computations, writing out the data in pages that look like good candidates for removal.

If there is not an independent kernel process that does this lookahead, the page fault handler in each user computation must periodically do some lookahead, so that writing of pages is ahead of reading of pages into memory most of the time. Choosing the right point in time to do this lookahead (before reading the page in, or after?) and the right frequency of executing

the lookahead algorithm (every page fault or every third one?) as well as the right amount of lookahead to do each time the lookahead algorithm is entered (depends on the queuing facilities available for writes, the average frequency of reads, and other factors) can be quite complex. The complexity of these choices arises from the artificial constraint that the page removal algorithm must be in lock-step synchronization with the handling of page faults, contrasted with the basic requirement that the page removal algorithm must run ahead of the page fault handling for efficiency. Most of this complexity has been removed in a design proposed by Huber [10], by putting the page removal algorithm in its own processor. The page removal algorithm then can be relatively autonomous in its choice of how far to look ahead and how fast.

There are many algorithms in operating systems that are only loosely coupled with user-requested operations. In Multics, such algorithms include managing the paging pool (as in the example), managing the in-core copies of page maps, moving data coming into the system on I/O devices and stored in primary memory buffers into secondary memory, and updating the accounting records stored in the virtual memory from accounting variables stored in the primary memory by kernel type managers below the virtual memory level of the kernel.

7.5 Resulting Structure

The result of carrying out the structuring specified in this section will be to create an operating system in which the kernel is made up of a set of processes, each associated with a particular physical resource or shared abstract resource. These processes will all be implemented on dedicated level 1 processors, where the environment of the virtual processor is configured to exactly conform to the environment needed by the process. For example, the disk manager process will have an environment that includes only the wired-down disk accessing code and data bases, and a wired-down message queue with which it communicates to the virtual memory systems that control the reading and writing of disk pages. The manager of the page data type will have access to the disk queue, and wired-down page tables that it manages. It will be controlled by a queue of requests provided by user processes that take page faults, or by the segment manager, which may need to create or delete pages.

A non-exhaustive list of algorithms of the Multics system that would benefit from being implemented on a dedicated level 1 processor follows.

1. Device management (currently done by interrupt handlers). One level 1 processor for each I/O channel.
2. Page removal algorithm. (Designed by Huber [10])
3. Page fault handler. Havig this processor would allow I/O devices to access virtual memory as described earlier.
4. Environment descriptor manager. In the environment of the environment descriptor manager, each environment descriptor could

be known as a data segment. Thus manipulation of environments of all user processes, needed to handle revocation of access to and simultaneous sharing of environments is only done by one process.

5. System debugger. In Multics, the state of a crashed system is inspected by a stand-alone program that is loaded on a crash into the memory. An alternative would be to design it as a level 1 processor that awaits an eventcount that is advanced by a crash. Since level 1 is fairly simple, and is the bottom level of the system, it should rarely be the case that a system crash causes the implementation of level 1 to fail. The system debugger can then be designed in an environment where parallelism works.
6. Page table removal algorithm. For the same reasons that I pointed out for the page removal algorithm, removal from primary memory of page tables for segments is simplified by decoupling it from operations explicitly called by user algorithms.
7. Salvaging of directories. Currently two separate mechanisms handle salvaging the data in directories if the data is discovered to be inconsistent. One mechanism is a stand-alone program run by the system debugger while the system is crashed. The other is a part of the kernel that is invoked when a direcory manager operation discovers that the directory being manipulated is inconsistent. These mechanisms could be merged into a program that runs on a dedicated level 1 processor that awaits requests to salvage directories. Like the system debugger, this program could still run, even if most of the higher level programs have stopped due to software failure.
8. Consistency checker. A processor could periodically check the consistency of important system data bases, in the hope of catching trouble before other software encounters it. For example, a process could check to see that two distinct pages were not assigned to the same disk block.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Chapter Eight

Conclusions and Suggestions for Further Research

To sum up the research described in the thesis, I first would like to put in capsule form the major insights I have found in the progress of the research. Then, I present a number of topics that I have not had the opportunity to investigate fully, but which definitely deserve further investigation.

The technique used to disentangle the virtual memory - virtual processor mutual dependency was to break up the virtual processor implementation into two levels, the first of which provided no new memory accessing capability and could be used to provide processing power to the algorithms that implemented the virtual memory. This technique is a special case of a method Parnas has recently called "sandwiching" [22], that in general allows elimination of mutual dependencies between two modules, A and B, by splitting A into two pieces so that the functionality B depends on is in the lower level of A, while of the two pieces, only the higher level of A depends on the functionality provided by B.

In developing a design for the two levels of the virtual processor implementation, I have avoided introducing new mutual dependencies between either of the levels of virtual processors and the virtual memory. In the case of the virtual memory - virtual processor mutual dependency, then, the sandwiching technique has been successful in practice, as well as in theory.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

The use of abstract type managers as a metaphor for describing the two level virtual processor hierarchy has given an unexpected dividend in showing that the cache management pattern of type extension first developed by Janson [11] can be used to describe the structure of processor multiplexing algorithms as well as the virtual memory implementation. The cache management pattern is a basic pattern in the design of operating systems because operating systems create abstract types as tools to manage scarce resources. As far as I know, the use of types as tools to manage scarce resources is not yet well understood. However, the cache management pattern seems to play a quite important role in using abstract types to describe the implementation of operating systems.

In the design of both levels, a certain degree of simplicity arises from centralizing the mechanism that does the actual multiplexing of processors in one or more dedicated processors. As I have shown in the latter part of chapter five, it is fairly easy to take a design that uses a centralized control and convert it into a design that has distributed control. The inverse transformation is not easy, however. An algorithm initially designed to be distributed on the processors being multiplexed, such as that presented by Bredt and Saxena [2], tends not to be as clear because the legitimate orderings of actions taken by the distributed algorithm is not directly represented in the algorithms.

The use of eventcounts for IPCC in the design has had two effects. First, protection of information transmitted by the IPCC mechanism is guaranteed by the virtual memory protection mechanism. This eliminates the need for a special access control mechanism on IPCC that would make the implementation of the IPCC mechanism more complex. Second, because eventcounts are simply words in the virtual memory, the same semantics apply to the IPCC mechanisms provided at both levels of virtual processor implementation. Further, because the storage for eventcounts is provided by the memory, the same eventcount can be used by processors implemented at different levels, allowing inward and outward signalling. Providing semaphores as the basic IPCC mechanism seems to preclude outward signalling. In Bredt and Saxena's design [2], which provides semaphores, it is required that a level 2 processor that takes a page fault remain bound to the level 1 processor until the page fault is satisfied. In my design, a level 2 processor that takes a page fault can wait for the page fault to be satisfied using level 2 await, and be unbound in the interim.

An important part of the design of the second level was providing an administratively variable policy mechanism that could be varied arbitrarily without compromising the correct operation of the kernel of the operating system. While the mechanism proposed does not prevent denial of service to users, the policy algorithm is run in an environment containing only the privileges needed to make scheduling decisions. The actual integrity of the virtual processors being scheduled and the data that they operate on cannot be

compromised by the scheduling policy mechanism. In part, the policy mechanism was easy to include in the design because the processes used to perform kernel functions are protected from the policy mechanism by being permanently bound to level 1 processors.

The design developed in the thesis has, serendipitously, allowed the kernel to be constructed as a set of cooperating parallel processes. Just as decomposing the kernel into a set of modules that can be independently understood and verified is aided by using abstract types, decomposing the kernel into a set of loosely coupled or uncoupled parallel processes is a tool that allows designing and verifying small pieces of the system independently, because only the essential ordering constraints are specified in the design.

Further Research Topics

In this thesis, I have proposed a fairly detailed design for two levels of processor multiplexing, and a much less detailed sketch of how the rest of the system could be structured around the two levels. A very important step in proving my results is the actual implementation of the two level processor multiplexing design. Further, there is certainly much to be done in actually structuring the design of an operating system such as Multics in terms of dedicated virtual processors. Huber [10] has taken the first step in this direction by designing and implementing a version of Multics page control that runs in several dedicated Multics processes. However, using the level 1 processors of my design to replace the interrupt handlers used to manage I/O

devices in systems like Multics promises to provide a great deal of simplification. Some of the other suggestions for using processors made in chapter seven seem to have promise also.

An important reason for actually implementing the two level design is to verify that the two level design does not reduce the performance of the system. I have given a brief argument in chapter three to show that performance is not necessarily reduced, but only an actual implementation that has good performance can actually prove that performance is not a problem.

In chapter five, I proposed a non-traditional computer architecture that uses a dedicated microprocessor to control the short-term multiprogramming of a multi-processor system. Actually constructing such hardware can simplify both the hardware and software structure of a computer system, by eliminating the need for complex interprocessor control mechanisms, such as interrupts. In chapter five, the actions taken by the general purpose processors was implemented by software. It seems to me that a hardware implementation of the algorithms in the general purpose processor that implement level 1 functions would greatly simplify and improve the performance of the system. Such an implementation seems quite feasible for a microprogrammed general purpose processor.

A final topic that requires more study is the relationship between type managers and interpreters. The interpreter for each type manager in the system is the real processor. The algorithms for all type managers are

expressed in terms of instructions that are executed on the real processor. At the abstract level, though, each type manager can be viewed as an interpreter for the operations on the type. Viewing the type managers as algorithms to be executed on real processors is essential for developing a design that is actually implementable on a small number of real processors. Processor multiplexing can be viewed as a mechanism for ensuring that the real processor resources get distributed to all type managers that need such resources. On the other hand, viewing each type manager as an interpreter of its own operations seems to be much simpler. The relationship between these two views in the design and implementation of systems deserves more study.

BIBLIOGRAPHY

[1] Bobrow, D., et al., "TENEX - A Paged Time Sharing System for the PDP-10," CACM 15, 3 (March 1972), pp. 135-143.

[2] Bredt, T., and Saxena, A., "A Structured Specification of a Hierarchical Operating System," Proceedings of the International Conference on Reliable Software, Los Angeles, April 1975.

[3] Brinch-Hansen, P., "The Nucleus of a Multiprogramming System," CACM 13, 4 (April 1970), pp.238-41.

[4] Dahl, O.J., Myrhaug, B. and Nygaard, K., The Simula/67 Common Base Language, Publication S-22, Norwegian Computing Center, Oslo, 1970.

[5] Dennis, J., "Concurrency in Software Systems," Computation Structures Group Memo 65-1, M.I.T. Project MAC, June 1972.

[6] Dijkstra, E.W., "Cooperating Sequential Processes," in Programming Languages (F. Genuys, ed.) Academic Press 1968, pp.43-112.

[7] Dijkstra, E.W., "The Structure of the 'THE' Multiprogramming System," CACM 11, 5 (May 1968), pp.341-46.

[8] Field, M.S., "Multi-Access Systems -- The Virtual Machine Approach," Cambridge Scientific Center Report 320-2033, IBM Corporation, Cambridge, Mass. (September 1968).

[9] Hoare, A., "A Structured Paging System," Computer Journal 16, 3 (August 1973), 209-15.

[10] Huber, A., "A Multiprocess Design of a Paging System," S.M. Thesis, M.I.T. Department of Electrical Engineering and Computer Science, May 1976 (to be published as an M.I.T. Laboratory for Computer Science Technical Report)

[11] Janson, P., "Using Type Extension to Organize Virtual Memory Mechanisms,", Ph.D. thesis in preparation, M.I.T. Department of Electrical Engineering and Computer Science (expected completion, August 1976).

[12] Kanodia, R., and Reed, D.P., "Eventcounts: A Model for Process Synchronization," in preparation.

[13] Liskov, B., "An Introduction to CLU," Computation Structures Group Memo 136, M.I.T. Laboratory for Computer Science, February 1976 (to be

published in the ALGOL Bulletin).

[14] Liskov, B., and Zilles, S., "Programming with Abstract Data Types," Proceedings of the ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9 (April 1974), pp. 50-59.

[15] Luniewski, A.L., "A Certifiable System Initialization Mechanism," S.M Thesis in progress, M.I.T. Laboratory for Computer Science.

[16] McKenzie, A. "Host/Host Protocol for the ARPA Network," ARPA Network Current Network Protocols, Network Information Center, Augmentation Research Center, Stanford Research Institute, Menlo Park, Ca. (NIC 8246, Jan. 1972).

[17] Meyer, R. and Seawright, L., "A Virtual Machine Time-sharing System," IBM Systems Journal 9, 3, pp. 199-218 (1970).

[18] Montgomery, W. A., "A Secure and Flexible Model for Secure Process Initiation in a Computer Utility," S.M. and E.E. thesis, M.I.T. Department of Electrical Engineering and Computer Science (May 1976); to be published as an M.I.T. Laboratory for Computer Science Technical Report.

[19] Saltzer, J.H., "Introduction to Multics," M.I.T. Project MAC Technical Report TR-123, 1974.

[20] Neumann, P.G., et al., "A Provably Secure Operating System," Final Report of SRI Project 2581, Stanford Research Institute, Menlo Park, CA., 1975.

[21] Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules," CACM 15, 12, December 1972, pp.1053-8.

[22] Parnas, D. "Some Hypotheses About the 'Uses' Hierarchy for Operating Systems," Fachbereich Informatik, Technische Hochschule Darmstadt, Forschungsbericht BS 76/1.

[23] Rappaport, R., "Implementing Multi-Process Primitives in a Multiplexed Computer System," S.M. Thesis, M.I.T.; M.I.T. Project MAC Technical Report TR-55.

[24] Rowe, L.A., "The Distributed Computing Operating System," University of California at Irvine Department of Information and Computer Science, Technical Report 66.

[25] Saltzer, J.H., "Traffic Control in a Multiplexed Computer System," Sc.D. Thesis, M.I.T., M.I.T. Project MAC Technical Report TR-30.

[26] Saltzer, J.H., and Schroeder, M.D., "The Protection of Information in Computer Systems," Proc. IEEE 63, 9, pp. 1278-1308 (Sept. 1975).

[27] Schell, R., "Dynamic Reconfiguration in a Modular Computer System," Ph.D. thesis, M.I.T., M.I.T. Project MAC Technical Report TR-86.

[28] Schroeder, M.D., "Engineering a Security Kernel for Multics," Proc. ACM 5 Symposium on Operating Systems Principles, ACM Operating Systems Review 9, 5 pp.25-32 (November 1975).

[29] Sturgis, H.E., "A Postmortem for an Timesharing System," Ph.D. thesis, University of California at Berkeley (1973), Xerox PARC Technical Report TR 74-1.

[30] Wulf, W., et al., "HYDRA: The Kernel for a Multiprocessor Operating System", CACM 17, 6 (June 1974), pp. 337-345.

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Appendix A

Level 1 Processor Interface Summary

Operations (underscoring indicates output arguments)

Used by level 2 implementation for control of multiplexing:

```
VP1$bind (llproc, state, error)
VP1$unbind (llproc, state, error)
VP1$run (llproc)
VP1$stop (llproc)
VP1$next_stopped (llproc)
VP1$set_proc_interrupt (llproc)
```

Used by all level 1 processors:

```
VP1$await (ec1, value1, ec2, value2, ec3, value3)
VP1$advance (ec)
VP1$begin_atomic_operation ()
VP1$end_atomic_operation ()
VP1$get_fault_data (processor_state)
VP1$restore_processor_state (processor_state)
```

Used for managing lower level hardware:

```
VP1$propagate_map_change ()
VP1$add_cpu (cpu_id)
VP1$del_cpu (cpu_id)
VP1$crash_system ()
```

Special Eventcounts

Used in level 2:

```
stopped
outward_signals
```

Used in all level 1 processors:

```
clock
I/O processor event eventcounts
```

PROCESSOR MULTIPLEXING IN A LAYERED OPERATING SYSTEM

Appendix B

Level 2 Processor Interface Summary

Operations (underscoring indicates output arguments)

```
VP2$create_processor (envptr, startptr, schedclass, procname)
VP2$destroy_processor (procname, envptr)
VP2$await (ec1, value1, ec2, value2, ...)
VP2$advance (ec)
VP2$set_processor_interrupt (ec1, value1, ec2, value2, ...)
```

Internal Interfaces for Scheduler Domain of Binder/Scheduler

```
schedule (level_2_processor, quantum)
next-rescheduling (level_2_processor, nomore)
pre-empt (level_2_processor)
```

Eventcounts

```
reschedulings -- number of reschedulings that have happened.
free -- number of freed level 1 processors.
```

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-164

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 208 (213-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-208) UNNUMBERED TITLE PAGE, 2-207, UNNUMBERED BLANK
(209-213) SCANCONTROL, PRINTER'S NOTES, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 1/12/96 Date Returned: 1/18/96

Scanning Agent Signature: Michael N. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 01/12/1996
M.I.T. Libraries
Document Services