This blank page was inserted to preserve pagination.

DYNAMIC RECONFIGURATION IN A MODULAR COMPUTER SYSTEM

Roger R. Schell

June 1971

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

ACKNOWLEDGEMENT

I especially express my appreciation to my thesis supervisor, Professor Jerome H. Saltzer, for the substantial time and effort he spent supervising this thesis, and in particular for his helpful comments which greatly improved the presentation of this thesis.

Thanks are also due my readers, Professors F. J. Corbató and R. M. Fano, for their review and comments.

Appreciation is extended to Project MAC for making the Multics system available to the author both for conducting the research reported in this thesis, and for composing and reproducing this thesis document on-line.

A special note of thanks is due my wife, LaVonne for her assistance in typing this thesis, and for her patience and understanding throughout my years of graduate study at M.I.T.

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

DYNAMIC RECONFIGURATION IN A MODULAR COMPUTER SYSTEM*

Abstract

This thesis presents an orderly design approach for dynamically changing the configuration of constituent physical units in a modular computer system. Dynamic reconfiguration contributes to high system availability by allowing preventive maintenance, development of new operating systems, and changes in system capacity on a non-interference basis. The design presented includes the operating system primitives and hardware architecture for adding and removing any (primary or secondary) storage module and associated processing modules while the system is running. Reconfiguration is externally initiated by a simple request from a human operator and is accomplished automatically without disruption to users of the system. This design allows the modules in an installation to be partitioned into separate non-interfering systems. The viability of the design approach has been demonstrated by employing it for a practical implementation of processor and primary memory dynamic reconfiguration in the Multics system at M.I.T.

---

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degree of Doctor of Philosophy, May 1971.

TABLE OF CONTENTS

SECTION		PAGE
	Table of Contents	4
	Illustrations	8
1.	INTRODUCTION	9
	0. Modular Computer Systems	9
	1. Statement of the Problem	10
	1. The Environment	10
	2. The Nature of the Problem	12
	2. Background	15
	3. Method of Presentation	20
2.	SYSTEM STRUCTURE	22
	0. A General Model for Reconfiguration	22
	1. Resource Management Model	22
	2. Reconfiguration Model	28
	3. Design Approach for Reconfiguration	34
	1. Modular Characteristics of System	37
	1. Processing Modules	39
	2. Memory Modules	40
	2. Structure for Intermodule Communication	42
	1. Intermodule Links	43
	2. Module Interface Ports	44
	3. Summary of System Structure for Reconfiguration	46

SECTION			PAGE
3.	CHANGING MODULE UTILIZATION		48
	0.	Dynamically Changing Resource Usage	48
	1.	Storage Resource Management Environment	50
		1. Allocation With Unlimited Storage Capacity	51
		2. The Storage Utilization Problem	53
		3. The Effect of Limited Storage Capacity	57
		4. The Storage Allocator	58
	2.	Removing a Storage Module	63
		1. Freeing Demand Managed Storage	70
		2. Freeing Wired Storage	73
		1. Relocating Wired Information	74
		2. Avoiding Conflict	76
		3. Self-reference Problems	80
		4. Implications of External I/O	81
		3. Freeing a Storage Module Used as a Relay	85
		4. Summary of Storage Module Removal	87
	3.	Adding a Storage Module	88
	4.	Removing a Processing Module	90
	5.	Adding a Processing Module	96
	6.	Summary of Changing Module Utilization	99
4.	CHANGING THE HARDWARE CONFIGURATION		101
	0.	Dynamically Changing Physical Module Configuration	101
	1.	Changing Module Availability	104

SECTION		PAGE
2.	Changing Module Binding	109
	1. Storage Module Configuration	116
	2. Processing Module Configuration	118
3.	Partition-system Initialization	123
4.	Review	125
5.	THE MULTICS IMPLEMENTATION	127
0.	Background	127
1.	Removing a CPU	133
2.	Adding a CPU	139
3.	Removing a Memory	143
4.	Adding a Memory	148
5.	Experience With the Multics Implementation	150
6.	CONCLUSIONS	153
0.	Summary of Results	153
1.	Implications	154
	1. Module Interchangeability	155
	2. Operator Participation	156
	3. Automatic Configuration Control	156
	4. Resource Utilization	157
2.	Additional Research	158
3.	Remarks	160

SECTION		PAGE
BIBLIOGRAPHY		161
Appendix I	-- Multics Operator Instructions	163
Appencix II	-- Summary of Multics Supervisor Changes For Reconfiguration	170
Appendix III	-- Analysis of Current Multics Hardware For Reconfiguration	174
Appendix IV	-- Multics Hardware Design Proposal For Reconfiguration	183

ILLUSTRATIONS

FIGURE		PAGE
2.1	Fixed Configuration Resource Model	25
2.2	Model of Resource Management	27
2.3	Usage State Transitions	31
2.4	Resource State Transitions for Reconfiguration	32
2.5	Typical Module Interconnection Network	45
3.1	Usage State Transitions of Logical Resources	49
3.2	Address Translation for Paging	56
3.3	Address Translation for Segmentation	62
3.4	Example of a Storage Map	67
3.5	Example of Making Storage Unavailable	68
3.6	Environment for Moving Wired Storage	75
3.7	Environment Before Relocating Page Table	82
3.8	Environment After Relocating Page Table	83
4.1	Usage State Transitions of Physical Resources	103
4.2	Operator Interface Matrix	106
4.3	Operator Interface Matrix Schematic	108
4.4	Typical Intermodule Communication	112
4.5	Disabling Intermodule Communication	113
4.6	Enabling Intermodule Communication	115
4.7	Model of Processing Module Port	117
4.8	Model of Processing Module	121
5.1	Multics Hardware Installation	128

CHAPTER ONE

INTRODUCTION

1.0 Modular Computer Systems

The digital computer has evolved from a single processing unit dedicated to one sequential task at a time to the modern computer utility simultaneously serving many users. As the size and complexity of computers have grown, a modular hardware design has commonly been employed. As with other utilities (such as electrical power and telephone services), the computer utility must be able to change its hardware configuration without disrupting the services provided. This thesis presents an abstract model of reconfiguration operations, and thus provides a discipline for designing operating systems and hardware modules for a computer utility so that the configuration of constituent modules can be dynamically changed. The research reported here also includes a practical processor and memory module reconfiguration capability for the Multics system [1] at MIT.

The complexity of a large computer has naturally led designers to view the system as being composed of a number of smaller modules. Not only is such a modular viewpoint indicated for purposes of system design and analysis, but also the practical problems of maintenance, operation and system evolution have made it desirable to construct large systems

from a small number of distinct types of physical modules. These individual types of physical modules typically reflect the primary functions of the design -- memory modules, processor modules, input/output channels and input/output devices are common examples. The total system is formed by interconnecting a collection of these modules in a particular configuration, and the quantity of each type of module determines the system capacity.

1.1 Statement of the Problem

This thesis considers basically the following design problem for reconfiguration in the modular computer system: provide an architecture for the modules and their interfaces and a design for the operating system that permits a computer operator to dynamically (viz., automatically and without interrupting the service provided to users of the system) add modules to and remove modules from the system. This design must be flexible enough to apply to all modules of a given type, and must minimize the potential for disastrous errors.

1.1.1 The Environment

The ability to dynamically modify the configuration of hardware modules in a computer system is primarily of interest where a large data processing capacity is required and where a high system availability is desired. Over the last few years

substantial work has been directed towards systems providing multible users easy access to common information. A single, high-capacity system is required to serve many users with common information. To serve many users effectively, the information (and thus the system) must be available on the demand of the user: this type of computer system has been referred to as a "computer utility".

As with other utilities, the computer utility must have substantially greater resources than those required by any one user. Furthermore, as with other utilities, the system must be able to, without becoming unavailable, withstand the removal of any module for maintenance. It is this need for a large, concentrated, continuously available computational capacity that underlies many of the more significant design features of a computer utility such as the Multics [1] system.

In the early development of digital computers, the need for more powerful machines was met by building faster machines with more storage capacity; however, as technological and economic limitations were approached, multiple units were used to increase capacity. For example, processor speeds are limited by the velocity of propagation and the physical dimensions between computational logic and the instruction storage medium; therefore, additional total capability is sought by including multiple processors in the system. Similarly, storage and input/output capabilities have been increased by the use of multiple modules.

An interconnection of various different types of computers can also provide an increased capacity; for example a small computer for reading cards and driving a printer can be connected to a larger computer for more compute-bound portions of a job. However, a single modular system with a number of each of a very few distinct types of modules reduces the complexity since there is only one operating system, provides a higher availability since only a few modules are required to provide a spare for every type of module, easily expands to a larger system since modules are merely added without removing the previous system, and is simpler to maintain due to the commonality between similar modules. Various commercial systems have used a modular interconnection scheme including the Univac 1108 Multi-processor System [2], the Burroughs D825 [3], the GE 635 [4], and the IBM 360 Model M65MP [5].

1.1.2 The Nature of the Problem

One problem is matching the demands of the load to the capacity of the system. The computer utility must always have enough capacity to provide a response within a few seconds, since people tend to stop working if they have to wait. To be fully effective the services must be continuously available, since the stored information must be accessible when the user wants it. Utilization efficiency would indicate that during

the greatest load, a major portion of the processing capacity (viz., essentially all the modules) would be used to meet customer demands, while during lighter loads it is desirable to remove unneeded modules.

Some of the objectives to be achieved by modifying the configuration are to reduce costs associated with operating the modules, forming an independent processing unit with some of the modules, allowing periodic and preventive maintenance to be performed, and reducing the amount of hardware in the system in order to decrease the probability of hardware failure, i.e., increase the reliability. These include the basic reconfiguration goals that were considered important for the Multics computer utility; these goals have been given elsewhere [6 ,7 ,8], but are briefly reviewed here:

1. An essential requirement is that the reconfiguration (addition or deletion) of a module be accomplished dynamically, i.e., rapidly and without disruption of the service provided.

2. The reconfiguration design should treat all identical modules in the same way -- for example, it should be possible to add or remove <u>any</u> primary memory module. Maintenance requirements can be expected to apply to every module, so no specific physical unit should be allowed to be indispensable, either by hardware constraint or software convention. Continuous service can only be approached by a reconfiguration design in which no single module is essential for continued

system operation.

3. It must be possible to partition the modules of the installation into separate noninterfering systems. These independent partition systems may be used for providing service to users, for development of changes to the operating system, or for maintenance testing of the hardware. This partitioning of modules should be possible without massive physical effort, such as changing of cable connections, which is slow and prone to errors.

In this thesis dynamic reconfiguration is considered as a response to a request from outside the system -- specifically a computer operator. Therefore, an important consideration is the human interface. In particular, an operator should be able to initiate a reconfiguration that is completed rapidly (within a few seconds) and automatically (without further operator intervention), and there should be no possibility for operator errors to cause disastrous results. This thesis will not address the more difficult problem of spontaneous reconfiguration (e.g., in response to an observed component failure); however, the results presented here provide a basis for future research in this area.

For simplicity of design and implementation reconfiguration should be provided by applying the normal capabilities of the system whenever possible. The operator controlling reconfiguration should be just another user of the system who has been given the privilege of using the required

system primitives. The processing and storage overhead required when reconfiguration is not in progress should be minimal.

This thesis will explicitly consider only the modules that make up the central data processing capacity of the system. In particular, modules associated with input/output to devices external to the system itself are excluded. In addition, it has been pointed out by Dennis [9] that modular interconnection schemes do not readily generalize to very large numbers (e.g., hundreds) of modules. However, practical systems for the present and foreseeable future can find a modular design very useful, since systems can reap the major benefits of simplicity, high availability, expandability, and commonality with quite a small number (e.g., a few tens) of modules -- clearly a few modules are enough when each module provides a significant portion of the total capacity. This thesis will confine itself to such contemporary modular structures.

1.2 Background

As pointed out above, the importance of dynamic reconfiguration was clearly appreciated in developing the goals of Multics; however, the initial design did not include this capability, attesting to the non trivial nature of the problem. Although past systems have included (limited)

reconfiguration capabilities, a general solution has not previously been available.

A modular hardware organization is of course not new, and modular systems have frequently been applied to general purpose computational tasks. The need to operate general purpose systems with a variety of configurations has been widely recognized in the past. The manufacturer's description of the IBM 360 Model 67 [10] points out that the duplex configuration of this system can be partitioned into two isolated subsystems -- manual switches and processor capabilities are provided for making part of the system unavailable for program control. Not only has the need for partitioning been previously recognized but also the need for uniform treatment of similar modules has been considered. The GE 635 [4] and the IBM 360 Model 67 both include manual switches to provide "floating addressing", i.e., any memory module can be used for any required address interval, so that any memory module can have maintenance performed on it without disabling the entire system. It has also been noted that the ability to operate with various configurations of modules permits changing the system capacity to meet changing demands -- for example, the manufacturer's description of the Burroughs B6500 [11] points out the expansion capability implied by a modular structure.

Although systems have previously been developed to operate with a variety of configurations, a significant

interruption of service is usually required to change the configuration. In most cases the configuration is specified prior to loading the system on a raw machine, and reconfiguration is accomplished by essentially reloading with a new configuration. This was generally the operation of the initial Multics design. Limited reconfiguration capabilities have been provided for general purpose computer systems, most notably the IBM 360 Model M65MP [5]; however, without a general design model these systems have left unsolved some of the more difficult problems, such as removal of the memory modules containing the "resident supervisor".

Although dynamic reconfiguration is important for general purpose computer systems principally in the computer utility environment, special purpose systems have in the past used modular reconfiguration to enhance reliability. Although quite successful in their intended application, the design in these systems is so permeated with the peculiarities of the particular application and specific hardware that little general structure is evident.

A form of reconfiguration has been provided in special purpose systems using redundant modules. With this technique identical processing and storage functions of the system are simultaneously performed using separate physical modules -- the reliability motivation is that if one module fails it can be removed from the configuration, and the results of a good duplicate module can still be used. An example of this type

of reconfiguration can be found in the Bell (Telephone) System Electronic Switching System [12]. Also the American Airlines' SABRE system [13] used duplicate (viz., on-line and standby) IBM 7090's, redundant I/O terminal interchanges, and multiple copies of vital records on storage modules in order to achieve the capability for rapid restart of the system with minimum risk of information loss in case of failures. Not only is it costly to provide duplicate hardware, but also this type of reconfiguration does not allow the capacity of the system to be dynamically changed.

Special purpose systems have also frequently used a "snapshot" reconfiguration technique that is not acceptable for the general purpose computer utility. Periodically a "snapshot" is made of a small amount of data from which all computations can be restarted. The system can then be stopped (destroying the computations in progress), the configuration changed, and the system restarted from the snapshot data. This technique requires a detailed knowledge of the computation being performed, and requires that it be acceptable to destroy a portion of the computation. A related technique takes advantage of a carefully designed cyclic behavior for the computation by changing the configuration only at a fixed point in the cycle where the computation can be continued from a small amount of saved data. On the other hand in a computer utility the user can specify computations whose nature is unknown to the system and for which any

disruption may be totally unacceptable.

This "snapshot" technique has been used in various command and control systems. The SAGE [14] air defense system, one of the earliest (about 1958) large scale computer systems, had an embryonic reconfiguration capability to obtain greater reliability: its "users" were approximately 100 operators at display consoles, and its vacuum tube central computer (logically a single "module") was duplexed so that when there was a hardware failure, the entire spare computer would be put on line to continue service to the "users" from a periodically saved snapshot. A backup to this system, known as BUIC [15], used the modular, transistorized Burrough D825 computer. BUIC used online fault detection and an automatic modular reconfiguration capability to provide an operational failure rate much less than the inherent hardware failure rate.

One of the largest and most recent of these special purpose real-time systems is the IBM 9020 developed for the FAA air traffic control system, which gives attention to special hardware features to facilitate reconfiguration for increased reliability. Although these systems and others like them serve to demonstrate the key role of dynamic reconfiguration in approaching the goal of "continuous operation", the observation made for the IBM 9020 system places the specific achievements of this type of system in perspective: "It deserves emphasis that the multiprocessing

system under discussion is application-oriented in the sense that many of its functional capabilities are designed to meet explicit requirements. It would be another matter to formulate such capabilities for a general purpose environment" [24].

1.3 Method of Presentation

A substantial portion of the research effort reported here has been directed towards the design and implementation of a dynamic reconfiguration capability for the "Multics" system (Multiplexed Information and Computing Service) at MIT. This early engineering design gives Multics the capability to dynamically add and remove central processing units and memory modules with no disruption to the users. These capabilities are regularly used in the normal operation of this system which currently supports more than 50 simultaneous time-sharing users. Although the author has found this experience invaluable in gaining insight and practical understanding of the issues involved, this thesis is not intended to be a description of a bag of programming tricks used to arrive at a particular initial reconfiguration capability for Multics: the primary goal of the research has been to develop a design approach that can be applied to the evaluation of an existing system or the design of a new system where dynamic reconfiguration is desired. The successful

application of this orderly design approach to Multics demonstrates its viability, and specific examples are drawn from Multics to aid in the explanation of the design approach.

In chapter two we first develop a general model of reconfiguration -- the concept of binding is used to model the operations of reconfiguration. We show how this model can be interpreted in terms of contemporary modular computer systems.

Next the general model is used to develop a specific design procedure for an operating system and hardware architecture to provide reconfiguration. In chapter three we identify a design for the program oriented primitives required to dynamically change the set of modules actually being used by the operating system. Then in chapter four we develop a hardware oriented structure that allows dynamically changing the set of modules actually accessible to the system -- this structure is directly influenced by the need for automatic reconfiguration, viz., without human operator participation.

Chapter five presents the experience with the experimental Multics version of the ideas presented. Some of the tradeoffs involved and the compromises required in the specific Multics implementation are presented. An appendix is also included that, based on the ideas of this thesis, proposes a specific hardware design for an improved reconfiguration capability for Multics.

CHAPTER TWO

SYSTEM STRUCTURE

2.0 A General Model for Reconfiguration

A primary task of any computer system is to transform the capabilities of the hardware units into resources that can be used to perform desired computation for users of the system. The concept of binding, defined below, can be used to model the system functions which organize the raw hardware capabilities into a usable form. In this chapter such a model is constructed and then augmented to provide a model of the reconfiguration operations. The model is used to identify the structure of computer systems for which reconfiguration as presented in this thesis is applicable, and it is shown that the architecture of contemporary modular computer systems is representable by this structure. In the following chapters the generalized model is used to develop a specific model of the individual functions needed for reconfiguration.

2.0.1 Resource Management Model

The model developed here is based on the observation by Dijkstra [17] and others that in a sequential process only the time succession of the various states has logical meaning, but not the actual speed with which the sequential process is performed. In particular to develop our model for

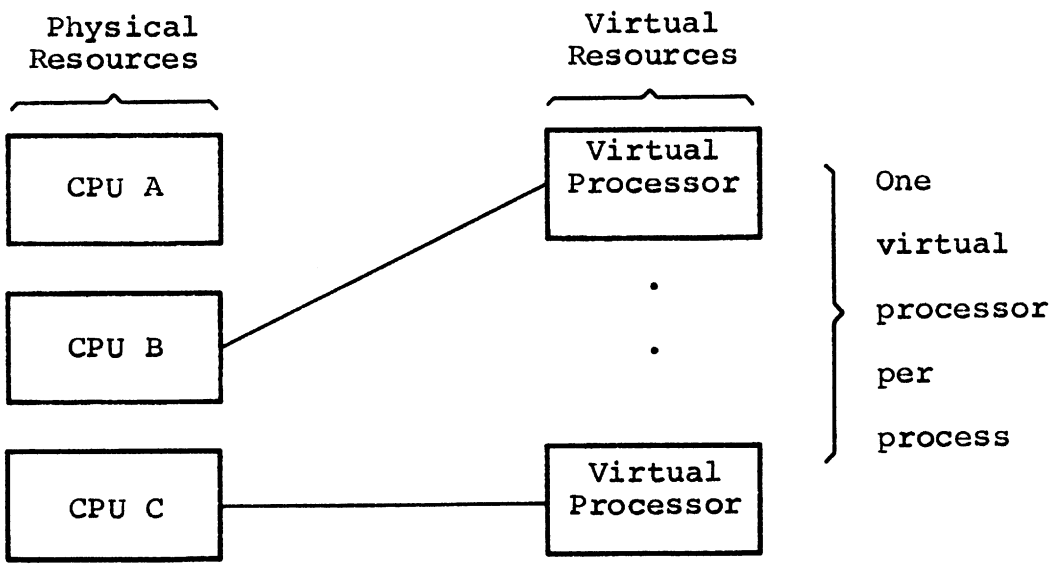
reconfiguration we use the concept of the state of a process represented as stored information which is subjected to functional transformations as a process proceeds. The view of computation as asynchronous does not completely represent real computer systems since synchronous computations also occur. However, for purposes of reconfiguration the operation of most computer systems is asynchronous enough that this view of a sequential process is adequate.

In the computer utility environment there may be a number of computations (i.e., processes) in progress at the same time, although at any instant in time many of these may not actually be executing on a hardware processor. However, since all the computations are in the long term view proceeding, each and every process can be considered as the execution of a program on a "virtual processor" with its instructions and data stored in a portion of a "virtual memory". These processor and memory resources required by a process will be termed virtual resources -- virtual resources are used to model demands for actual processing and memory capability. At any point in time the set of virtual resources of all processes in the system represents the demand for system resources.

The actual processing and memory capability of the system is provided by some (usually fixed) configuration of physical resources, viz., hardware devices. Since a process can proceed only when the physical resources are actually

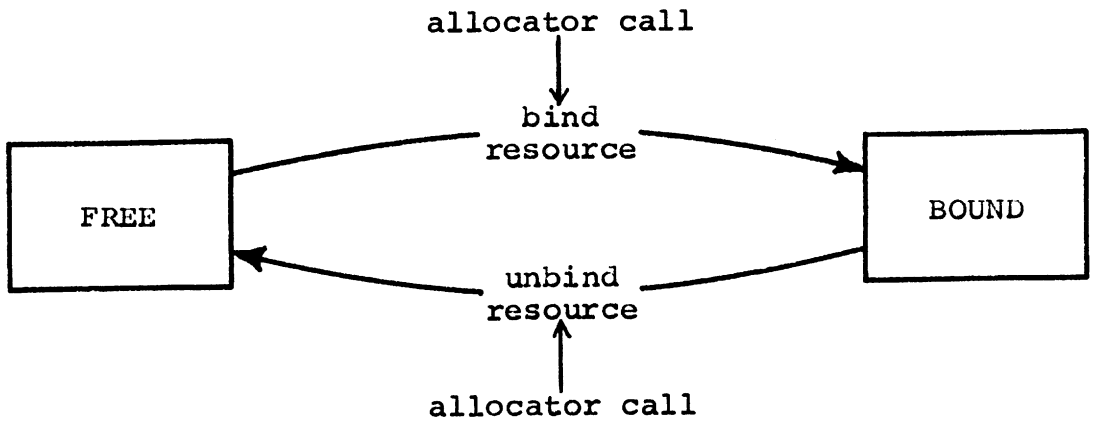
available to it, the system must include an allocator which assigns the available physical resources to meet the demands of the processes. If binding is considered as an operation of establishing a correspondence between objects, then in this model a primitive function of the allocator is to bind some subset of the virtual resources to physical resources: we define these resources to be in a bound state. The allocator also includes a primitive to unbind (typically on a millisecond basis) the available resources so they can be used to meet other demands. At any given time, a physical resource may not actually be bound to a virtual resource due to either the lack of demand or due to allocator transients: we define such resources to be in a free state. Figure 2.1 illustrates this model of resource management.

In the above model we have identified processing and memory capability with the actual hardware units. In a real system a given hardware unit represents a useful resource only when electrically connected to other hardware units of the system in a known way. In studying reconfiguration of resources we must consider such connections in some detail; therefore, we explicitly distinguish between a physical component and the capability actually available from it. Furthermore, we introduce the notion of logical resources to represent the abstract processing and memory capacity. It is logical resources that are managed by the allocator. For each available logical resource there must be some physical



Lines show binding that exists --
changed on millisecond basis by allocator

Example of Resource Allocation



State Transitions of an Available Resource

Figure 2.1 -- Fixed Configuration Resource Model

resource, viz., a hardware device. The binding of the logical resources to the physical resources represents what is commonly called the configuration of the system: it changes only through reconfiguration. With this model, hardware capabilities being applied to a process are represented by physical resources bound to logical resources which are in turn bound to virtual resources of the process.

At this point a few observations can be made about binding as a model of the system's resource management. We redraw the resource allocation illustration of Figure 2.1 to take into account logical resources -- Figure 2.2 indicates the relationships that can exist between physical, logical and virtual resources. This shows the simplified case where each hardware unit contains a single unit of resource -- in Multics parlance, when each memory module can store only a single page of information. In this context "binding" refers to establishing a mapping or correspondence between the names of physical and logical resources and of logical and virtual resources. The total set of physical resources is determined by the physical hardware present in the installation. Each physical resource has a name used (for example by a human operator) to identify the specific hardware: a central processing unit is an example of a physical resource. A subset of the physical resources is included in any given configuration, i.e., is bound to logical resources. These physical components are electrically interconnected so that

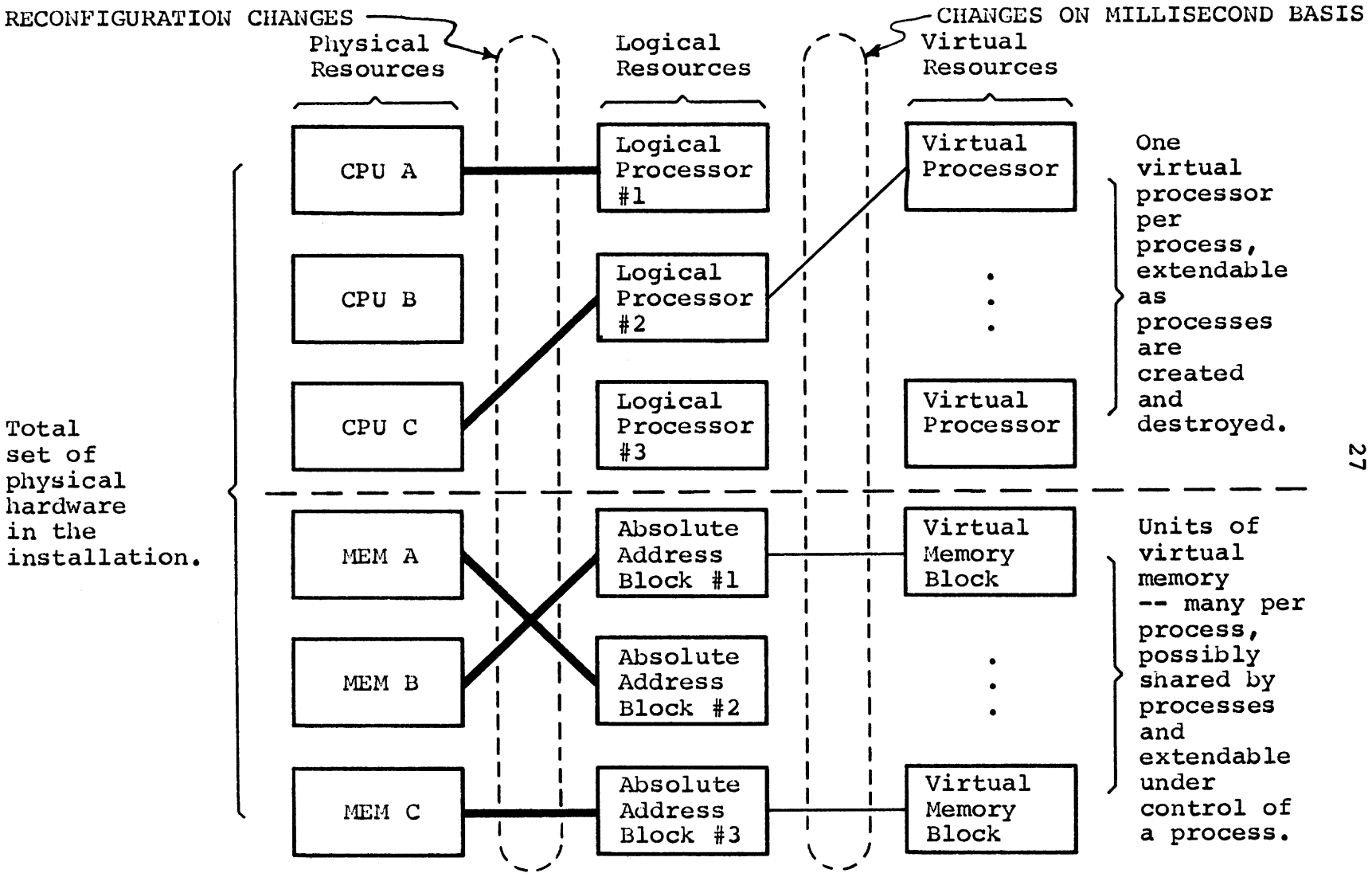


Figure 2.2 -- Model of Resource Management

their capabilities are accessible to the system.

The total set of logical resources is determined by the set of permissible names. For example, an absolute address is a name for a logical memory resource -- the total set of logical memory resources in the system is determined by the range of absolute addresses provided in the design. The computation capacity available to meet the demands represented by virtual resources is modeled by those logical resources bound to physical resources in the configuration. A subset of these available logical resources is actually bound to virtual resources and the remainder is free to be used for future allocation. The current set of virtual resources (i.e., demands for resources) changes dynamically as processes proceed with their computation.

In the above model, the state of resource management for the system at any point in time is represented by physical resources bound to logical resources and logical resources bound to virtual resources. This model is sufficiently general that, with an appropriate interpretation, it can represent a wide variety of computer systems. Next this model will be used to consider the problem of reconfiguration.

2.0.2 Reconfiguration Model

In the typical computer system the configuration (represented here as the binding of physical resources to

logical resources) is established during the initialization of system operation and remains fixed until the system is shutdown and reinitialized with a new configuration. In contrast, dynamic reconfiguration is the adding and removing of physical resources while the system is running, and the model developed above allows us to view binding as the central issue in reconfiguration. In particular, the problem of removing physical resources from the configuration concerns the reversibility of binding, and adding physical resources to the configuration is an example of delayed binding.

We have previously observed that physical and logical resources can be either bound or free. When reconfiguration is introduced we need the notion of available and the inverse, unavailable. We will introduce this concept in terms of the operations of the allocator on logical resources. In our model, any logical resource bound to a physical resource is accessible to the system; with a static configuration all these logical resources are available to the allocator and may be either bound or free. All logical resources not bound to physical resources are unavailable to the allocator, and they must be free, since clearly they cannot be used to meet resource demands. To add logical resources at reconfiguration time, an unavailable (and thus free) resource is made available. To remove an available logical resource (which may be either bound or free), it must be made both unavailable and free. If we attempt to insure that the resource is free

before making it unavailable, we can have a race condition with the normal allocator functions asynchronously attempting to bind free resources; therefore, we make the resource unavailable for future allocation before insuring that it is free. Now the state transitions shown in Figure 2.1 can be redrawn in Figure 2.3 to include the resource usage state transitions introduced by reconfiguration.

These state transitions can be applied to physical as well as logical resources. For a physical resource the usage state reflects binding to a logical resource. A physical resource is available when the system is able to change its binding to logical resources. An available physical resource is bound when it is associated with the name of a logical resource. Since the state is changed only through reconfiguration, race conditions can occur only if there are simultaneously executing reconfiguration routines. For example, a system spontaneously adding free modules to meet a peak in load could race with an operator trying to remove a specific module; however, recall that in this thesis we are considering only (strictly sequential) operator initiated reconfiguration. Therefore, for physical resources we can omit the "bound and unavailable" state added in the previous discussion of logical resources.

The relationship between physical resources and logical resources is illustrated in Figure 2.4, which shows all the resource state transitions that are involved in dynamic

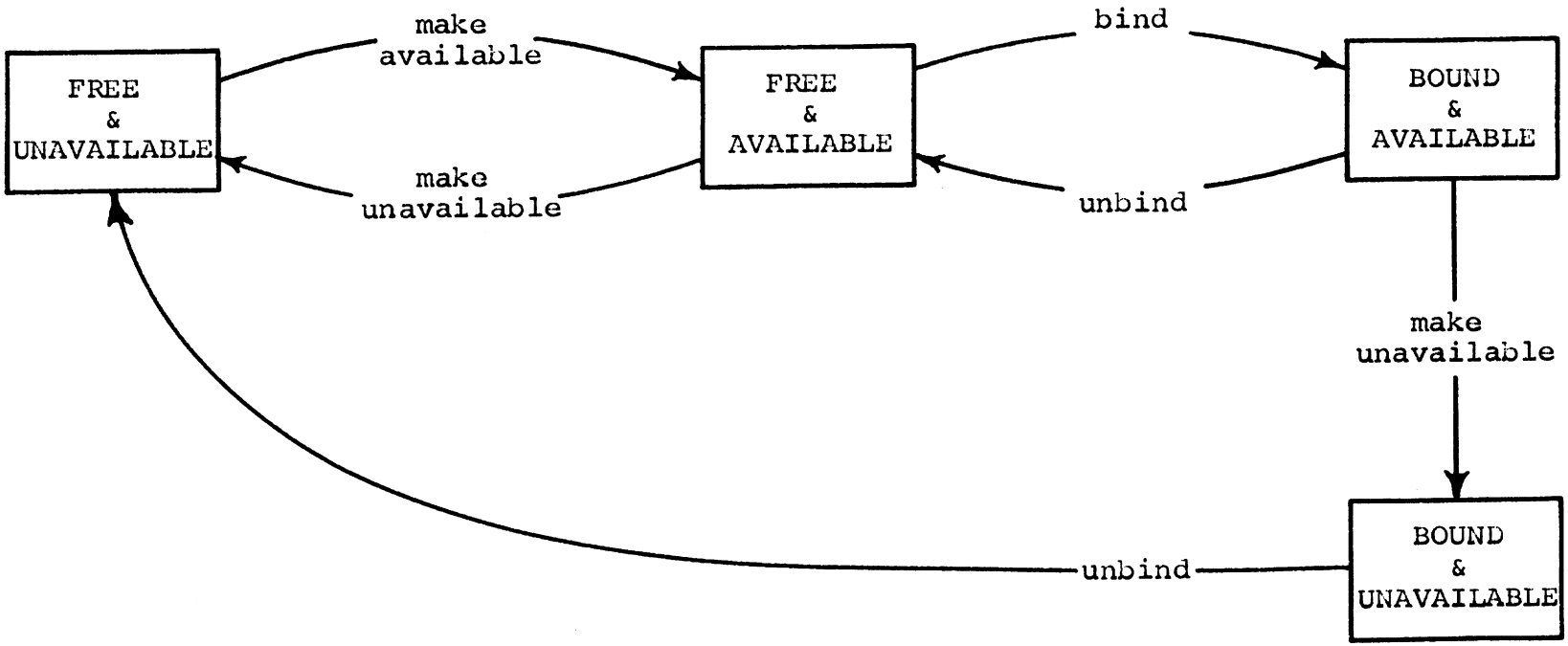


Figure 2.3 -- Usage State Transitions

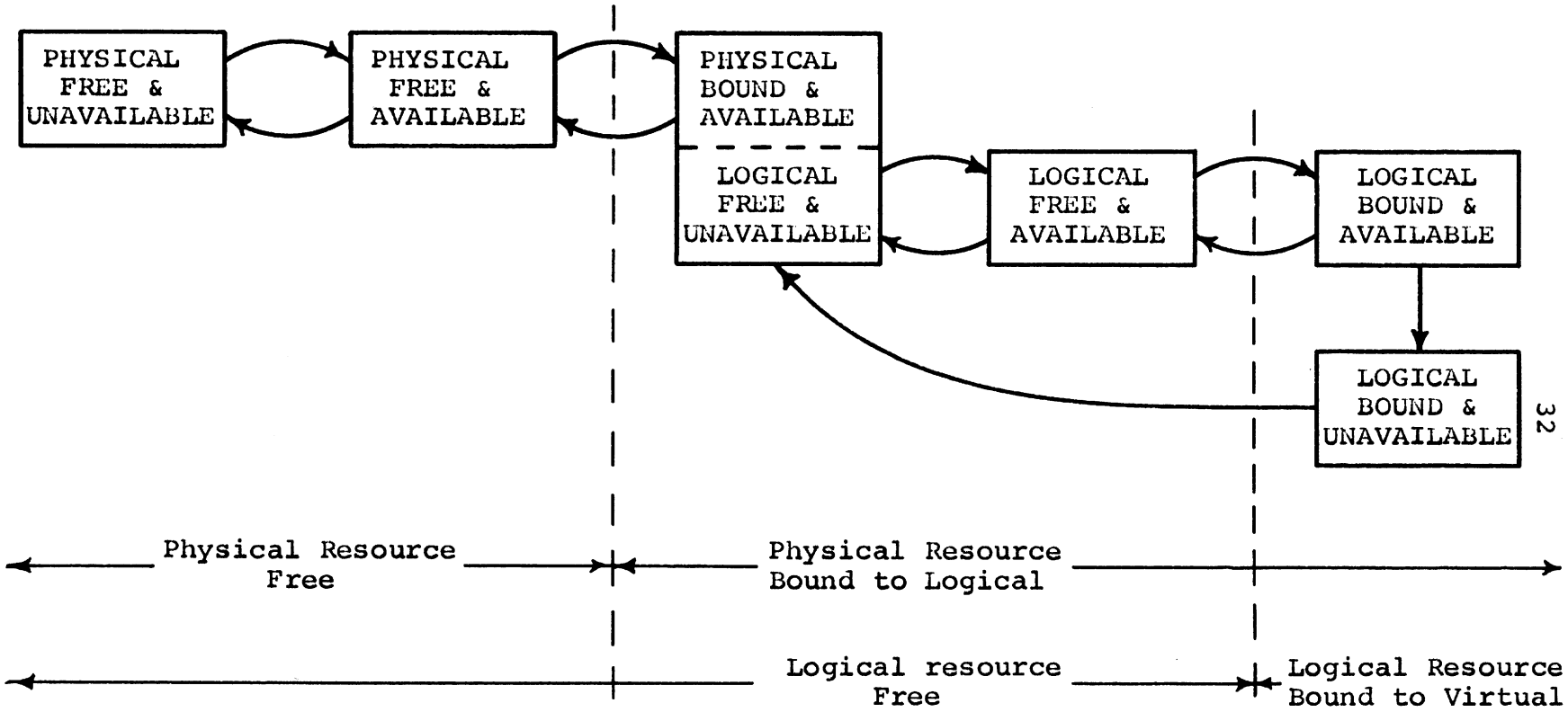


Figure 2.4 -- Resource State Transitions for Reconfiguration

reconfiguration. State transitions for a logical resource are defined only when there is an available physical resource bound to this logical resource, as illustrated in Figure 2.4 by a single state for both "physical bound and available" and "logical free and unavailable".

Based on the model that has been developed we can identify the primitive reconfiguration operations required to change the usage states when removing a resource from the system. The following sequence of steps is required:

1. The corresponding logical resource is removed from the set of logical resources available to the allocator for future binding to virtual resources.

2. The logical resource is made free by removing any existing binding to a virtual resource.

3. The physical resource is made free by removing the binding to the (free) logical resource.

4. The physical resource is made unavailable for binding to logical resources.

When a physical resource is added to the configuration the corresponding logical resource is initially free. The following sequence of steps is required to add a physical resource:

1. The physical resource is made available for binding to logical resources.

2. The physical resource is bound to a logical resource not currently available to the allocator.

3. This logical resource is added to the set of logical resources available to the allocator for binding to virtual resources, viz., is added as a free and available logical resource.

4. The system then binds the logical resource to virtual resources as a result of the normal operation of the allocator.

We have now developed a complete model for all the reconfiguration operations. In chapter three and four this simple model will be applied in detail to, respectively, logical and physical resources.

2.0.3 Design Approach for Reconfiguration

Throughout this thesis reconfiguration is viewed as changes in binding. This view of reconfiguration allows us to recognize the general form of a design procedure for implementing dynamic reconfiguration. First, it is clear that reconfiguration is directly related to the design of the allocator for the resources to be reconfigured. For reasons other than reconfiguration, substantial research has previously been done in the area of allocating a computer system's logical resources to meet the demands of user processes; therefore, when applicable, currently available technology will be summarized rather than reinvented.

We will now define allocator primitives to implement the

logical resource state transitions that have been identified. First, the allocator must have primitives which can be invoked to add and remove elements from the set of available logical resources being managed by the allocator. This means, of course, that the allocator must be able to properly manage changing amounts of resources, although there may be bounds on the allowable size of the set of resources, e.g., a system with no processor will probably not work at all. Using the notation of the PL/I language, we introduce these allocator primitives as generic closed subroutines of the operating system:

```
call Make_available (resource);
call Make_unavailable (resource);
```

where "resource" is the name of the unit of logical resource affected. These primitives affect the binding of logical resources to virtual resources that can occur in the future, but they do not change the current state of resource binding.

In addition, the allocator needs a primitive for freeing logical resources. For reconfiguration to be dynamic it must not disrupt (viz., change the outcome of) user processes, and the change in configuration must be completed in a period of time that is responsive to an operator request (viz., on the order of a few seconds). If the system can assure that a logical resource will be made free within a short period of time, then when removing a resource it is satisfactory to merely wait until the resource is free. In the more usual

unbinding physical resources. While computer programs provide the allocator functions, the availability and binding state of physical resources is a model for the more hardware oriented mechanisms that establish electrical connections between the physical hardware components of the system. In systems not using dynamic reconfiguration, an operator typically accomplishes such connections by switches and cable connectors which cannot be changed without disrupting the computations in progress. For dynamic reconfiguration the system invokes primitives that, without disrupting user computations, change the connections between hardware units. Chapter four will consider in detail the primitive functions needed to change the configuration of the system.

We have identified the primitive functions necessary to provide a dynamic reconfiguration capability for a computer system. Although the model just presented did not explicitly distinguish between the various elements of resource, in a typical contemporary computer system there are clearly distinct classes of resources, e.g., processors and memories. The remainder of this chapter is primarily devoted to making those distinctions relevant to modular reconfiguration.

2.1 Modular Characteristics of System

As indicated in chapter one we are primarily concerned with computer systems composed of distinct physical modules.

From the viewpoint of reconfiguration, a "module" is a subset of the physical resources of the system which comprise an identifiable hardware unit. To reconfigure a module, all the physical resources comprising the module are added or removed using the design just outlined in the preceding section.

For convenience we assume that a module contains an integral number of units of physical resource. This approach is further suggested by the observation that nearly all contemporary modular computer systems are designed in this way, with each unit of physical resource in a single module, as illustrated by the store protection blocks of the IBM 360 series [18] and the memory pages of the GE-645 [19].

The significance of this modular constraint is illustrated by an example from Multics. Primary memory is allocated in blocks of 1024 words. A hardware core memory unit has a capacity that is a multiple of 1024 words and is usually considered as a module. However, in an attempt to reduce conflicts between multiple devices simultaneously referencing the same physical memory, it is possible to interlace two or four of these memories -- with interlace, consecutive double word addresses are located in different physical memories. This means that a single unit of resource (viz., a 1024 word block) is contained in more than one memory; therefore, for purposes of reconfiguration the set of interlaced memories must be considered as a single module, and these memories can be added to or removed from the

configuration only as a single entity. If individual memories are to be reconfigured when interlace is needed, then the interlace feature should be designed so that only locations within a single memory are interlaced.

Since a module is reconfigured by individually reconfiguring each of its component physical resources, no loss of generality results from assuming that a module contains only a single type of resource. It is also observed that many contemporary systems are designed with such homogeneous modules -- for instance the IBM 360 Model 67 [10], the GE-635 [4] and the Burroughs Model B6500 [11]. This assumption allows a computer system to be viewed as an interconnection of processing and storage modules, which is convenient for considering reconfiguration.

2.1.1 Processing Modules

A processing module primarily performs functional transformations on data stored external to the module. Therefore, a processing module must be able to read its instructions and/or data from and write its results into external storage in the system. At any point in time a processing resource is executing in behalf of only one process -- the process to which the system has allocated this resource is moving through a time succession of states as a result of the transformations being performed. The state of this

process is represented by information stored in two places: data stored internal to the processing module (e.g., in registers) and data stored externally. We will assume (as is typical of contemporary processing hardware) that, conditional only on explicit locks, it is always assured that the internal data are of importance only to the process currently executing.

This model of a processing module is easily understood as a representation of the typical central processing unit (CPU) of a computer system. However, most computer systems have other devices manipulating stored information that can also be considered as processing modules. The most common examples are channels used to access storage managed by the system, such as magnetic disks or drums used for on-line storage. In his discussion of traffic control Saltzer [20] has noted that a channel is really nothing more than a simple processor with a wired-in program. However, recall that channels with interfaces outside the direct control of the system (viz., "source" and "sink" input/output) are not specifically considered in this thesis.

2.1.2 Memory Modules

A memory module provides some physical medium for the storage of data that is used by the processing modules of the system, and it is required that memory modules never modify

stored data. A significant characteristic of a memory module is that it may often be referenced in parallel by multiple processing modules, and uncoordinated data references can potentially produce conflicts. The reconfiguration primitives must provide for avoiding any additional potential conflicts which they introduce.

Primary storage contains information which must always be accessible to some processing module, in particular some instruction or command that can be referenced to control the next processing action. This information is commonly termed the "resident supervisor" or "wired down" programs and data. Primary memory is usually provided by a relatively fast, random access storage medium such as core or semiconductor memory: due to relatively high cost primary memory can usually meet only a small portion of the total demand for storage.

Potential conflicts in primary memory are often avoided by providing areas reserved for use by a single processing module. Since there may be no system primitives to prevent conflicting access to primary memory, reconfiguration primitives may require additional mechanisms to prevent conflict. For instance, when removing primary memory, reconfiguration primitives will be required to copy (and therefore access) all the information in a memory module -- including that reserved for use by other processing modules.

The memory of the system that does not require immediate

access is generally known as secondary storage, and is typically provided by slow, high capacity devices such as rotating magnetic disks and drums or even magnetic tape. The significant implication for reconfiguration is that a system will typically have storage allocator primitives to move data to and from secondary storage without risking conflicts.

2.2 Structure for Intermodule Communication

Although the total computation capacity of a modular system is provided by a collection of processing and storage modules, it is clear that the system design must include an interconnection network to satisfy the requirements for communication between the modules. This network is of direct concern to reconfiguration since communication paths between modules reflect the binding between the physical resources contained in the module and logical resources of the system.

Rather than dilute the discussion by considering each basic idea in terms of many possible structures, a general form for the interconnection network will be developed here and used throughout this thesis. This structure of communication links between modules, and ports to provide an interface between these links and the individual modules, is representative of contemporary modular systems.

2.2.1 Intermodule Links

Intermodule links are required between modules that need to communicate with each other. It is clear that, for transfer of data, every memory module must have a link to at least one processing module and every processing module must have a link to at least one memory module. Since a memory module is completely passive, there is no need for communication between memory modules. However, communication is needed between processing modules. Common examples are the use of interrupts for signals from a channel to a central processing unit, and a central processing unit issuing commands to a channel. In addition, interrupts are often used to control the allocation of processing resources (viz., for traffic control), as examined in detail by Saltzer [20].

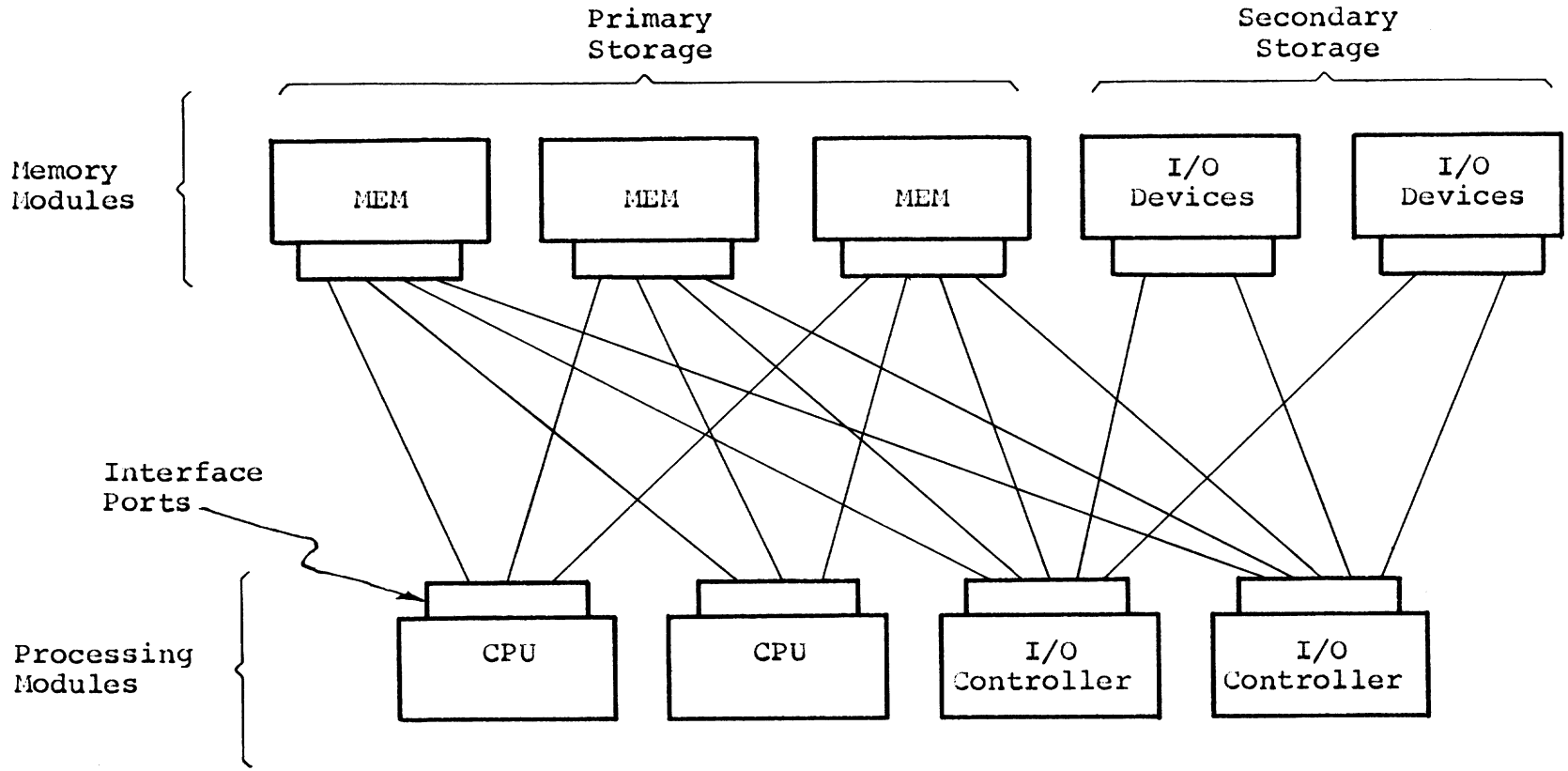
A direct link could be provided for communication between processing modules. However, in some systems (Multics in particular) a memory module is used to relay signals, using the processing/memory link. One motivation for such a structure, with links only between processing and memory modules, is that the total number of links is reduced. In addition, by using a memory module for a relay, signals can easily be broadcast to multiple processing modules -- say to permit a channel interrupt to be processed by the first CPU that is available, thus giving the fastest possible response to the interrupt signal.

2.2.2 Module Interface Ports

The links between modules can be thought of as the data transmission paths between modules. To complete the interconnection network there must be a "port" which provides an interface between each link and the physical resources within the module. Figure 2.5 shows a typical modular system that reflects this interconnection network structure.

As we have seen, a processing module must make references to external storage for instructions and data. For each reference, the port of the processing module uses some "selection logic" to map an absolute address for the desired type of storage (viz., a reference to a logical resource) into a reference through a link to a particular storage module and a particular location within that module (viz., a reference to a physical resource). In addition the processing module port must be able to send and receive signals for communication with other processing modules. In chapter four we will examine specific alternatives for the implementation of a processing module port.

The port for each storage module will respond to references from processing modules. The storage module port will receive over a communication link commands to read/write and will receive the corresponding address within the module: data that are to be read/written will also be passed over the link. In addition, the port must receive commands which cause



(Lines represent intermodule communication links)

Figure 2.5 -- Typical Module Interconnection Network

signals to be relayed to processing modules (e.g., send an interrupt or mask an interrupt).

Although we have developed a structure representative of current modular computer technology, it is pointed out that this structure has only been applied to systems with something in the order of ten modules. Because of the communication network "crossbar" problem it is doubtful if this structure would be effective for more than a few tens of modules in a system, and this thesis will not attempt to develop new structures for a larger number of modules.

2.3 Summary of System Structure for Reconfiguration

We have, in this chapter, used changes in binding between physical, logical, and virtual resources to model reconfiguration. Based on this model we have identified the computer system structure for which this thesis will consider the problem of dynamic modular reconfiguration. We have considered a computer system as made up of a collection of physical modules (each containing an integral number of units of processing and storage resources) interconnected by a communication network of links and interface ports. This framework has been developed so that the following chapters will have a firm conceptual foundation on which to build detailed solutions to the particular problems of dynamically adding modules to and removing modules to modules from a

computer system.

CHAPTER THREE

CHANGING MODULE UTILIZATION

3.0 Dynamically Changing Resource Usage

In this chapter we assume a static hardware configuration and make a detailed study of how to dynamically change the resources the system is actually using; we defer until the next chapter the problems of changing the hardware configuration. The usual operating system has an initialization phase during which the processing and storage modules used by the system are established by operator inputs and system conventions (e.g., assuming a zero-based contiguous range of absolute core addresses). We will now examine in detail how the system can stop using a module currently in use and start using an additional module without reinitializing the entire system.

In terms of the model introduced in chapter two, we must basically develop a design for changing the usage state of logical resources. Figure 3.1 illustrates the operations that are required -- notice that this is just the specific instance for logical resources of the state transitions presented in chapter two (Figure 2.3). In this chapter we develop specific reconfiguration primitives for the storage resource allocator and processing resource manager, viz., the traffic controller. We first clearly identify the critical properties that,

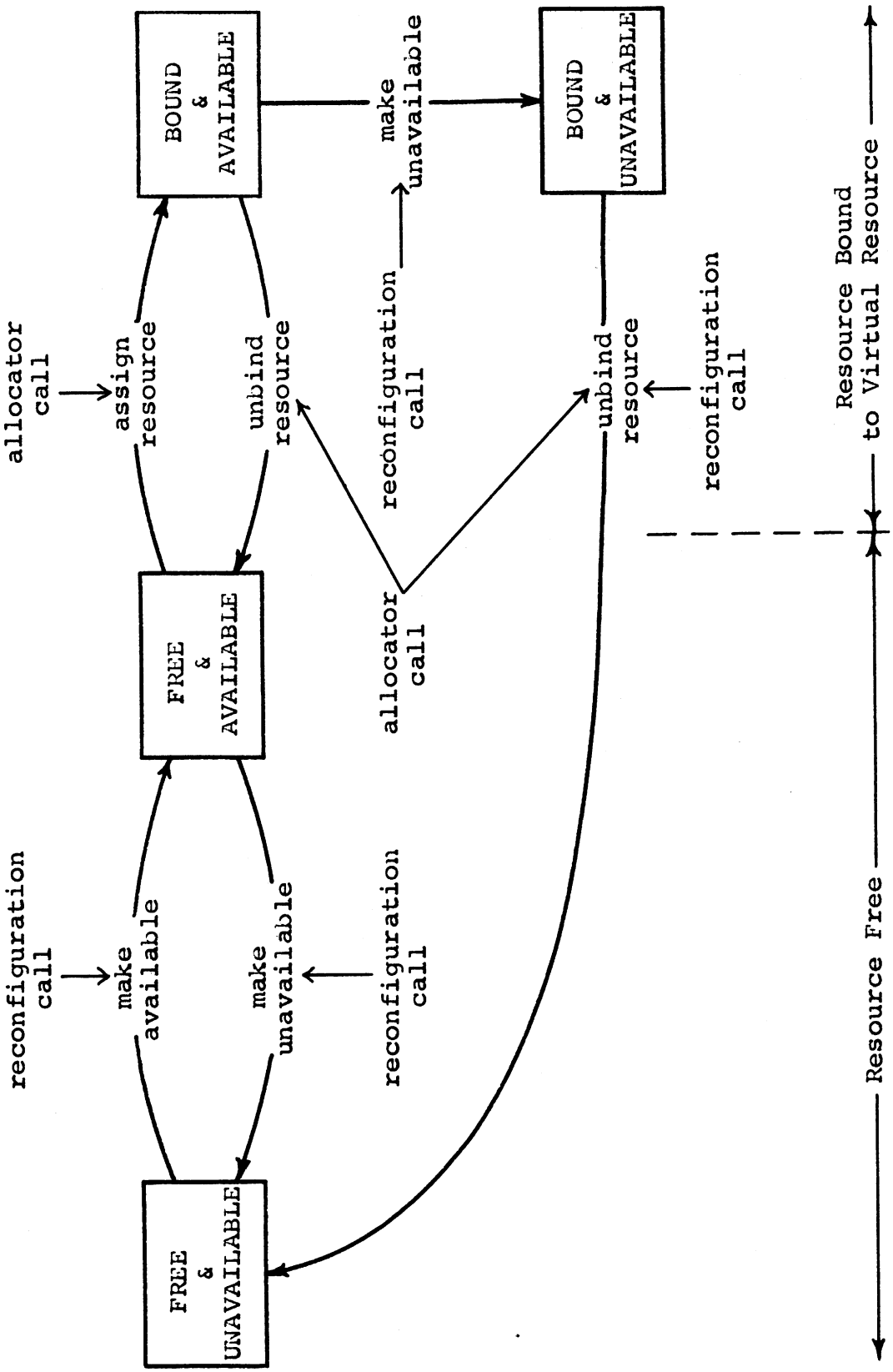


Figure 3.1 -- Usage State Transitions of Logical Resources

regardless of reconfiguration capabilities, we expect to find. Then the reconfiguration primitives are presented in terms of closed subroutines and system-wide data bases.

3.1 Storage Resource Management Environment

The purpose of this section is to provide a fairly extensive review of the technology currently available for storage management. This review focuses on those features which tend to solve storage reconfiguration problems.

The strategy used to manage the use of storage resources has a significant impact on the feasibility of reconfiguration -- particularly on the ability to remove a storage module. Removing a storage module removes some range of absolute addresses (which are modeled as logical resources) from use, and the system must provide, in some other module, a valid copy of the stored information. A basic problem is insuring that all references to this information are directed to the new location. We will consider some common examples of (primary) memory management to illustrate the storage allocator characteristics important for reconfiguration. To identify the intrinsic problems we first assume an unbounded amount of available primary memory, and then we consider the technological problems introduced when there is a limited

storage capacity.

3.1.1 Allocation With Unlimited Storage Capacity

First we examine one of the simplest examples of storage management -- the textbook batch processing system. The system loads a program into a contiguous block of primary memory at a known absolute address, and binds all relocatable addresses to the absolute addresses that resulted from loading: typically the program is modified by inserting the required absolute addresses. We now consider the problem of removing the primary memory module containing this program, after this program has begun execution.

One might naively think that the necessary steps are merely interrupting the execution, moving the instructions and data by some increment of absolute address to a new location, and restarting the execution. Obviously the absolute addresses originally generated by the binder are going to be incorrect; on the other hand, since the binder initially found where absolute addresses were needed, the system should now be able to go back and add to these addresses the appropriate address increment. However, the previous execution may also have stored addresses elsewhere as data (e.g., return points for subroutine calls) for future use. Thus we conclude that the system needs a method for causing the absolute address of all future memory references to be incremented by the amount

the program was moved. To do this the operating system must be able to locate and modify every occurrence of a stored absolute address that may be used in the future.

We need not invent methods of doing this since various relocation techniques have already been developed for reasons other than reconfiguration. Multiprogrammed batch processing systems provide a common example. These systems are designed to execute one program until the process cannot proceed (e.g., until the process must wait for some input/output operation), and then switch the processor to the execution of some other process that is able to proceed. To this end, the system loads more than one program into primary memory at once and chooses the "best" one to run. If one program waits for a long time (for example while an operator locates and mounts a magnetic tape) then for efficiency the system may unload this program from storage so some other program can be loaded; however, when ready to continue the execution the same block of memory may not be available, so the system must relocate the program to a different absolute address in core. One way to accomplish this relocation is with a relocation base register in the processor hardware: all addresses generated for instruction fetches and data are relative to this relocation base. The addresses appearing in the program are no longer absolute addresses but are virtual addresses -- the relocation base register provides an address mapping that binds virtual addresses to absolute addresses.

We note that the relocation base register is the only memory of any absolute address in the system, and therefore in terms of our model completely specifies the binding of virtual storage resources to logical storage resources (named by absolute addresses). The system can relocate an executing program by the following steps:

1. Stop the execution of the program so that the data and instructions in this block of memory will not be accessed, viz., stop the progress of the process so its state will not change.

2. Make a copy of the block of instructions and data at a new absolute address outside the memory module being removed.

3. Reset the relocation base register to reflect the address of the new copy.

4. Resume the execution of the program, viz., continue the progress of the process from this new state which is equivalent to the state when it was interrupted.

3.1.2 The Storage Utilization Problem

So far we have seen that the intrinsic problem in removing a storage module is reversing the binding of a virtual address to an absolute address, and then binding to a new absolute address -- in the example, the relocation base register provides this ability. Although the basic issue has

memory the system assigns the number of storage blocks required. These blocks may not be contiguous even though the virtual addresses in the corresponding pages of the program are.

The processor requires a page table accessed by page number that gives the absolute address of the corresponding block of storage (typically the page size is a power of two, so that the page number consists of some high order bits of each virtual address). Each page table word provides a mapping of virtual addresses in that page to absolute addresses in exactly the same manner that the relocation base register did. The page table may itself be located in storage if the processor has a page table base register containing the page table's absolute address. For each instruction fetch or data reference the processor consults the appropriate page table word to arrive at the correct absolute address. Figure 3.2 illustrates the address translation for paging. The system can remove a memory module by making a copy of just the pages stored in it: the page table word for each page moved (rather than just the value of a single relocation base register) must be updated to the new absolute address. It is again emphasized that paging is not essential for reconfiguration; however, for simplicity we will continue to use paging to decouple the problem of storage utilization, allowing a clearer view of the basic issues of

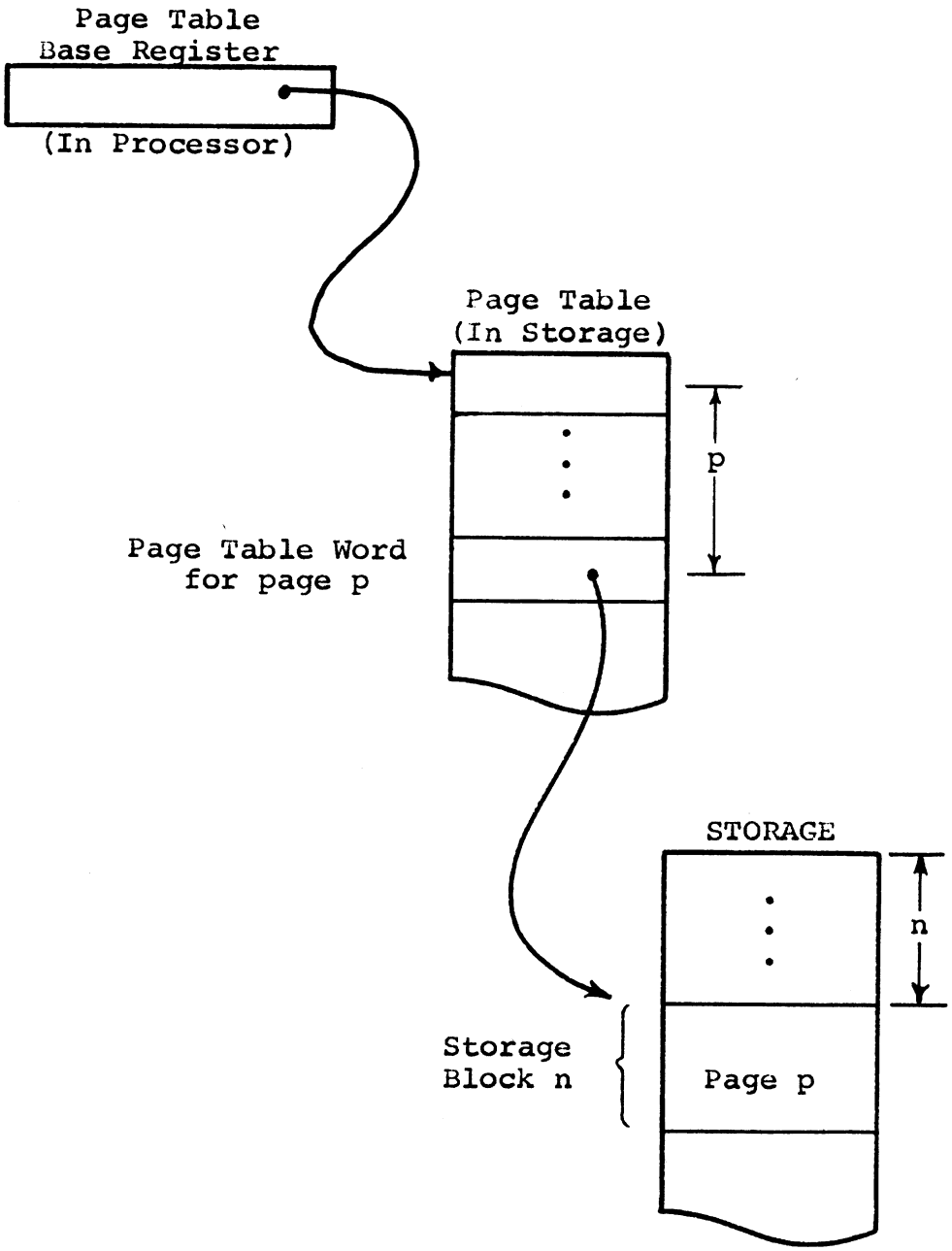


Figure 3.2 -- Address Translation for Paging

reconfiguration.

3.1.3 The Effect of Limited Storage Capacity

Although paging answers the technological problems of storage utilization, there remains the problem of limited storage capacity. Similar modules may not have adequate storage for a copy of the information in the module being removed. Again we note that similar problems exist and have been solved for reasons not related to reconfiguration. Recall that in our earlier discussion of multiprogrammed batch systems, when a program not yet in primary storage was ready to run, the operating system could get space for it by unloading some other program currently in primary storage but not able to continue execution. Implicit in this is the existence of what we have called secondary storage modules (typically magnetic drums or disks) which have adequate room for copies of the programs "in progress" whose execution has been interrupted. Similarly, to remove a primary memory module the system can either unload programs in the module or unload programs from other modules to make room for a copy of the information from the module being removed.

Where paging is used, reconfiguration can benefit from the technique of <u>demand paging</u>. In a system with demand paging, an entire program is not necessarily loaded at once, but the storage allocator loads only selected pages. One

common design has a "fault tag" in each page table word without a valid absolute address (viz., the page is missing in primary storage). Any attempt by the processor to reference a "missing page" will cause a hardware fault. As a result of the fault, the allocator retrieves the page from secondary storage, places the absolute address in the page table word, removes the "fault tag", and then allows the execution to continue. If no free storage block is available, then the allocator copies some page currently in primary storage (for example the least recently used page) to secondary storage, so its storage block can be allocated to meet the new demand. Similarly when removing primary memory, demand paging provides a tool for pushing out excess pages so that the number of pages in primary memory does not exceed the storage left after removing the module. Thus the basic solution to the problem of limited primary storage capacity is moving information to secondary storage modules, and a storage allocator with demand paging can make this easier. Secondary storage must, of course, have enough capacity to hold the information moved from primary storage.

3.1.4 The Storage Allocator

Recall that the objective of the above discussion of storage allocator designs has been to identify characteristics that are relevant to reconfiguration. This has been motivated

by a desire to benefit from the existing state of the art rather than reinventing existing features. In addition, the significance of the storage allocator to reconfiguration should now be clearer. For simplicity the discussion so far has been primarily directed to primary memory -- this needs to be generalized to encompass any other type of storage module.

Although the previous discussion of demand paging was framed in terms of primary memory, the essential concepts are in fact common to other types of storage modules; however, the usual terminology is somewhat different. File systems typically manage secondary disk and drum storage, for example, by considering each module to be composed of a number of independent fixed length "records" -- the records are directly analogous to the blocks of primary memory. To store the information contained in a "file", the required number of records are assigned and a "file map" is maintained to show the address of each record of the file -- a file map is directly analogous to the page table for a program, and a file may be just a secondary storage copy of a program. Not only may the system copy information from primary storage to some device such as a drum, but also various types of storage modules may be arranged in a hierarchy. The file system may move records from one secondary storage module down to another secondary storage module in order to make room higher up in the hierarchy. For example the least recently used record may be kept on the type of module with the longest access time.

The significant observation is that the storage allocator characteristics identified as relevant to reconfiguration are applicable to both primary and secondary storage modules.

Since the above discussion has identified the allocator characteristics important to reconfiguration, it is now appropriate to introduce a very specific allocator model to facilitate our discussion. The Multics file system will be used as a case study, and we will discover that it has all the desired characteristics. This particular design is chosen because it includes a practical and currently working example of a demand paging storage allocator whose motivation and details are available to the interested reader from several sources [1, 6, 21, 22], and because the research reported in this thesis includes implementation of reconfiguration capabilities for this specific system.

The Multics storage allocation design includes a feature known as segmentation: rather than considering a process in terms of a single program (with instructions and data), a Multics process has multiple segments. Therefore, each virtual address has two dimensions -- a segment number and an offset (address) within the segment. Each segment has its own page table, and the two-dimensional address space of each process is defined by a descriptor segment, which is basically a table (indexed by segment number) of segment descriptor words giving the absolute address of the page table (in the same manner as a page table base register) for each segment.

The descriptor segment is itself a segment in primary memory and the processor has a descriptor segment base register containing the absolute address of the page table for the descriptor segment. Figure 3.3 illustrates the address translation for segmentation. Although segmentation itself does not solve any reconfiguration problems, the significant observation is that segmentation specifically contains the relocation capabilities needed for reconfiguration.

Now an understandable reaction might be that with all these levels of indirection it is certainly inefficient for the processor to translate a logical address into a reference to the ultimate absolute address of interest. Multics reduces the number of memory references needed to complete the indirection by providing a high speed associative memory within the processor. This memory maintains the sixteen most recently used page table words or segment descriptor words [23].

First we note that the associative memory distorts our model of a processing module as a module with no memory of data common to more than one process: we will see later that this is a problem that has to be specifically dealt with. Aside from this one problem area, all absolute addresses in the system for the location of instructions and data are still in page table words as discussed for demand paging. In addition there are absolute addresses of page tables in segment descriptor words and the descriptor segment base

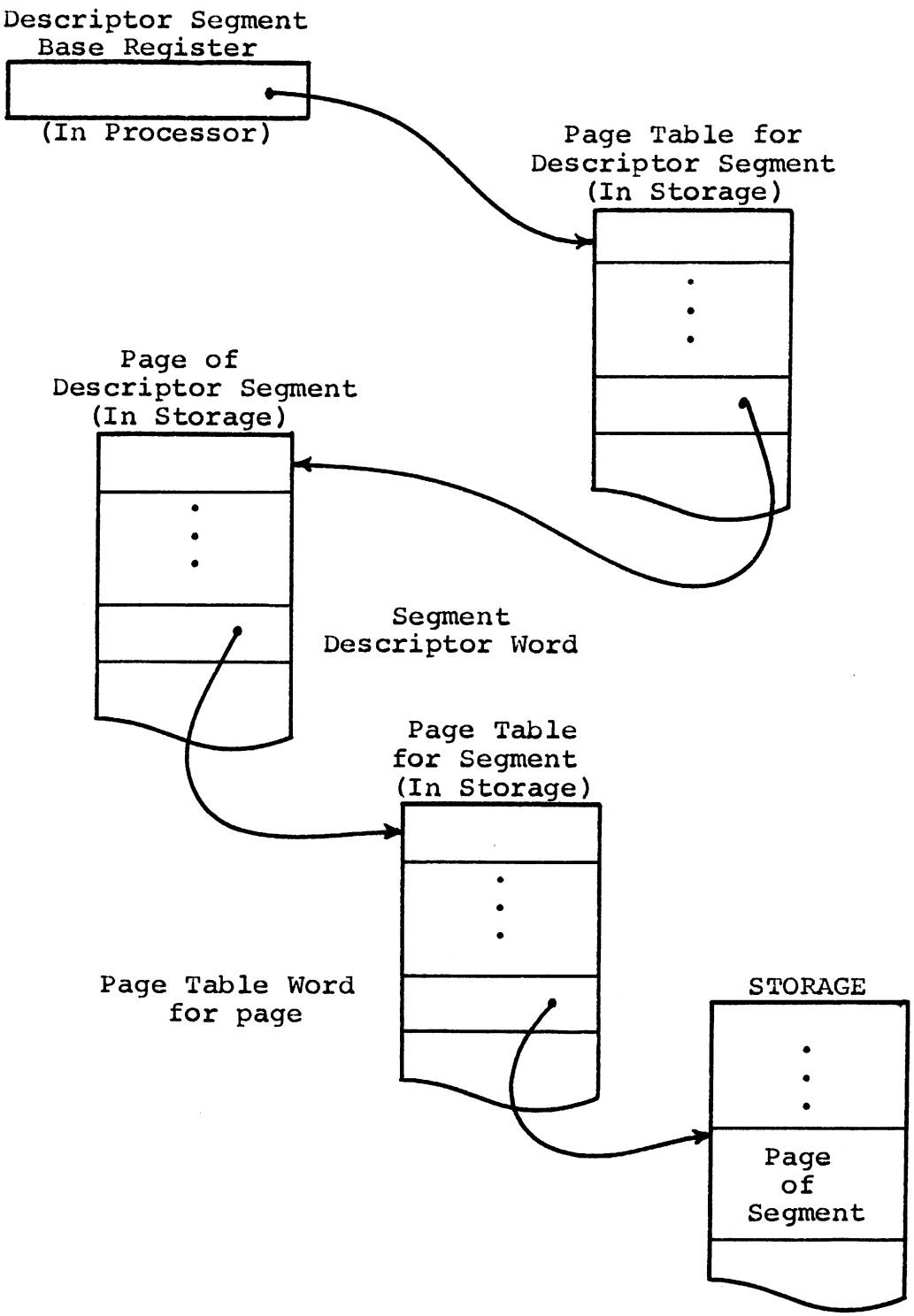


Figure 3.3 -- Address Translation for Segmentation

register. However, the file system maintains all absolute addresses and can locate and update them if required. The file system maintains for each type of storage module a storage map that allows the system, for any absolute address, to determine that the storage block is either in use and the appropriate page table (or file map) entry can be found, or is free and can be used to meet demands for storage, or is not part of the available resources managed by the file system.

Although the Multics file system includes storage allocator features not directly applicable to reconfiguration, in this case study we have seen that Multics does provide the relocation capability essential to reconfiguration, and demand paging is available to solve the technological problems of limited storage capacity. We will refer to the Multics file system as a specific model of storage allocator design when useful as an aid to the presentation.

3.2 Removing a Storage Module

The preceding discussion has established a storage management environment for reconfiguration. We now develop a design for removing a storage module from the resources being used by the system. Basically, while the system is running, reconfiguration primitives move the stored information and then locate and update absolute addresses to reflect the new location. A major goal is providing a general design which is

(as much as possible) independent of the kind of information stored in the removed module.

We digress a moment to observe that during reconfiguration the system needs information about each module in the configuration. Therefore, we introduce the module configuration table as the primary reconfiguration data base, and we will identify the information contained in it as we discover the need. The system creates this module configuration table during initialization and updates it during reconfiguration. From our general model we can anticipate that the module configuration table will reflect the binding of physical resources to logical resources -- in the case at hand, the table relates each physical storage module to a range of absolute addresses.

Now let us suppose that an operator requests removal of a specific storage module. The reconfiguration procedures of the operator's process must first verify that a viable configuration will remain. For example, after removing the module the minimum amount of storage for this type of device must remain, and the module must not be essential for relaying signals between processing modules. Since motivations, such as preventive maintenance, for removing a module are associated with physical hardware entities, it is convenient for the operator to identify the module by a (usually arbitrary) physical name; therefore, we extend the module configuration table to include for each module the name used

by the operator.

To remove a storage module in response to the operator request, first the system makes certain that the storage is not available to meet any future storage demands, and then the system frees any storage already in use, by moving the information to another module. In terms of the model outlined in chapter two and illustrated in Figure 3.1, the system invokes reconfiguration primitives to change the usage state of the logical storage resources to "free and unavailable". For each unit of resource in the module, the system first invokes the generic subroutine Make_unavailable (defined in chapter two), and then invokes the generic subroutine Unbind (also defined in chapter two).

From the notion of a logical resource usage state of "available" we may invent immediately the "available list", a list of all logical storage resources available to the allocator for use in meeting storage demands. The Make_unavailable primitive prevents future allocation of storage by removing it from the available list. Using the notation of the PL/I language, we now introduce the first specific instance of the generic closed subroutines for reconfiguration:

```
call Make_unavailable (addr);
```

where "addr" is the name (e.g., the absolute address) of the unit of logical storage resource to be moved from the available list to a "removing list". The unit of storage

resource must, of course, be uniquely identified: if the names themselves are not unique (e.g., the same absolute address may occur in different types of storage devices), there may in fact be more than one distinct Make_unavailable subroutine (e.g., one for each type of storage), but without loss of generality we will consider only one.

For an example of the Make_unavailable primitive, consider systems that use paging, such as Multics, where the potential set of logical storage resources are easily represented by a storage map -- a table indexed by storage block number (computed as the absolute address of the base of the block divided by the block size). As illustrated in Figure 3.4, each entry in the storage map either has a pointer to the page table word containing the storage block address, or has a "null" pointer indicating that the storage block is free. Since there may be some absolute addresses that cannot be referenced (i.e., are unavailable), a threaded "available list" is constructed from the storage map entries to reflect the pool of storage blocks actually available to the allocator. The Make_unavailable subroutine simply threads the indicated storage block entry out of the "available list" and threads it into a "removing list". The usage state (viz., "unavailable") must be recorded; for example, when the allocator frees the storage block for a deleted page, an "unavailable" entry is not threaded onto the "available list" as might otherwise be the case. Figure 3.5 illustrates the

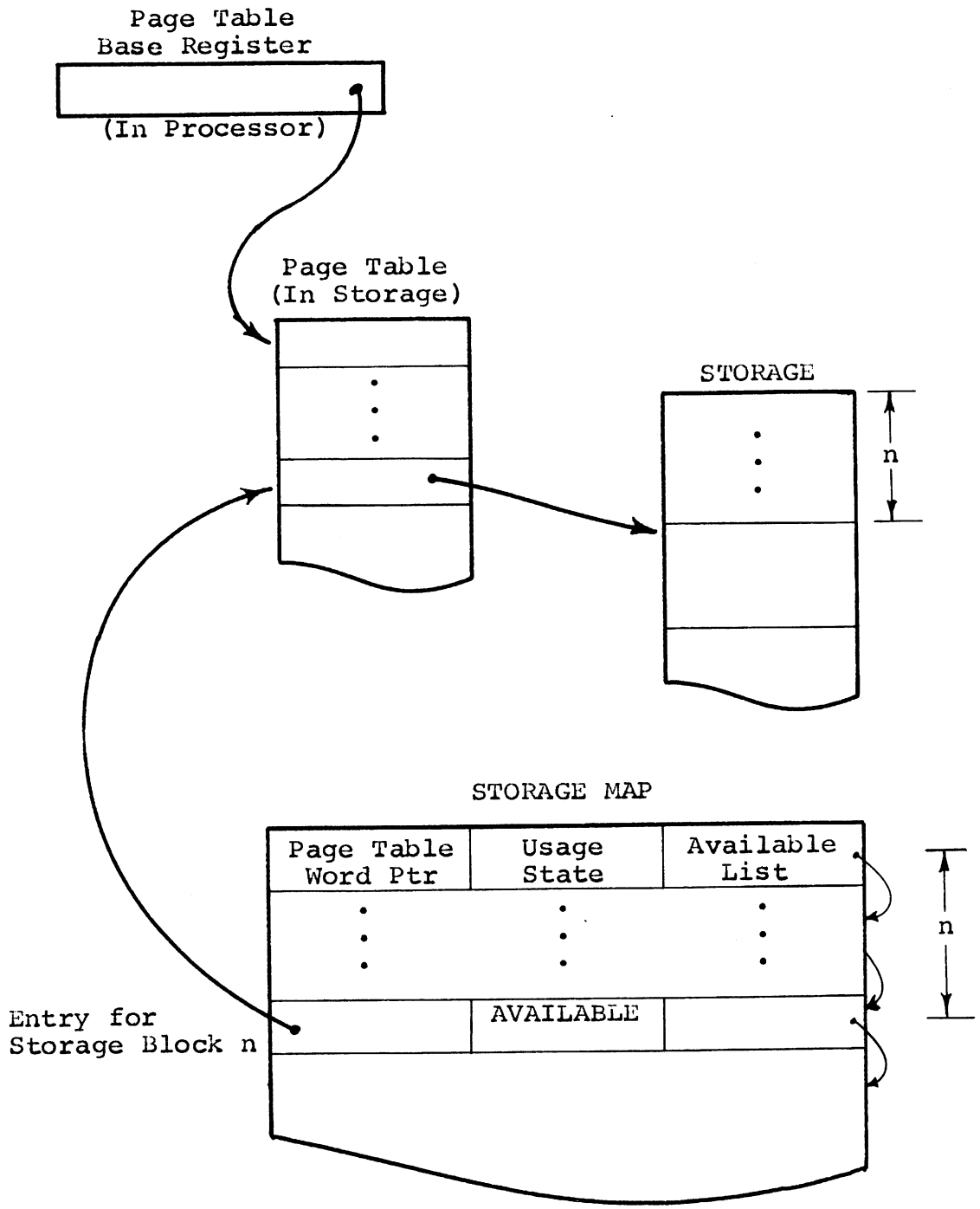


Figure 3.4 -- Example of a Storage Map

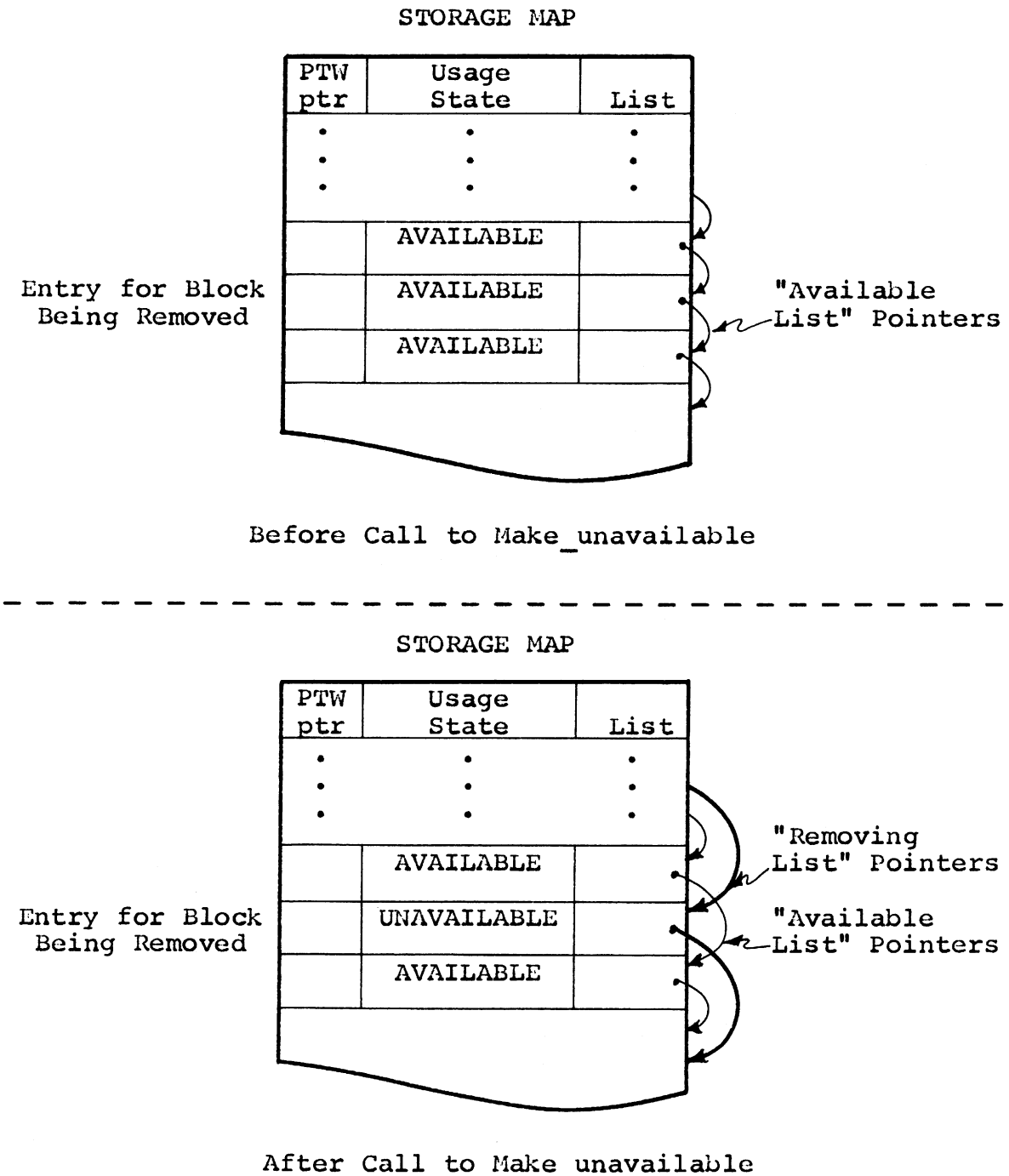


Figure 3.5 -- Example of Making Storage Unavailable

bound storage -- "demand allocated" and "wired". This class is recorded in each entry of the removing (or available) list. The techniques Unbind uses to free the storage depend on the class and are discussed in detail in the following two sections. These possible classes of bound storage are defined as follows:

1. Demand allocated storage is currently used to store information in response to the demand of some process, but the allocator may without the explicit "consent" of this process move the information (viz., reverse the binding to storage).

2. Wired storage has been allocated to store information which must at all times have a valid binding to (primary) storage. For example, the handlers for interrupts and missing page faults are typically in wired storage.

3.2.1 Freeing Demand Managed Storage

Although wired storage can only be in a primary storage module, both secondary storage modules and primary storage modules (such as a magnetic core or semiconductor memory) can contain demand allocated storage: typically only portions of the supervisor use wired storage, and all user programs are demand allocated. Unbind will free demand allocated storage by invoking the system's normal allocator functions to move the information to a new location.

For an example consider the demand paging environment,

such as that of Multics. The previously discussed storage map entry is extended to indicate the state -- free, demand allocated, or wired -- of the storage block. Now in the demand paging environment, we can say that Unbind merely "pages out" all the demand allocated primary storage on the removing list. First the allocator primitive (called by Unbind) places a "fault tag" in the corresponding page table word to prevent modification to the page during the copy operation. Then the allocator signals a processing module (viz., a channel) to copy the information to a free block of secondary storage. It is handy to set an "out of service" flag for the block to show that the copy operation is in progress. When the channel signals that the copy is completed the storage map is updated to indicate that the block is free. It is the ability to use the fault tag as a lock to deny immediate access to a page that permits "paging out" and thus distinguishes the storage as demand allocated, as opposed to wired storage which always requires a valid primary storage address for the page.

Now recall that in a Multics central processing unit a copy of a page table word may be maintained internally in an associative memory. When a page table word is modified the multiple copies that may be present in other processing units introduce a synchronization problem. For example, a fault tag in the page table word is used to prevent modification to a page while the page is being copied to secondary storage.

However, when another processor happens to be using the same page, then that processor's associative memory copy of the page table word has no fault tag, and that processor continues to reference the page. Therefore, after setting the fault tag, the allocator must signal all processors whose associative memories contain this page table word, to clear the invalid page table word from their associative memories. The allocator can safely initiate the copy operation only after each processor signals back that its associative memory has been cleared. The associative memory causes the processor to deviate from our ideal model of a processing module, but this does not prevent reconfiguration. The price to be paid is that there must be communication between processing modules in order to reverse the binding to primary storage.

For secondary storage the techniques are basically the same, although processing module hardware typically cannot directly move information from one secondary storage module to another secondary storage module. The system will usually move information from secondary storage to primary storage and then from primary storage to a free secondary storage location. In any case the intrinsic problem is relocation -- moving the information and changing all references to the secondary storage location. We previously noted that one technique is to provide a file map giving the secondary storage location of each unit of virtual storage. This is conceptually similar to the primary storage page table -- in

terms of our general model, both represent binding between pages of virtual memory and blocks of storage. Since the file map is the only place where a secondary storage address is used, a given address can be searched for and updated -- a storage map can be used to aid the search.

In the above discussion we have identified the resource management techniques that can be applied to "demand allocated" primary and secondary storage in order to leave the storage in a "free" state. Basically the reconfiguration subroutine Unbind invokes the system's normal allocator primitives to free demand allocated storage -- there remains the problem of freeing wired storage.

3.2.2 Freeing Wired Storage

Nearly every operating system has a minimum set of instructions and data that must always reside in primary memory. In other words, there is information that some process expects to always have a valid binding to primary storage -- the storage allocated to this information we have termed as "wired" storage. For example, wired storage is typically used for interrupt handlers, for the procedure that handles missing page faults and, as we will see, for the relocation procedure itself. The intrinsic problem is, of course, still one of relocation; however, the unsolved problem is avoiding conflicts while the new copy is being made

and the relocation mechanism is updated.

3.2.2.1 Relocating Wired Information

First we will indentify the basic operations of Unbind, and then we will consider the problem of avoiding conflicts; we defer until later the special cases when Unbind moves itself or moves part of the relocation mechanism. Unbind must move information from one absolute location to another absolute location. But recall we already concluded that references to stored information should use virtual addresses instead of absolute addresses. Rather than propose an "absolute mode" of operation for Unbind, we will see how the system's normal relocation mechanism can be used. This will be discussed in the context of a paging environment.

The virtual memory of the reconfiguration process has two pages of particular interest to Unbind: the "real" page of information to be moved and a "shadow" page used as a temporary work area. This "shadow" page will eventually contain the moved version of the wired "real" page. Since the allocator knows the location of all absolute addresses, Unbind can use its input argument (viz., the absolute address of the storage block to be removed) to determine which page of the virtual memory is the "real" page. Unbind also invokes the normal allocator functions to assign primary storage for a temporary "shadow" page. The address mappings at this point

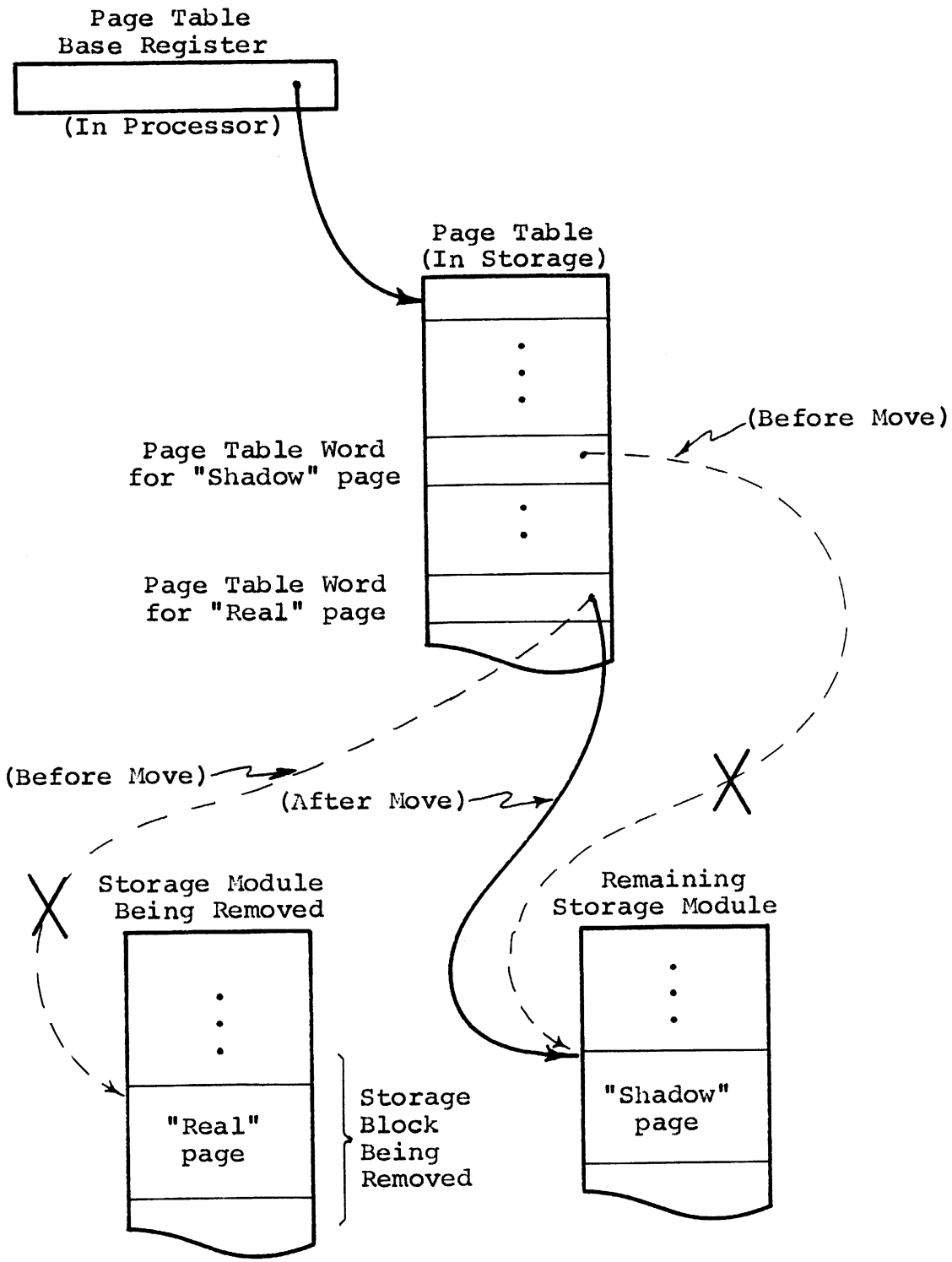


Figure 3.6 -- Environment for Moving Wired Storage

(before relocation) are shown in Figure 3.6.

Unbind now copies the information from the "real" page to the "shadow" page of the virtual memory. To complete the relocation, Unbind places the absolute address of the "shadow" page in the page table word for the "real" page: the page table is referenced as a normal part of the virtual memory. Figure 3.6 also shows the address mappings after relocation. Unbind then updates the allocator data bases (e.g., the storage map) to reflect that the desired storage block is free and that a new storage block has been allocated to the "real" page; the "shadow" page has no storage allocated to it. We note that Unbind is an extension of the allocator design and must, of course, follow the normal allocator conventions -- for example, check a lock before modifying the allocator data bases. This completes the basic design of Unbind for wired storage. There remains the problem of avoiding conflict.

3.2.2.2 Avoiding Conflict

Since our goal is to reconfigure a system which is still running, Unbind introduces two types of potential conflict with other user processes. While the reconfiguration process is reading (viz., moving to a new location) information in a block of wired storage, some other process may modify the storage; therefore, the new copy may be outdated or inconsistant. On the other hand, while some other process is

reading an absolute address from a page table word (viz., resolving a virtual memory reference), Unbind may be modifying the address to reflect the new location. The "obvious" solution is to provide explicit locks for all information requiring wired storage. This solution lacks generality since it requires Unbind to keep track of all the locks. In addition, checking the lock involves a processing overhead for all functions using wired storage. Instead of explicit locks, our basic approach will be to "stop" potentially conflicting processes during the execution of Unbind.

One possible way for Unbind to avoid conflict readily suggests itself: before starting the copy operation, Unbind sends a signal (viz., interrupt) to all other processing modules that causes them to "halt" until Unbind signals them that the copy operation is completed. Unbind also inhibits all interrupts and faults to prevent conflict with an implicit interrupt or fault handler process. This synchronizing approach solves the intrinsic conflict problem, since it temporarily reduces the system to a single process. However, we are interested in an effective engineering solution, and since forcing all other processors to stop for the entire duration of the copy operation can be unnecessarily wasteful and does not scale well with more processors, we will examine an alternative design.

Although some other processor has the <u>ability</u> to modify information being relocated, there may be a low probability it

will actually do so while Unbind is executing. In a demand paging environment it is useful, regardless of reconfiguration capabilities, to know when a page in primary memory has actually been modified -- for example, unmodified primary storage pages need not again be copied to secondary storage. The hardware can provide for a "modified bit" for each storage block that is set on by any processing module making a modifying reference. The processing hardware (channels and central processing units) of some contemporary computer systems, e.g., the IBM 360 Model 67 [10], include this capability.

With hardware that provides such a "modified bit", Unbind can use what we term the "trial copy method". The trial copy method has four basic steps:

1. Unbind turns the "modified bit" off for the storage block being removed.

2. Unbind copies the information to the new storage block.

3. Unbind tests the "modified bit".

4. If still off then Unbind updates the absolute address to reflect the new location; otherwise, return to step 1.

Any specific implementation of this trial copy method must consider three design questions. First, if another process can modify the old storage block after step 3 but before step 4 is complete, we have a critical race: this modification is never reflected in the new copy, and so is

essentially lost. Again we can solve the problem by signalling all other processors to halt, but now they are stopped for a much shorter time period. In fact, a suitable "conditional store" instruction for the processor could perform both steps 3 and 4 as one indivisible operation, without any explicit signalling.

Secondly, the references of a process are usually localized, so if a page has been modified while attempting to copy the page, the (conditional) probability that it will be modified again is much higher. For this reason and in order to place an upper bound on the processor resources required by Unbind, it may be desirable to stop the other processors for the second try.

Finally, if the "modified bit" is used by procedures other than Unbind, then the "modified bit" is part of a system-wide data base, and Unbind must follow system locking conventions to prevent other processes from explicitly referencing the "modified bit" being manipulated by Unbind. In addition, Unbind must leave the "modified bit" for the new storage block in the correct state: only if the information was modified before or during the relocation should the "modified bit" be <u>on</u>.

3.2.2.3 Self-reference Problems

We have developed a design for removing wired storage that is generally independent of what information is stored; however, the design of Unbind must consider two special cases of wired storage required for the operation of Unbind itself.

First, the storage being removed may contain instructions and data of Unbind itself. Now recall that to prevent conflict with implicit handler processes, Unbind cannot allow any interrupts or faults during its execution. This implies that, to avoid "missing page" faults, the instructions and data of Unbind must be in wired storage. Therefore, when moving wired storage Unbind may move itself, and in particular may move its internal variables. Now if, because of its execution, Unbind modifies an internal variable (for example, a loop index) it is moving, then the new copy of the variable will not be correct if it is modified after it has been copied. Again (as in conflicts with other processes) an outdated or inconsistant copy results when the information is modified (in this case by Unbind itself) while being moved. Unbind can avoid this conflict-like problem by modifying only internal registers of the processor during the copy operation. If using internal registers is impractical, Unbind must specifically check for this special case and explicitly update the new version after the relocation is completed.

A second, particularly awkward special case occurs when

the relocation mechanism used by Unbind (when storing the new address in the page table word for the "real page) is part of the information being relocated. This special case occurs when Unbind moves a page table which contains the page table word for the page (containing the page table) being moved. The example of Figure 3.6 is redrawn in Figure 3.7 for this special case (after the copy is made but before updating absolute addresses). Recall that Unbind usually updates the absolute address in the page table word for the "real" page using a normal virtual memory reference. However, this virtual memory reference to the page table word will reference the old copy of the page table, leaving the old address in the new copy of the page table. Therefore, Unbind must explicitly update the page table word in the "shadow" page in order to complete the relocation (as shown in Figure 3.8).

3.2.2.4 Implications of External I/O

We have developed a design for storage removal based on an asynchronous model of the system. In particular, Unbind must be able to "stop" all processing modules referencing wired storage in order to avoid conflicts. Unfortunately external I/O channels are used for synchronous operations, and therefore cannot in general "stop" for an arbitrary length of time. However, I/O channels usually include some sort of "buffering" that allows them to stop for a bounded period of

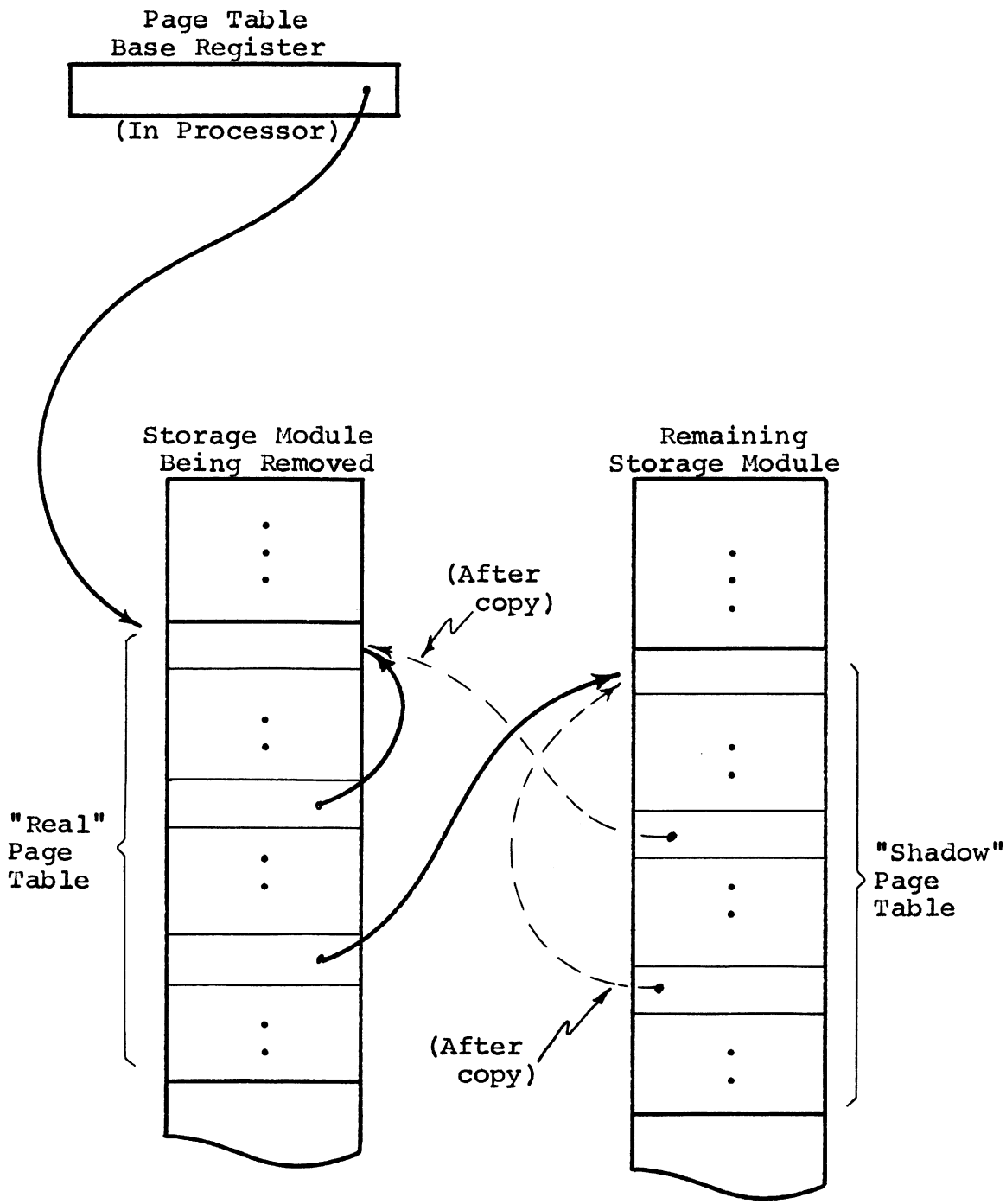


Figure 3.7 Environment Before Relocating Page Table

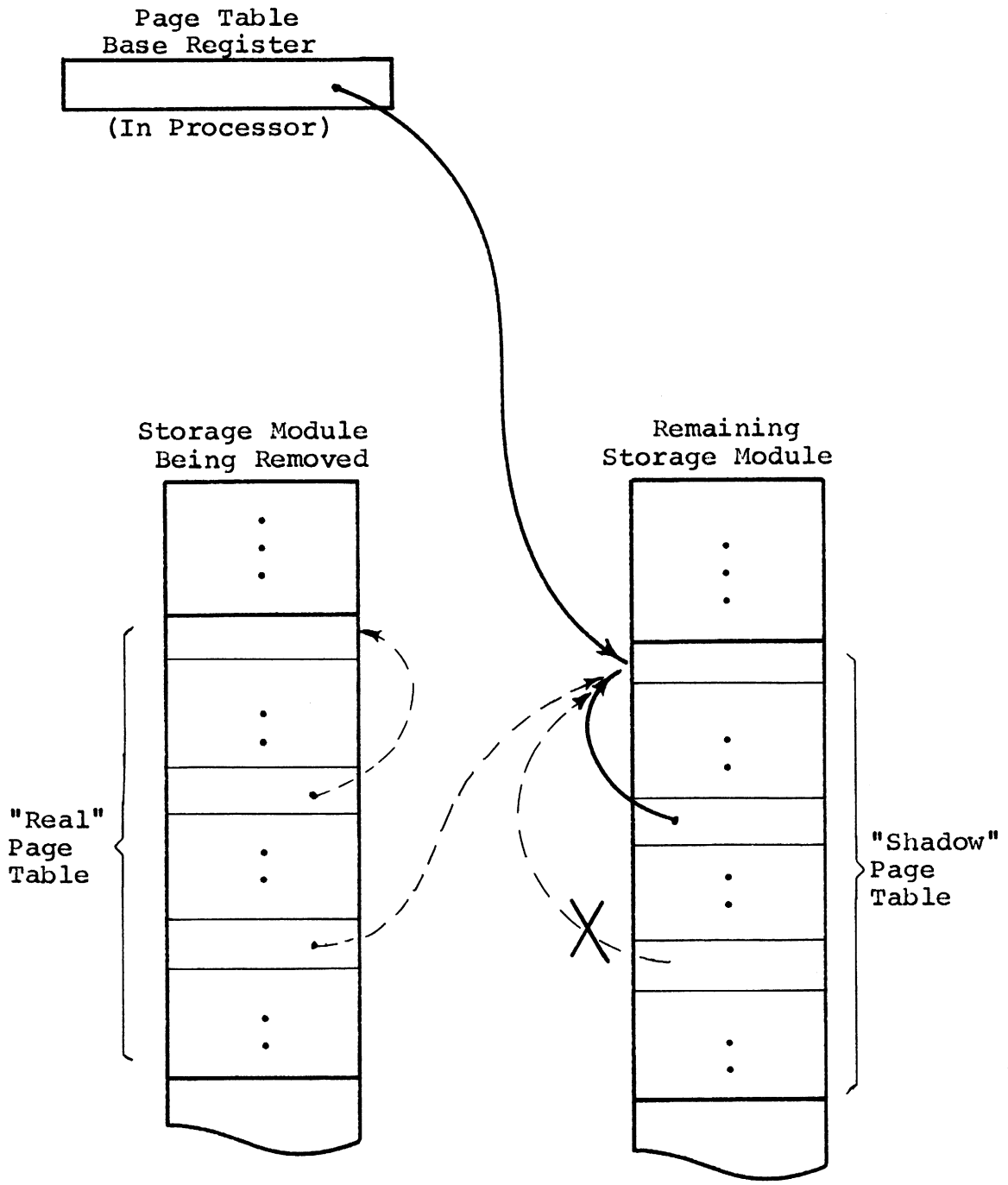


Figure 3.8 Environment After Relocating Page Table

time. As already noted, with the trial copy method this time period can be kept short (if a complete copy can be made without the information being modified); however, the trial copy method implies that an I/O channel updates "modified bits" and has a convenient relocation mechanism. In fact, many contemporary I/O channels use absolute addresses directly, and make no record of storage modification.

As already noted, a detailed consideration of external I/O is beyond the scope of this thesis; however, a few observations about the I/O impact on storage removal seem useful. Operating system I/O primitives often know every place that an absolute address is used by the channel. Even though there is no general relocation mechanism, this knowledge may allow each channel to have a special case primitive to aid the removal of storage used by that channel.

The I/O channels of a system can often be classed as either "high speed" or "low speed". For "high speed" channels, any operation in progress is guaranteed to be completed soon. The reconfiguration primitive for "high speed" channels can insure that the future I/O operations will use only storage outside the module being removed for such things as buffers. The primitive can also signal the reconfiguration process when any current I/O operation is completed (i.e., when the channel is "stopped"); special per channel control words, etc., that use absolute addresses can be moved without conflict while the channel is stopped.

Although it is not practical to wait for "low speed" channels to complete, the data transfer rate is usually so slow that essentially synchronous primitives can be designed to change the storage used, for example by a "fast" forced stop and restart of the channel. Finally, for a fast channel that takes arbitrarily long to complete an operation, we can only suggest that the channel be designed with a hardware relocation mechanism, such as paging, and be designed to "stop" long enough for Unbind to complete relocation.

In summary, the Unbind design we have developed can usually be applied to removing storage used by external I/O channels, when these channels include the same general relocation mechanism as other processing modules. Without an explicit relocation mechanism, we can only say that for many practical systems it seems likely that special case solutions to the intrinsic relocation problem can be formulated.

3.2.3 Freeing a Storage Module Used as a Relay

We have identified primitives to remove from use the units of logical storage resource in a module. However, the reconfiguration procedures must also insure that a storage module is not used as a relay for signals between processing modules. For purposes of reconfiguration we can view the storage module as also being used for a small specialized memory -- the information stored is an intermodule (interrupt)

signal or possibly a "mask" to prevent the relaying of a particular signal.

The processing module(s) to receive each signal is determined by the configuration of the hardware modules and is recorded in the module configuration table entry for each storage module. The system design includes a convention for using storage modules to address a particular signal to a specific processing module -- we invent the notion of a signal address table to reflect which storage module is used as a relay. When a storage module is removed, the reconfiguration procedures make it unavailable as a relay by replacing all its occurrences in the signal address table with some remaining module. The stored signal information in the removed module could then be copied into its replacement, just as for other storage in the module. However, updating the signal address table prevents further writing into (i.e., sending signals to) this module, so it is reasonable to just wait until all the signals are sent to processing modules.

Multics provides one implementation example that demonstrates the reconfiguration features. A single primary memory module is used to relay all signals to each central processing unit (CPU) from any CPU. The system uses an explicit table for addressing signals to a particular CPU, and this is updated by reconfiguration procedures. Since signals are stored in a storage module for only a short time (viz., typically a few milliseconds), the signal information is not

actually copied, but the reconfiguration process just waits until no stored signals remain. Signals to and from I/O channels may use any primary storage module -- the module used is determined by the location of certain control data. Thus when storage is freed (i.e., the data is moved), it is implicit that the module is no longer used as a relay.

3.2.4 Summary of Storage Module Removal

We have considered the problems of dynamically removing a storage module from use by the system, and have identified a design approach to solve each of these problems. The basic problem is moving and relocating all references to information stored in the module, and we have seen that a demand paging environment can simplify the implementation.

It is emphasized that relocation is an intrinsic problem that must be solved; a system design that does not consider dynamic reconfiguration can thwart solution of this problem. For example some systems implicitly assume and always require a zero-based and contiguous absolute address range, thereby severely restricting dynamic memory reconfiguration. In addition some systems include "wired-in" reserved absolute storage locations: examples are the location where the processor state is saved at the time of an interrupt and the place where an I/O channel expects to find its next command. Basically any explicit or implicit use of a fixed absolute

address can restrict dynamic memory reconfiguration.

3.3 Adding a Storage Module

A request to add a storage module is satisfied by making all the storage in the module available for use by the system. Recall that in this chapter we are assuming that a storage module to be added is idle but accessible to the system. Accessible means that a range of absolute addresses is assigned to the storage module, and each processing module has a communication link enabled to the storage module. In terms of our general model described in chapter two and illustrated in Figure 3.1, the logical storage resources in the module are initially in a "free and unavailable" state. To add the storage, the system must invoke the Make_available primitive (identified in chapter two) to perform the operations represented by the state transition to the "free and available" state.

Let us suppose that an operator requests that a specific physical module be added. Recall that the module configuration table reflects the binding of physical to logical resources, and thus permits the reconfiguration routine to determine the range of absolute addresses to be added. First the system verifies that the operator's request is acceptable. For example, the module configuration table must have an entry for the specified module (i.e., the module

must in fact be accessible to the system), and the module must not already be in use. For each unit of logical storage resource in the module the operator's process invokes the Make_available primitive.

Recall that the "available" list defines the set of logical resources currently available to the storage allocator. Therefore, we provide a specific instance of the generic closed subroutine Make_available to add a unit of storage to the available list:

```
call Make_available (addr);
```

where "addr" is the name (viz., absolute address) of a unit of logical storage resource. Make_available merely adds the storage to the available list, and the usage state is recorded as "free". Thus, when Make_available returns the storage is available for use by the allocator to meet future demands for storage.

After all the units of storage have been added, the system records the module in the module configuration table as available. Whether or not storage modules can be used to relay intermodule signals is immaterial, since adding a storage module does nothing to require any change in the management of signals between processing modules.

We have seen that adding a storage module to the resources being used is relatively simple and consists primarily of verifying the operator's request and then adding the logical storage resources to the available list.

3.4 Removing a Processing Module

In the computer utility environment, each unit of processing resource (which we refer to as a "processor") is at any point in time allocated to execute in behalf of some process. To dynamically remove a processing module, the reconfiguration process must force the processor(s) in the module to stop executing for any useful process of the system. We now interpret the general reconfiguration design, outlined in chapter two and illustrated in Figure 3.1, in terms of logical processing resources.

The motivation and techniques for processor allocator designs applicable to the computer utility are not detailed here, since a lucid discussion of these is available [20] to the interested reader. Processor reconfiguration is basically independent of the allocator design details; however, the allocator must have a primitive to force rescheduling for any specific processor, i.e., must be able to preempt any processor.

The reconfiguration operations are modeled (see Figure 3.1) as transitions between four logical resource usage states. To understand these states we must provide an interpretation for the conditions bound/free and available/unavailable in terms of logical processors. A logical processor is "bound" when it is executing for some useful (user) process in the system -- that process is said to

be running. A processor is "free" when not executing a user process. So that this idling condition is not a special case to the allocator, we introduce the notion of an "idle process" (one for each processor) to insure that the allocator always has some process to run on each processor. When running an idle process, the processor can be thought of as executing in a "loop" or possibly just executing a "halt until signaled" instruction.

The allocator maintains in a system wide data base a list of all the running processes. This running list identifies which process is running on each logical processor. When the allocator wishes to run a process not now running, it searches the running list for a processor to which the process can be assigned, for example the processor currently running the lowest priority process. Thus the "available" processors are just those which occur in the running list.

To remove a processor, the reconfiguration procedures must ultimately leave a processor in the "free and unavailable" state, which means the processor is assigned to its idle process and not in the running list. Removing a logical processor from use is followed by removing a physical processor from the configuration. Since at some point the processor must be shutdown, we introduce the further constraint that a logical processor in the "free and unavailable" state is halted, viz., will make no references to any storage and will neither send signals to nor receive

signals from any processor.

For any acceptable request to remove a processing module (viz., when a viable set of modules will be left), the system will remove the logical processor(s). From the general model of chapter two we know we must again (as for removing storage) invoke two primitives in sequence to first make the processor unavailable and then to make it free. The first primitive must remove the processor from the running list and add it to a "removing list" -- the entry for this processor is updated to record that the processor is in an unavailable state. Another instance of a generic closed subroutine is identified to provide this primitive:

```
call Make_unavailable (processor_no);
```

where "processor_no" is the name of the logical processor to be removed from the running list.

After the call to Make_unavailable for all resources in the module, the Unbind primitive must be invoked to insure that the processor is not being applied to any useful processing. In a direct analogy to storage resources, there are three ways that the unavailable processor may be assigned to its idle process:

1. The processor was running the idle process when Make_unavailable was called.

2. A running user process invokes a normal allocator function in order to release the processor -- for example, in Multics terms [20], the process running on this processor

called "block".

3. The reconfiguration process invokes a reconfiguration primitive to force a user process to stop running on this processor.

To insure that the processor is free (by one of these means), the system invokes the Unbind primitive for each logical processor being removed. Previous research [20] has provided an allocator design for managing logical processors as members of an anonymous pool so that any running process can be easily "preempted". Typically a central processing unit would be preempted by an interrupt, but a high speed channel would merely wait for completion of the current operation. The allocator's preempt primitive causes the running process (which may be the process performing reconfiguration) to be replaced with a process selected by the allocator -- for an unavailable processor only the idle process is selected to run. Thus the Unbind primitive forces an unavailable processor to run its idle process by invoking the allocator's normal preempt primitive.

Although we have seen how to free the processor from explicitly assigned processing, there remains the problem of processing implied by signals (e.g., interrupts) sent to the processor. There are two types of signals -- process signals intended for a specific process and system signals directed to the processor regardless of what process is running. Since the idle process is expendable we can completely control its

response to process signals without disrupting the operation of the system (viz., no other process expects to communicate with the idle process); therefore, we design the idle process so that it does not receive process signals (except for a signal to halt, as discussed below, from the recofiguration process) when its processor is unavailable.

System signals -- for example channel interrupts -- must be processed by some processor as determined by system convention, but since they are used to provide system-wide functions, any processor will do. We invent the notion of a signal target table that specifies the system convention for which processor(s) receives each type of system signal. To make the processor unavailable for processing future system signals, the system replaces each occurrence of the processor as a target in the signal target table -- a replacement processor can be selected from the available list. Any outstanding system signals are allowed to run out (i.e., the processing is completed), and then the processor is free from system signal processing.

We have identified a design to insure that the processor is allocated to only the idle process and is not performing implicit (intermodule signal) processing. If the idle process is implemented as a loop (as opposed to being halted), recall that by our definition the processor is not "free and unavailable" until it is halted. Therefore, if necessary the reconfiguration process sends a process signal to the idle

process: in response to this signal the idle process executes a "halt" as its last instruction. Now we know the functions required of the Unbind primitive -- preempt the process running on the processor, redirect all intermodule signals to other processors, and signal the idle process running on the processor to halt. We again provide these functions with an instance of the generic subroutine:

```
call Unbind (processor_no);
```

where "processor_no" is the name of the (unavailable) logical processor that is to be made free.

Finally, after invoking Make_unavailable and then Unbind for the processor(s) in the processing module being removed, the system records the module as unavailable in the module configuration table. Recall from chapter two that a processor only has memory of information related to the specific process it is executing -- in this case an idle process whose existence is basically immaterial to the operation of the system. Thus the (halted) processor contains no information needed by the system, and the completely expendable idle process can have all its resource demands (viz., virtual resources) removed. For example, the system can delete all the storage required for per process data and all allocator entries for this process. The system can also release any per processor resources, for example, the wired storage for saving the processor state at interrupt time. In addition, if storage modules are used as a relay for signals, the system

can remove all occurrences of the processor in the signal address table as superfluous.

We have developed a design to remove a processing module from use by the system. This design is predicated on the existence of an allocator primitive to preempt the process running on any processor. The basic problem is insuring that the processor is not in the future required by the system for explicit (i.e., via the allocator) or implicit (e.g., system interrupt) processing. We can see that the reconfiguration capability of a system can be restricted by designs that require a specific processor for any process (for example, as a "master" processor for controlling "slave" processors), or for intermodule signals (for example, to process I/O interrupts).

3.5 Adding a Processing Module

When presented with an acceptable request to add a physical processing module (viz., a request for a module that is part of the configuration but not now in use), the system makes the processor(s) in this module available to the allocator. For each unit of physical resource the name of the logical resource is determined from the module configuration table. In terms of our model (see Figure 3.1), the system invokes the Make_available primitive for each logical processor in order to perform the operations represented by

the transition from the "free and unavailable" to the "free and available" logical resource usage state.

The functions necessary to make a processor "free and available" are essentially creating its idle process, creating per processor data, forcing the processor to start executing the idle process, and finally adding the processor to the running list. To provide these functions we introduce the reconfiguration primitive as the (generic) closed subroutine:

```
call Make_available (processor_no);
```

where "processor_no" is the name of the logical processor being added. When a call to Make_available returns the processor is available to the allocator for use in meeting the explicit processing demands of any process, and is also available for implicit (e.g., interrupt) processing.

Make_available must provide an idle process for the added processor. Recall from chapter two that the set of virtual processors changes dynamically, viz., user processes are created and destroyed. Therefore, the allocator, for reasons other than reconfiguration, must have a primitive for creating a process. For reconfiguration we augment the process creation function so that an idle process can be created. The allocator primitive to create (idle) processes must provide an appropriate entry in the alocator's system-wide data base -- in Multics terms [20], must provide a filled in process table entry -- and must create any required per process data area (for example a call stack).

In order to control the first actions of a newly added processor, we want it to first execute a known, well-behaved process -- for convenience we use the new idle process. Now in general the allocator can choose any available processor to execute a process; however, we require the process creation primitive to create an idle process such that only one specific logical processor can be assigned to it. Thus even though the idle process exists it will not be selected to run until its processor has an entry in the running list. To complete the idle process the reconfiguration primitive must provide the instructions that comprise its "program". The reconfiguration primitive will also create any per processor data for the processor being added.

Now the crux of the problem is how to cause the idle process to actually start running. Recall that by our definition of the "free and unavailable" state, the processor is initially halted. Clearly a signal to the processor is needed to cause the processor to do anything other than stay halted. If storage modules are used to relay signals, then entries are added to the signal address table so that signals can be sent to the processor. If (as is common for channels) the program for the idle process is just a (wired-in) halt, we can now say that the idle process is actually running. On the other hand, for an idle process that has a "loop" program stored in memory, we must cause the processor to begin executing this program. A signal must be sent to the

processor, but where a processor will begin executing in response to a signal depends on the physical hardware configuration -- the issues involved in having a processor respond in a controlled manner will be examined in detail in the next chapter. For the moment we assume there exists some signal to start the processor executing the "program" of the idle process. Once the processor starts executing the idle process, it is added to the running list, making it available to the allocator.

We have examined a design for adding a processor. The operations required are more complex than those for the analogous adding of storage due to intermodule signals and the more involved interpretation of the "free" state for a logical processor. After making the processor(s) in the module available to the allocator, the system updates the module configuration table to reflect that the module is now available for use by the system.

3.6 Summary of Changing Module Utilization

In this chapter we have developed in detail specific designs for adding and removing storage and processing modules from the set of modules the system is actually using. These designs have demonstrated that the operations of reconfiguration can be conveniently modeled by resource usage state transitions as postulated in chapter two. The crucial

elements of each design have been presented as a combination of closed subroutines and system-wide data bases. The subroutines directly represent the state transitions of the general model (Figure 3.1). The primary system-wide data base introduced for reconfiguration is the module configuration table which reflects the hardware configuration and the utilization of each module. In addition we identified data bases of the system resource allocators which are significant to reconfiguration -- these include the "available list" for storage, the "signal address table" (when storage modules act as relays), the "running list" for processors, and the "signal target table".

Although we have constructed solutions to the problems of changing the modules being used by the system, these program oriented solutions are really useful only when coupled with some method of actually changing the hardware configuration of physical modules that comprise the system. In the next chapter we extend our investigation to consider in detail the hardware oriented problems of changing the configuration of modules in the system.

CHAPTER FOUR

CHANGING THE HARDWARE CONFIGURATION

4.0 Dynamically Changing Physical Module Configuration

The hardware modules in a computer installation can be used for any one of several applications, such as providing services to users or running diagnostic programs to aid the repair of a faulty module. To provide a high availability for a computer utility, it must be possible to partition the installation's modules into independent operating units (which we term "partition-systems") that can be simultaneously used for different applications.

Since there can be more than one totally independent partition-system in a single installation, an operator must intervene to specify which of the installation's modules are available for use by each partition-system. This chapter develops a hardware architecture that decouples this simple manual selection operation, from the complex operations that control the configuration of the modules available to each partition-system. This architecture allows each partition-system to automatically control the configuration of its own modules, for example, control the addresses assigned to its memory modules. When combined with the operating system primitives of chapter three, this architecture provides a complete dynamic reconfiguration capability.

Recall that in chapter two we modeled the configuration of a partition-system in terms of a binding between physical resources and logical resources. This chapter considers in detail the operations required to change the configuration. These operations are modeled in terms of changes in the usage state of the physical resources, as illustrated in Figure 4.1 -- this model is basically a specific instance of the general model of state transitions presented in chapter two (see Figure 2.3). The relationship between the operations on physical resources considered in this chapter and the operations on logical resources considered in the last chapter was previously discussed in chapter two and illustrated in Figure 2.4.

Now we need to interpret this model in terms of hardware modules. Basically the available/unavailable states are a model of whether or not an operator has specified the hardware module as part of the partition-system. The bound/free states model whether or not the partition-system has electrically connected the (available) module as an operable part of the configuration. From this model we see that there are two basic design problems we must solve in this chapter:

1. The operator needs a mechanism for specifying which modules of the installation are to be in which partition-system. We will provide a convenient operator interface at a single location for all modules in the installation. As the first step in adding any module, the

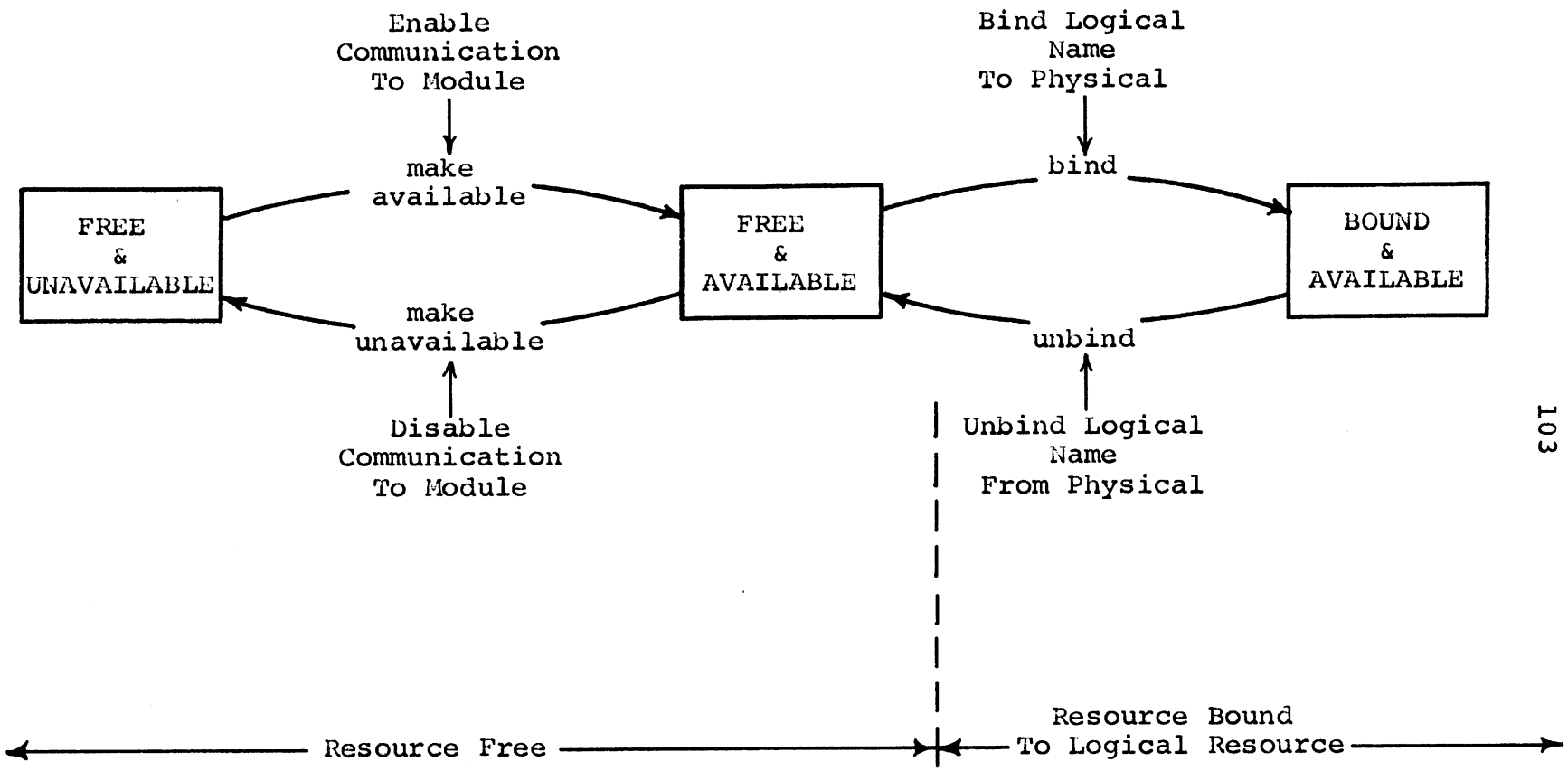


Figure 4.1 -- Usage State Transitions of Physical Resources

operator makes the module "available" to the partition-system. As the last step in removing a module, the operator makes the module "unavailable" to the partition-system.

2. Each module needs mechanisms so that the partition-system can control the configuration (i.e., binding and unbinding) of the modules made available to it by the operator. Basically these mechanisms control the electrical interconnection of the modules available to the partition-system. We conceive the functions for accessing a module's configuration control mechanisms as generic "instructions" executable by some type of processing module:

Set_config module data

Read_config module data

where "module" specifies the particular physical module and "data" is the configuration control information. (Read_config is needed primarily for partition-system initialization as discussed later, rather than for reconfiguration as such.) Later sections of this chapter examine in detail the specific configuration control mechanisms needed in the hardware modules.

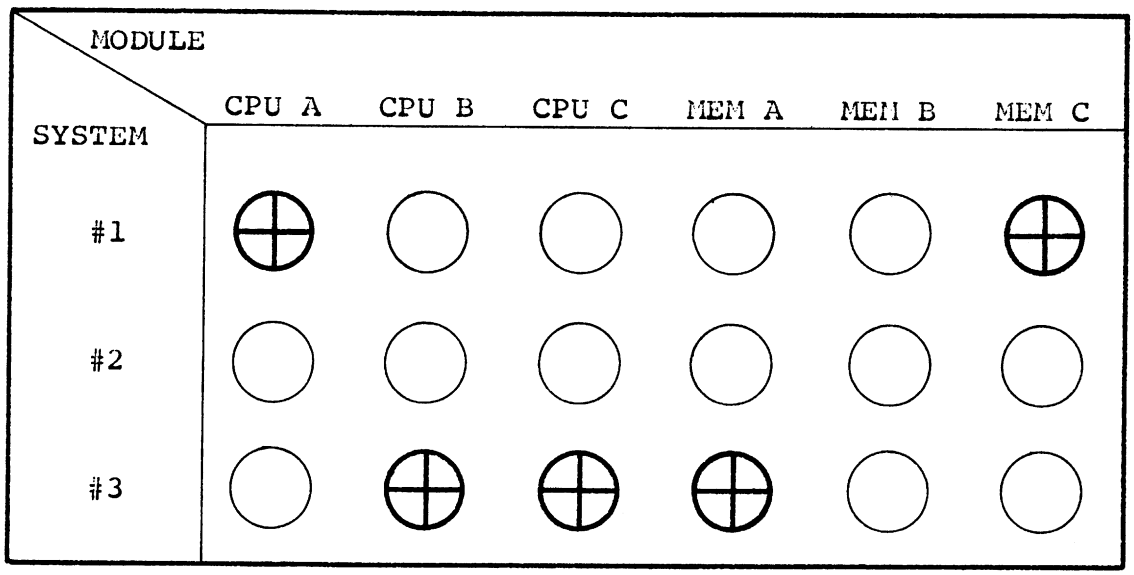
4.1 Changing Module Availability

Now recall that an operator selects which independent partition-system each module is to be in: this means specifically that he specifies which modules are available for

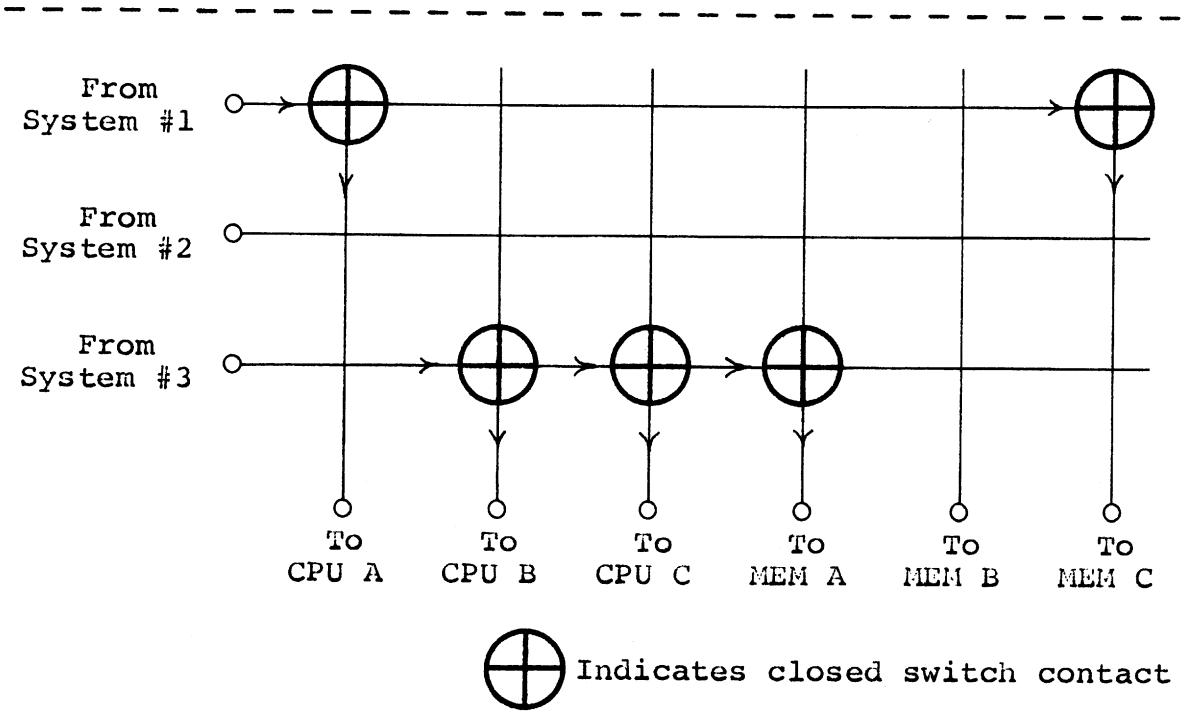
access with the Set_config instruction (for convenience in monitoring operator actions, we allow the "harmless" Read_config instruction to access all modules). We introduce an installation-wide operator interface matrix as the mechanism for operator selection. Externally this matrix is a set of operator activated "switch buttons" that selects a specific partition-system for each module in the installation, as illustrated in Figure 4.2 (there are, of course, interlocks to prevent making a module available to more than one partition-system). Internally this matrix is a switching network that controls the flow of Set_config signals from a partition-system to any module, as also illustrated in Figure 4.2.

The operator interface matrix must also insure that the Set_config signals from a processing module can only go to the partition-system which contains that module. Each partition-system must explicitly control which of its processing module(s) can actually send the Set_config signals -- in particular, a processing module just made available to a partition-system must not change the configuration of modules already in use, until the partition-system makes sure that the new module is "well-behaved".

Therefore, for each possible partition-system in the installation, the operator interface matrix has a program accessible send register (viz., an electronic "gate") for each processing module that can execute Set_config; a processor



Typical External Layout



Simplified Schematic Diagram

Figure 4.2 -- Operator Interface Matrix

can send Set_config signals for a partitular partition-system only if this send register is ON. The schematic diagram of Figure 4.2 is redrawn in Figure 4.3 to reflect this design. This design implies that instances of the generic Set_config and Read_config instructions reference the send register, possibly as part of the configuration control mechanism of the processing module -- for convenience in monitoring operator actions, we also allow Read_config to read all the operator activated switches on the operator interface matrix. The operator interface matrix enforces the following constraints on the send register:

1. A module's send register can be ON for a partition-system only if the module is available to that partition-system. In other words, one partition-system's processing modules can never send Set_config signals to another partition-system.

2. Whenever an operator selects a processing module for a different partition-system, the matrix initializes that module's send registers as OFF for all partition-systems.

In conclusion we observe that the number of independent partition-systems provided is determined by the operational goals of the installation, but each useful partition-system must include at least one processing module capable of executing the Set_config instruction. We defer to a later section the problem of intializing the operator interface matrix when starting the partition-system on a "bare machine".

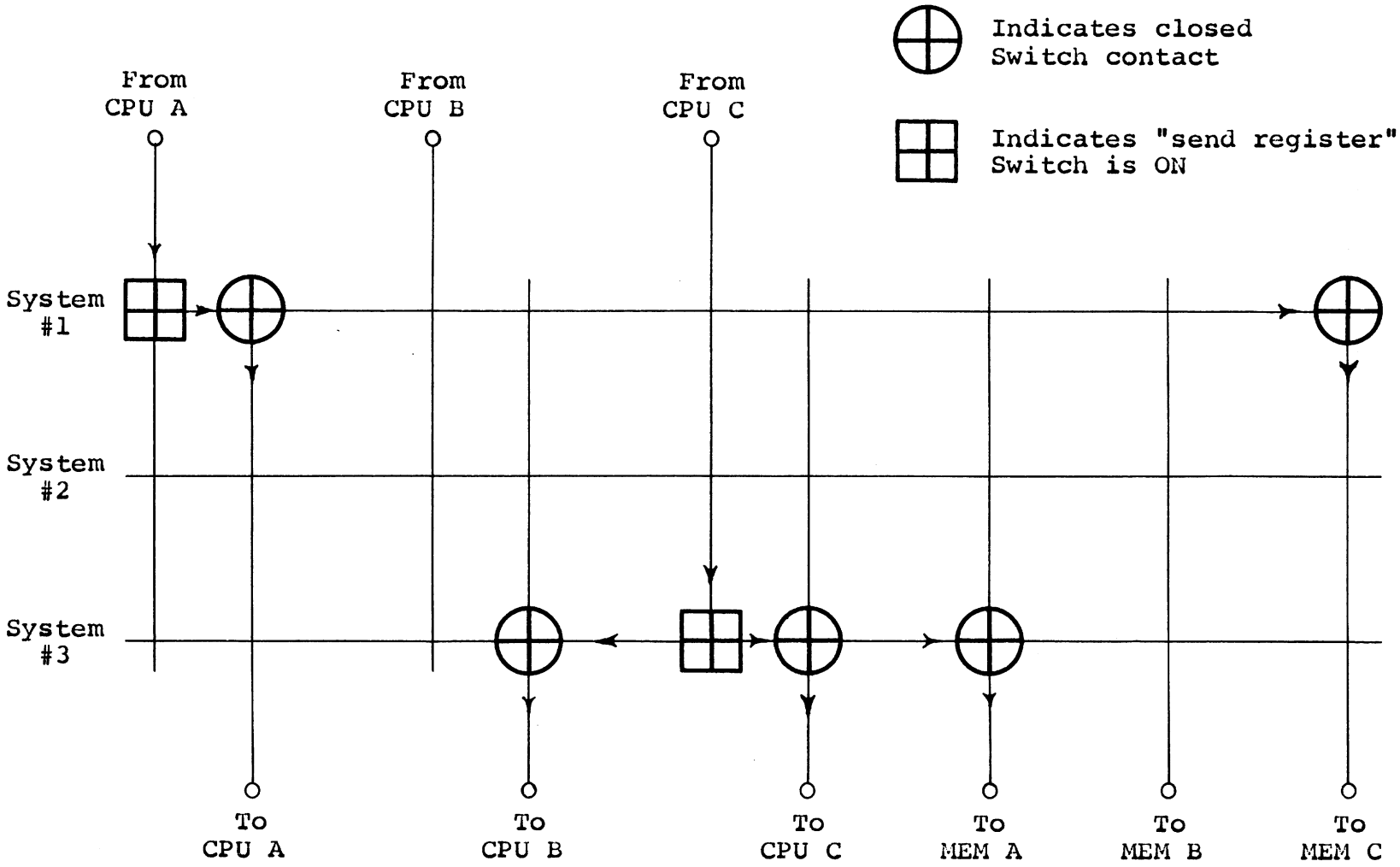


Figure 4.3 -- Operator Interface Matrix Schematic

In terms of our model (Figure 4.1), the operator interface matrix essentially defines which modules are "available" for a partition-system. We next examine in detail the configuration control mechanisms needed in each module to implement the "bind" and "unbind" usage state transitions of our model.

4.2 Changing Module Binding

Recall that chapter two characterized a modular computer installation as a collection of processing and storage modules with an interconnection network consisting of interface ports and memoryless links (see Figure 2.5). Although the network itself is fixed, the effective electrical connections are determined by configuration control mechanisms in each module. Binding and unbinding of physical is a model for the changes in these electrical connections. First we will examine the mechanisms needed for all modules, and then we will examine the particular requirements for storage and processing modules.

We digress a bit to note that a crucial feature of the modular structure is that the interconnection network is never dynamically changed. Since the topology of this network must be known for reconfiguration, we augment the module configuration table to reflect the "wiring diagram" of exactly how each link in the entire installation is connected to each port: this information is provided at partition-system

initialization time.

Now for a partition-system to use (viz., have bound) a physical module, it must have an effective electrical connection with all other modules in the configuration -- we say that its links to other modules are enabled. Our model of an installation as consisting of independent partition-systems immediately points to a problem in enabling links: although a module belongs to a single partition-system, a link (viz., between modules in different partition-systems) does not. In particular, we cannot simply use a single "on/off switch" in each link to enable/disable the link, because to add and remove a module, each partition-system must be able to manipulate the "switch" in each of the links to its modules. However, this ability to manipulate all its module's "switches" means that one partition-system can enable a link from one of its modules to a module being used by another partition-system, and thus cause damage -- for example, write in the memory of the other partition-system. Clearly each partition-system needs a way to guarantee its isolation from any other partition-system, regardless of the mistakes (e.g., errors by a technician repairing an off-line module or program bugs in the supervisor, viz., those unrelated to reconfiguration itself) that occur.

The required protection is provided by an "on/off switch" in the port of each module (instead of in the link). An "on/off switch" is of course just a form of binary memory, so

we will model this capability with a port enable register for each interfacing module. The port enable registers in each module are accessible by specific instances of the generic Set_config and Read_config instructions. A link is enabled if and only if the port enable registers at both ends of the link are enabled (viz., ON). Figure 4.4 illustrates this design. Now a partition-system can completely protect itself by disabling its own modules' port enable registers (and thus disable the links) for all modules not part of the configuration.

To remove a module from the configuration, the partition-systems uses the Set_config instruction to disable, in the remaining modules, the port enable register for the module being removed. Figure 4.5 illustrates the Set_config signals used to disable links to a module.

We digress a bit to recall from chapter three that when a processor is added to the configuration it is defined to be initialized in a "halt until signaled" state, since it is clearly dangerous to add a processor that is running in some unknown way. Therefore, for the processor(s) in a processing module we introduce a binary initialize register that is ON only if the processor is in a "halt until signaled" state. Not only does the initialize register reflect the current state, but also a partition-system can force a processor to halt (viz., become "initialized") by using Set_config to turn the initialize register ON. Once the processor is

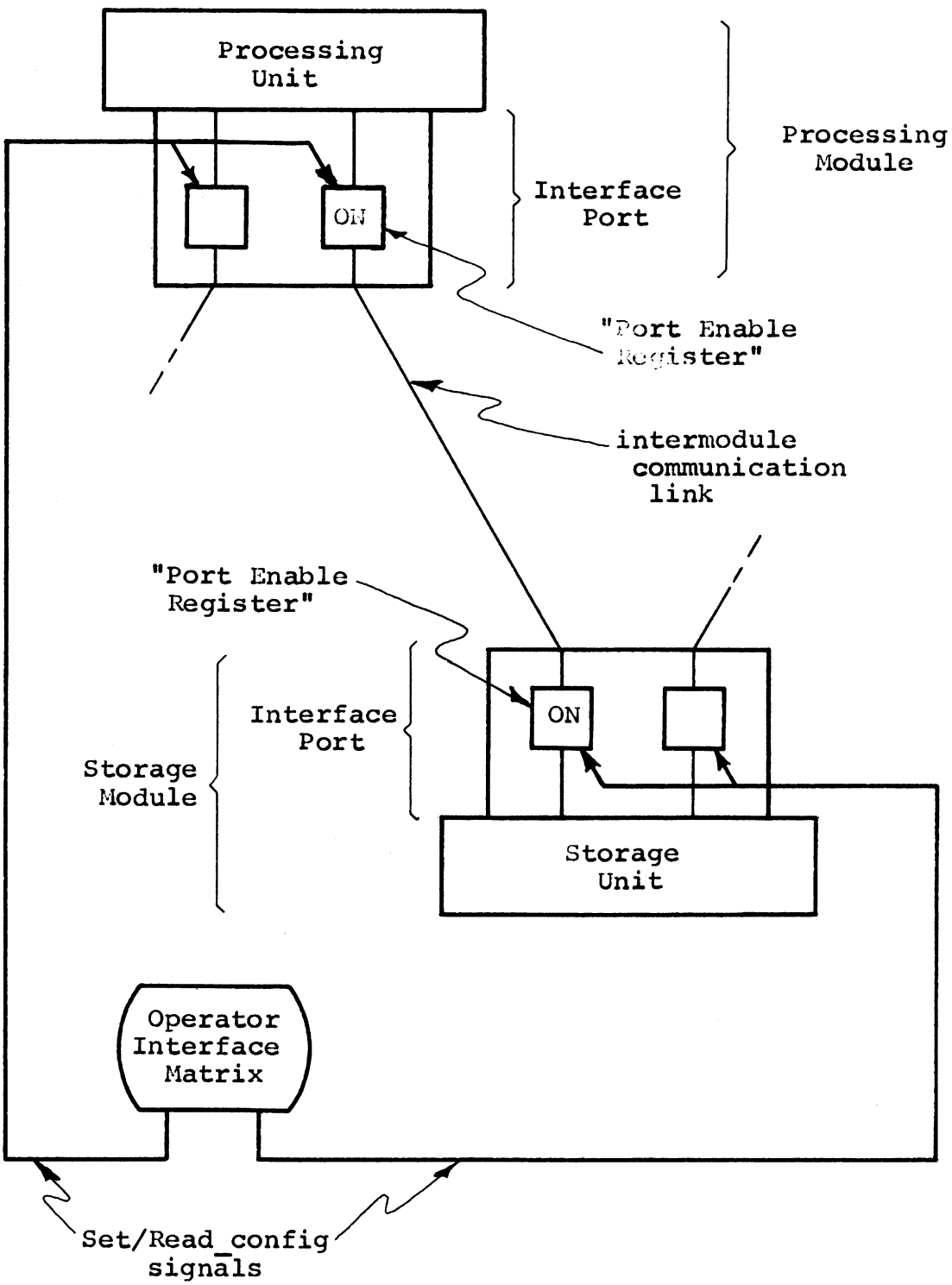


Figure 4.4 -- Typical Intermodule Communication

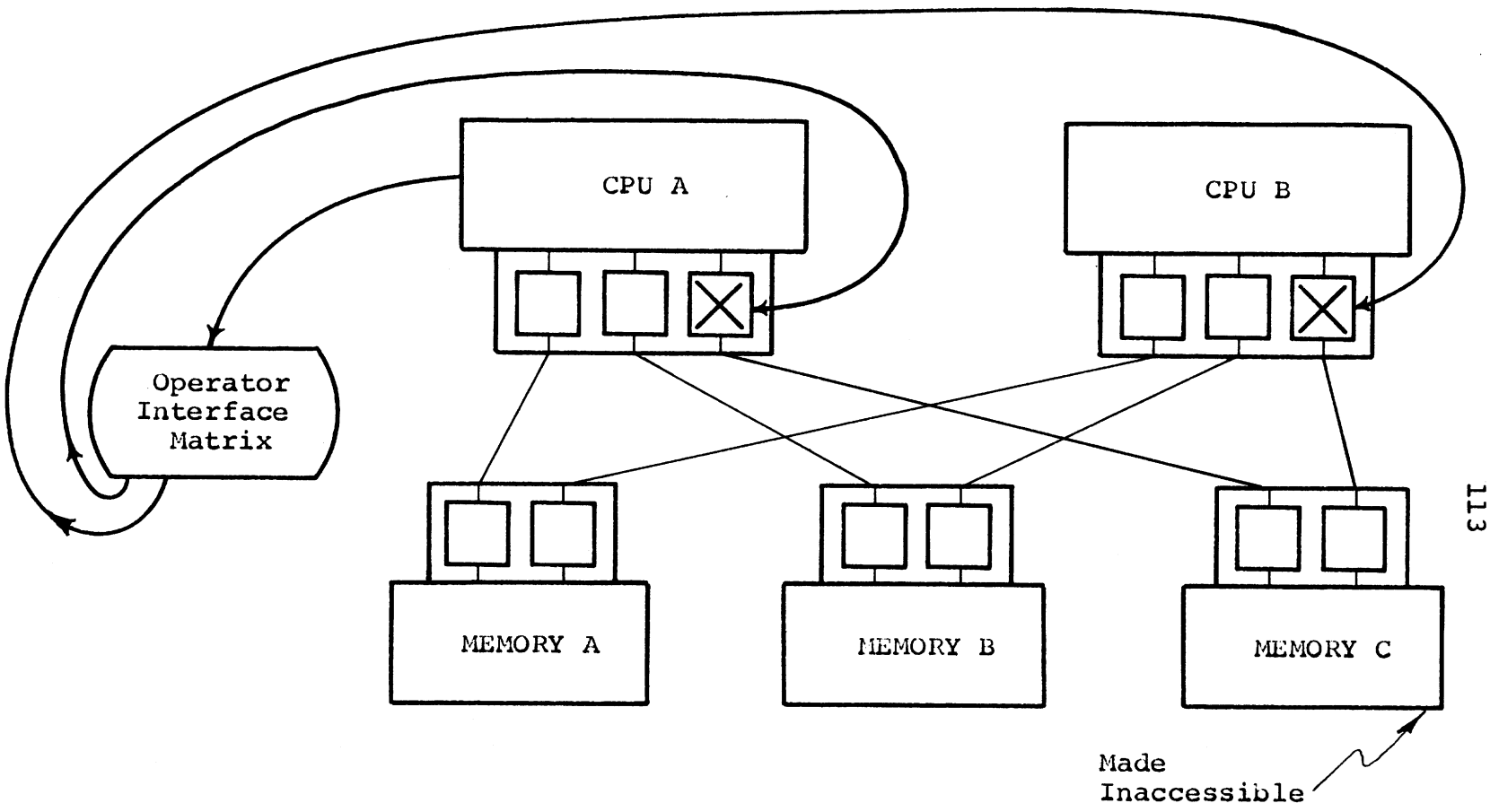


Figure 4.5 -- Disabling Intermodule Communication

initialized, the send register, if any, for the processor (in the operator interface matrix) can safely be set ON: this allows the partition-system to use this processor to execute Set_config. Similarly, each storage module used as a relay for signals between processing modules includes an initialize register; when Set_config turns it ON, the storage module clears any outstanding signals.

Now to add any (available) processing or storage module to the configuration, the partition-system first uses the Set_config instruction to enable port enable registers in the new module for just those links to modules already in the configuration. The partition-system next uses Set_config to set the initialize register ON (and to set ON the send register, if any). Then the partition-system executes Set_config for the modules already in the configuration to enable the port enable registers for the links to the new module. Figure 4.6 illustrates the Set_config signals to port enable registers needed to enable links to a module.

In summary, port enable registers in the port of each module primarily determine the effective interconnection of modules. These registers control all communication over the intermodule links (Figure 4.4). The only intermodule signals not controlled by port enable registers are the Set_config and Read_config signals; these signals are controlled by the operator interface matrix on an installation-wide basis, rather than on an individual partition-system basis. We now

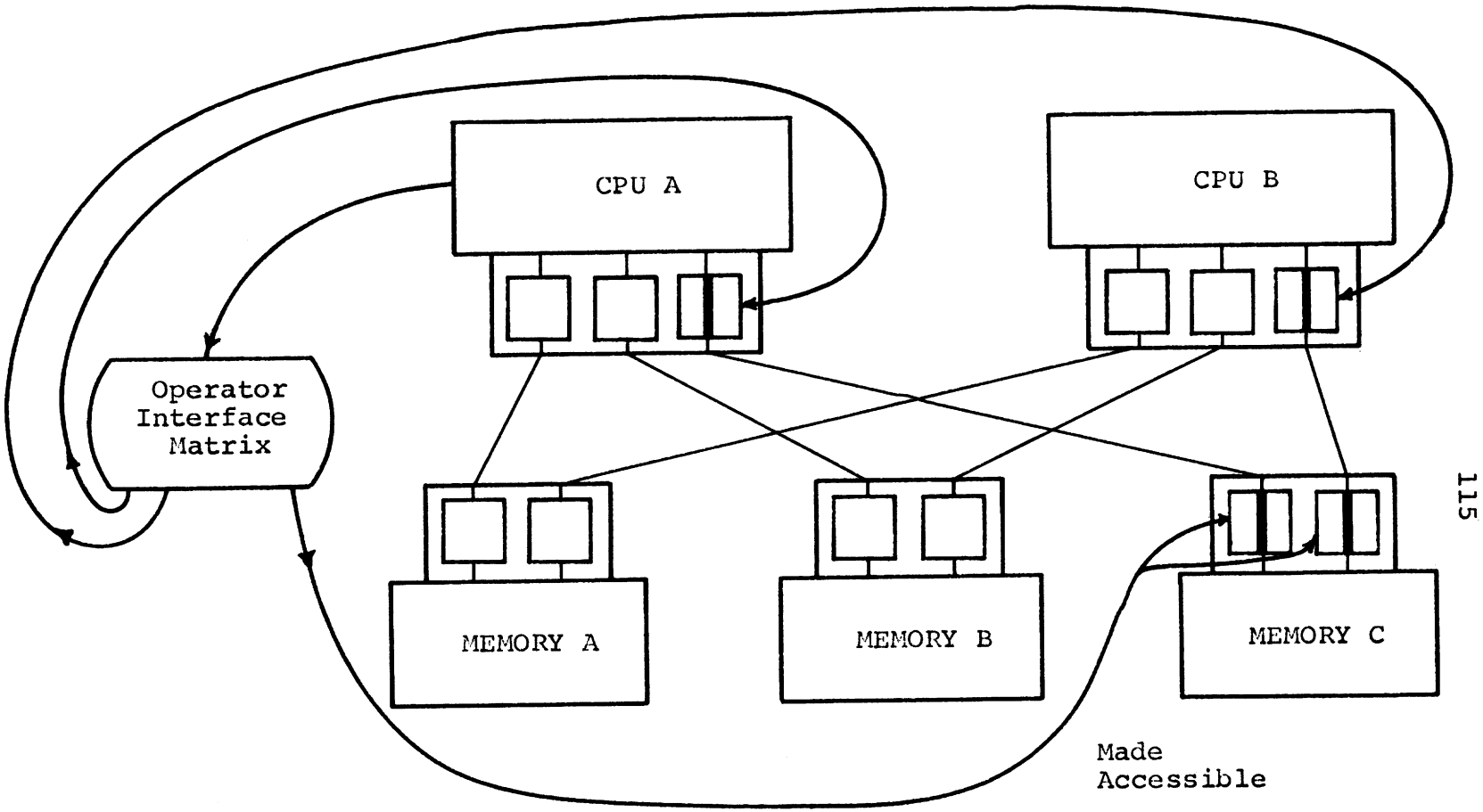


Figure 4.6 -- Enabling Intermodule Communication

examine in detail the additional operations required to bind/unbind a storage or processing module.

4.2.1 Storage Module Configuration

A physical storage module can only be used by the partition-system if it is "bound" to logical resources -- viz., a range of absolute addresses. We now need a mechanism for controlling this binding. Recall that the modular structure developed in chapter two requires "port selection logic" in each processing module to map an absolute address into a reference through a link to the proper storage module and a particular location within that module. To specify the absolute address range assigned to each storage module, we introduce (within the port of each processing module) an address interval register for every link. The address interval registers are accessible with specific instances of the generic Set_config and Read_config instructions. Figure 4.7 illustrates our model of a processing module port. The address interval register is typically implemented with only a small amount of memory in the port -- for example a "base address" and the "size" for the storage module.

The partition-system makes a physical storage module (being added to the configuration) "bound" by assigning to it logical resources (viz., an absolute address interval) not already assigned to another storage module. As noted in

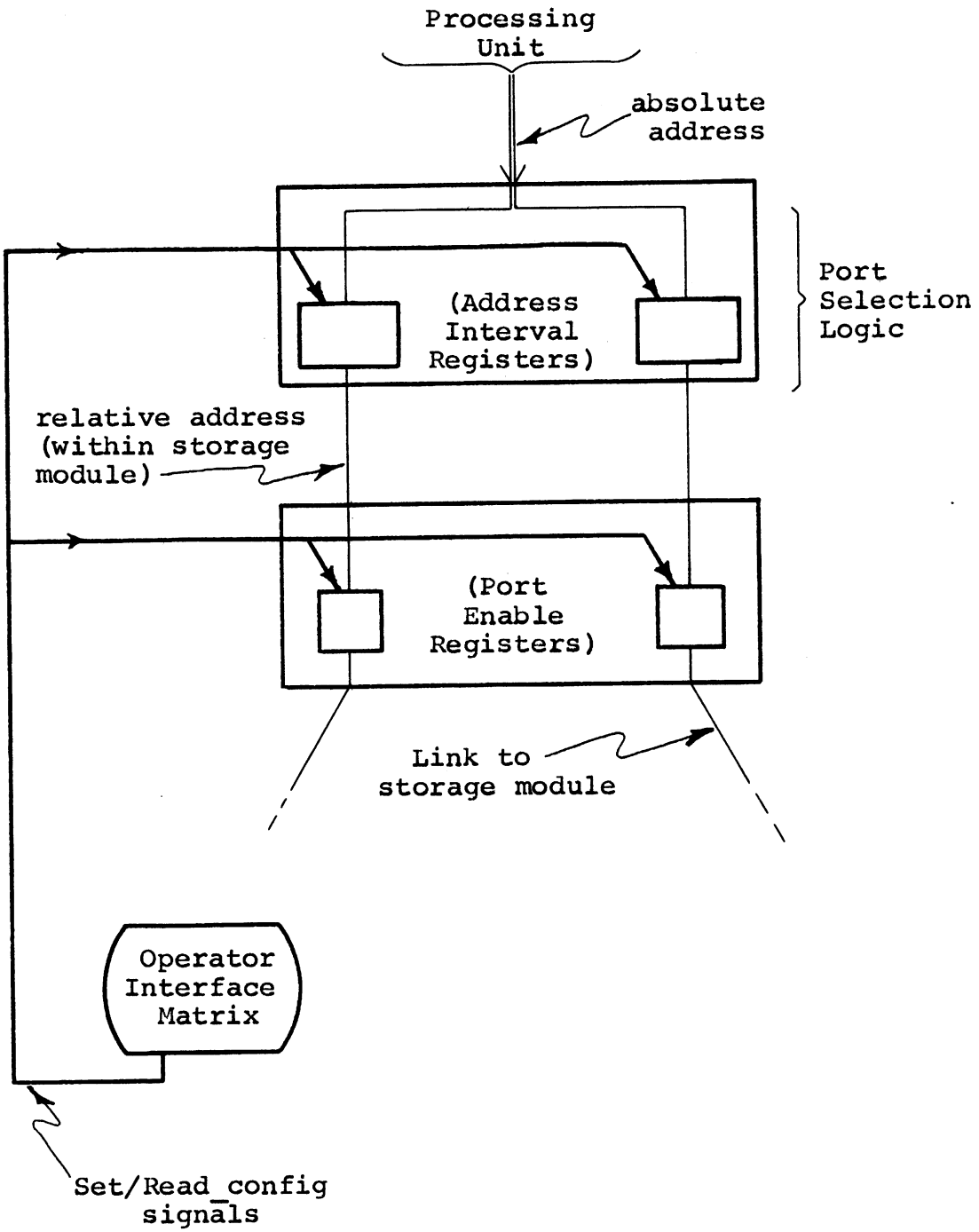


Figure 4.7 -- Model of Processing Module Port

chapter three, the module configuration table records the addresses already assigned to each storage module. The partition-system executes Set_config, for each interfacing processing module in the configuration, to set the address interval register for the processing module's link to the new storage module. The module configuration table is, of course, updated to reflect this binding.

To "unbind" a storage module (being removed), the partition-system updates the module configuration table to show that the module's address interval is unassigned. The value left in the address interval registers for links to the module being removed is basically unimportant, since the module can only be made unavailable or bound to a new address interval (see Figure 4.1).

In summary, the configuration of storage modules in a partition-system is determined by address interval registers in the interface ports of the partition-system's processing modules. The module configuration table is used to manage the binding of physical to logical resources, e.g., to prevent the partition-system from using two storage modules with the same absolute address range.

4.2.2 Processing Module Configuration

To be used by the partition-system, a physical processor must be bound to a logical processor. In addition, the

"instruction counter" for implicit processing (e.g., responding to interrupt signals) must be bound to an execution point. We now develop a design for these binding operations.

The logical processor number (i.e., the name of a unit of logical processing resource) may not seem as intuitively motivated as an absolute address (i.e., the name of a unit of logical storage resource); however, a unique processor number for identification is needed by the processing resource allocator. This binding of physical to logical processors is recorded in the module configuration table. We will view the processor number register as contained in the processing module; this register is accessible thru Set_config and Read_config. As contrasted to the contiguous absolute addresses in a storage module, the processor numbers for processors in the same module may bear no relationship to each other, except for uniqueness; if so, the module must have an explicit processor number register for each processor in the module. For example, channels often have individually assigned "channel numbers".

Although the processor number is used for managing the explicit processing, recall from the last chapter that a processor also performs implicit processing in response to signals. Now we digress a bit to observe that the intrinsic problem for a processor responding to any signal (e.g., an interrupt) is determining the single location where it can find its next instruction to execute. We will view this

location as being specified by a signal address register, which is accessible thru Set_config and Read_config. Figure 4.7 is now augmented in Figure 4.8 to illustrate our complete model of a processing module.

In a simple case (e.g., a highly specialized channel) this next instruction may implicitly be a wired-in program; however, usually an explicit signal address register specifies the location in primary memory of an instruction to be executed. The instructions at the specified location will typically save the state of the processor (so it can be restarted where interrupted) and transfer to a "handler" for the signal. The interpretation of this single absolute address will depend on the particular hardware design -- for example, in a processor with segmentation hardware the signal address register may in fact be the "descriptor base register" (refer to Figure 3.4), and the next instruction is found in a reserved segment. In any case, the signal address register must be included in the relocation mechanisms updated when removing a primary memory module, as already detailed in chapter three.

We now have all the mechanisms needed to control the binding of physical processing modules. The partition-system makes a processing module (being added to the configuration) "bound" by assigning a processor number and signal address to each processor contained in the new module. In addition, the partition-system sets the address interval registers in the

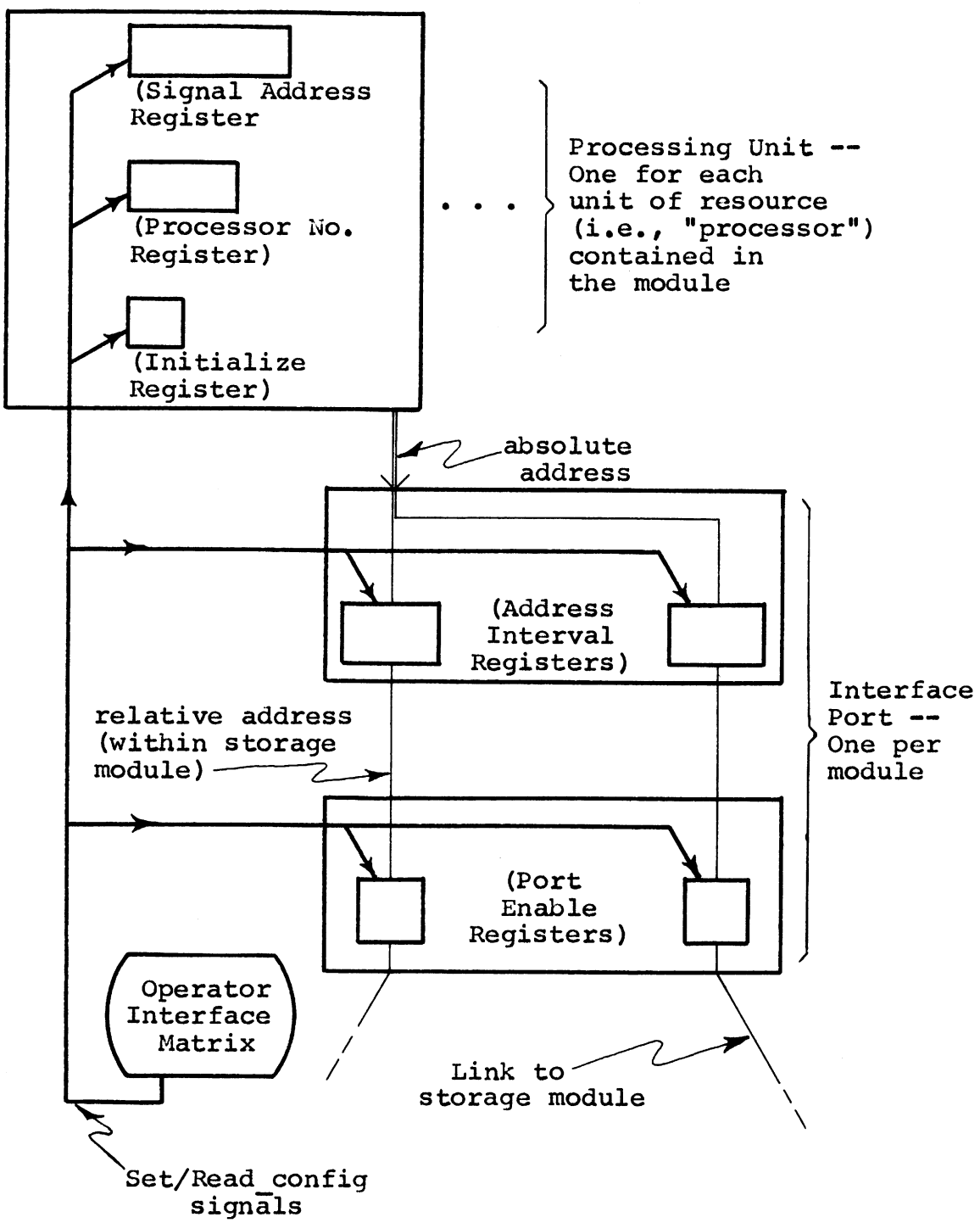


Figure 4.8 -- Model of Processing Module

new module's port to reflect the addresses assigned to the memory modules. With the internal registers set to desired values and the processor in a "halt until signaled state" the logical processor can be considered as "free and unavailable". The Make_available primitive of chapter three can then start the processor in a standard and controlled way by placing the desired instructions at the location specified by the signal address register and sending a signal.

To "unbind" a processing module (being removed) the partition-system updates the module configuration table to show that each physical processor is "free", i.e., that its processor number is unassigned. (By the conventions of chapter three, the processors are already halted.) The values in the various configuration control registers, of the module being removed, can be left unchanged.

In summary, the configuration for processing modules is primarily determined by a processor number and a signal address register within each processor. Although typical processing modules contain some form of the various configuration control registers (Figure 4.8), the crucial characteristic for dynamic reconfiguration is that the partition-system can insure suitable contents in these registers before a processor begins to execute as part of the configuration.

4.3 Partition-system Initialization

The design for reconfiguration must be compatible with the need to initiate partition-system operation on a bare machine. We now demonstrate how the above design can be used with a partition-system initialization strategy frequently found in modular hardware, such as that used by Multics. We view initialization in terms of a sequence of operating configurations (within the collection of available modules), each providing more capability than the preceding one:

1. We view the kernel of a partition-system as a self-sufficient "bootstrapper" (with wired-in instructions, data, and processing logic) contained in a single module. This bootstrapper first executes a wired in program that establishes a viable minimum configuration of modules. The bootstrapper then copies a "bootstrap program" from its wired-in data into a storage module.

2. This bootstrap program in turn executes on the minimum configuration and loads an operating system.

3. The newly loaded operating system expands the minimum configuration to include all the modules made available to the partition-system by the operator interface matrix.

4. Finally, the operating system begins its normal execution, using the full configuration of the partition-system.

To start the partition-system the operator activates the

wired-in "bootstrapper" with a manual switch; we augment the operator interface matrix to include a bootstrap switch for each partition-system. Typically the bootstrapper is contained in a normal processing module so it can use the same processing hardware; for convenience, we say that each processing module capable of executing the Set config instruction also contains a bootstrapper. When an operator activates a partition-system's bootstrap switch, the operator interface matrix makes a (basically arbitrary) selection of a bootstrap (processing) module from those modules available to that partition-system. The matrix first initializes the bootstrap module's send register ON and all other send registers for that partition-system OFF, so only the bootstrap module can change the configuration.

Then the operator interface matrix starts the bootstrapper. Using its wired-in program, the bootstrapper executes Read_config to determine the modules assigned to this partition-system, and from these modules selects a minimum configuration which includes the bootstrap module -- typically one central processing unit, one primary memory and one I/O channel are required. The modules in this minimum configuration are then "bound", using Set_config -- typically the logical resources used (for example, the absolute address interval) are determined by constants wired into the bootstrapper. This results in a useable, initialized (viz., non-running) configuration.

Next the bootstrapper loads (from its wired-in data) a small, fixed bootstrap program into primary memory at the location specified by a processor's signal address register. This transfer of data (viz., the bootstrap program) to memory is the first use of the normal intermodule links. The bootstrapper then signals a processor to start executing the bootstrap program: the bootstrapper then stops itself.

The bootstrap program in turn reads additional programs from a fixed I/O device (the particular device used is a "constant" in the bootstrap program), and then executes these programs. The programs read from the I/O device are an initialization portion of the operating system. The operating system adds the remaining modules assigned to this partition-system, initializes data bases (e.g., the module configuration table), and begins its normal operation.

4.4 Review

In this chapter we have presented a specific architecture that permits a partition-system to dynamically and automatically change its configuration of hardware modules, under the direction of an operator. We have used changes in binding as a model of the reconfiguration operations, and we have introduced hardware "registers" to provide the mechanisms for changing the configuration. Our strategy has been to identify as explicit, program accessible registers those

(often implicit or operator controlled) configuration dependent portions of contemporary modular architecture. We have also explicitly identified a simple operator interface to support the design.

The reconfiguration primitives of the last chapter taken together with the hardware architecture presented in this chapter provide a complete design for dynamic reconfiguration. This design assumes strictly serial, operator initiated requests to dynamically add and remove specific physical modules. To enforce this serial discipline we can view reconfiguration as being subject to a single software reconfiguration lock. This lock prevents conflicts over the use of reconfiguration data such as the module configuration table and prevents races when accessing the hardware registers.

CHAPTER FIVE

THE MULTICS IMPLEMENTATION

5.0 Background

The usefulness of the general design presented in the preceding chapters has been demonstrated by employing it to provide a practical dynamic reconfiguration capability for the primary time-sharing service at M.I.T. The Multiplexed Information and Computing Service (Multics) system has been developed to serve as a public utility, initially for the M.I.T. community. The general organization of Multics was described in 1965 and is available to the interested reader in the literature [1 ,6 ,7 ,8]. The basic Multics objectives are controlled sharing of information among users and highly available computational services to meet a wide spectrum of user needs. The M.I.T. implementation was developed as a research project, but has been generally available to users in the M.I.T. community since October, 1969. Since that time the system has attracted about 700 registered users and now (May, 1971) typically serves more than 50 users simultaneously when running the full equipment configuration.

The Multics system was implemented on the modular GE-645 [1] computer, and the principal modules in the M.I.T. installation are shown in Figure 5.1. Although Multics is available to a wide range of users, about one-third of the

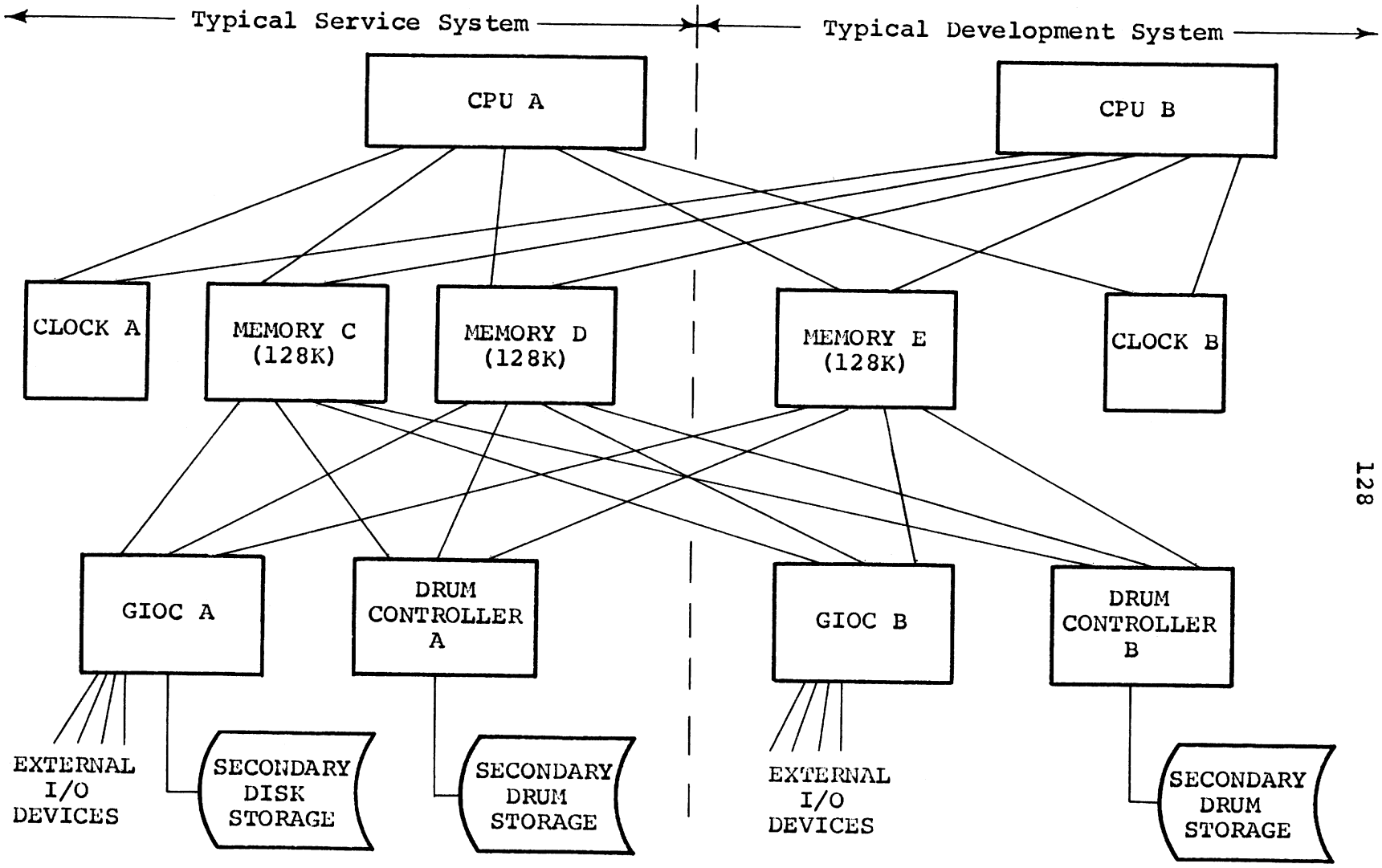


Figure 5.1 -- Multics Hardware Installation

system resources are currently consumed by the staff engaged in continuing Multics research. As a result of this continuing research, a major new version of the Multics supervisor is installed about twice a week. Since the initial implementation did not include dynamic reconfiguration, one central processing unit (CPU) and one memory module were dedicated to a "development system" partition for developing new versions of the supervisor and for maintenance. Each morning (about 5:00 a.m.) when there are few if any users, the "service system" partition was shutdown and reinitialized -- the physical memory and CPU modules were rotated between the service and development partitions to permit maintenance to be done on each module while it was part of the development system.

Within a few months after the initial offering of the Multics service, the demand during prime time exceeded the capacity of the one CPU and two memory service system, while the development system was often idle. Since in Multics all user files are maintained on-line in secondary storage the demands for secondary storage change slowly; therefore, a fixed partition of secondary storage between the service and development systems was entirely satisfactory. However, it was clear that the CPU and/or memory from the development system was needed to increase the capacity of the service system during part of the day, while on the other hand it was essential to have a development system during other portions

of the day. The change in configuration was needed at a time when there were many users logged in, and therefore it was totally unacceptable to shutdown the system and restart with a new configuration. Clearly a dynamic reconfiguration capability for CPU's and memories would be very useful.

As part of the research reported in this thesis the Multics supervisor was augmented to provide an early engineering design for dynamic reconfiguration. First reconfiguration was provided for only CPU's, but since a two CPU and two memory configuration turned out to be "memory limited", the CPU reconfiguration capability was not used on a regular basis. However, CPU reconfiguration was occasionally used when the service system CPU developed intermittent hardware failures. To avoid shutting down the system, the operator adds the development CPU and removes the faulty CPU for repair.

The memory reconfiguration capability was then added, although due to awkward but not intrinsic hardware limitations (as explained later) a choice was made to have one memory that cannot be removed. Once both CPU and memory reconfiguration were available, a daily schedule was established for adding the development CPU and memory to the service system during the time of peak demand (approximately 2:00 p.m. to 5:00 p.m. on weekdays), and whenever the development system was idle.

All reconfiguration is initiated by a computer operator who is logged into the system as a normal user. Only selected

users (viz., operators) are, of course, administratively granted access to the reconfiguration entry points. The operator types reconfiguration requests at an interactive terminal in the same way that any other request is made. In response he may receive instructions for operator actions (viz., manipulating switches), and ultimately he is advised that the requested reconfiguration is completed. Appendix I is a copy of the instructions currently provided the Multics computer operators for performing reconfiguration.

The Multics implementation is based on the general design of the previous chapters; however, since reconfiguration was added to an already operational system, the hardware design in general could not be changed, although there were a few minor corrections. On the other hand, the author had essentially complete design freedom to propose changes to the Multics operating system as needed for reconfiguration (within the limits of his ability to design, implement, and install these changes in a system used as a service facility). By examining the Multics implementation we hope to gain a clearer understanding of how the general design can be interpreted and applied to a specific system, even when some of the desired (hardware) features are not available. We will relate the elements of the Multics implementation to the general design primitives, data bases, and hardware registers introduced in chapters three and four. We will separately consider the design for each of the individual reconfiguration requests to

add or remove a CPU or memory module. First we will point out features of the GE-645 hardware used for Multics that are significant to reconfiguration:

1. The memory modules are used to relay signals between processing modules. However, a memory module can relay (interrupt) signals to only one CPU, instead of broadcasting to all CPU's, under control of the system, as in our general model of a modular system. The single CPU that receives interrupts is determined by a manual "control processor" switch on the memory module. This means that there must be at least one memory module for every CPU in the configuration.

2. Each processing module -- CPU, drum controller, and general I/O controller -- has a set of manual "base address" switches (instead of a program accessible "signal address register", as defined in chapter four), which specifies the absolute address of the instruction (or command) executed in response to an interrupt (or channel connect) signal. This serves as the "signal address register" we defined in chapter four.

3. The "port enable registers" in processing modules, which are used to control all intermodule communication with memory modules, are set by switches (instead of being program accessible, as proposed in chapter four) and cannot be changed while a processor is running.

5.1 Removing a CPU

In the idealized design we presented, any module can be removed by an operator issuing a single request to the system. To remove a CPU in Multics the operator must in addition change at least one manual switch and give at least two responses at his console. The Multics design for removing a processor is essentially the same as the general design developed in the preceding chapters, but contains the terminology of Multics; this design consists of the following specific steps:

1. The system locks the reconfiguration data base and checks the operator request for validity.

2. The system makes the CPU unavailable for allocation by forcing only the idle process to run on it.

3. For all memory modules relaying (interrupt) signals to the CPU, the operator changes the "control processor" switch so that interrupts are directed to some other CPU in the configuration.

4. The reconfiguration process signals the idle process for the CPU being removed that it will receive no more interrupts. The idle process returns a signal that it is about to halt, and then halts the CPU.

5. After receiving a signal of the imminent halt, the operator's process destroys the idle process and associated data bases for the removed CPU.

6. The system updates the reconfiguration data base, and makes all memory modules inaccessible to the removed CPU.

7. The system unlocks the reconfiguration data base and advises the operator that the CPU has been removed.

The Multics configuration is recorded in a system-wide data base that implements the "module configuration table". This data base is used to verify the validity of the operator request, e.g., that there will be at least one CPU left and that the CPU to be removed is actually part of the current configuration. This data base has an explicit lock that allows only one reconfiguration at a time.

The current implementation of the Multics traffic controller [20] uses scheduling priorities in conjunction with the "running list" to establish the "available" CPU's. When the traffic controller is entered a CPU is always passed (like a baton in a relay race) from the currently running process to the ready process with highest scheduling priority. A processor is available just as long as it can be passed to another (user) process. The Make_unavailable primitive is implemented by permanently giving the idle process for the specified processor the highest priority; then that processor will only be passed to the idle process, and thus is not available to be allocated to other user processes.

To stop any user process currently running on the processor, the Unbind primitive sends a normal traffic controller preempt interrupt to the processor. Since a

preempted processor is always assigned to the highest priority process, its idle process is run. The reconfiguration process executing Unbind is careful to unmask preempt interrupts, since the reconfiguration process may itself be running on the processor to be removed. Once the idle process begins to run on the processor being removed it will continue to run, since it has highest priority.

Next the implementation of the Unbind primitive frees the processor from interrupt processing. (Recall from chapter three that there are two types of interrupts -- "process interrupts" intended for a specific process, and "system interrupts" directed to the processor regardless of what process is running.) This implementation is made difficult by the hardware design which allows a memory module to relay interrupts to only the one processor specified by the manual "control processor" switch. This leads to the Multics implementation of the "signal address table" with the convention that each processor module has one memory module as its primary source of process interrupts. (In fact, with the exception of the real time clock interrupts directly from the clock module, all interrupts to a CPU come from a single memory module.) Furthermore, a portion of the "signal target table" is implicit in the hardware design since the system interrupts from the drum controller and general I/O controller are always directed to the memory module specified by their respective "base address" switches -- to simplify interrupt

masking it is required that such a memory module be the primary process interrupt source for some CPU.

For each memory module which can generate interrupts for the CPU to be removed, the manual control processor switch must be redirected to another CPU; however, the hardware design is such that to avoid undefined (and possibly disastrous) results, interrupts must be masked from leaving the memory module while the control processor switch is being moved. To prevent races and lost interrupts, Multics has the software convention that the primary interrupt source can be masked only by the CPU that it interrupts.

There are two cases; in the first case, to change the control processor switch on memory modules not used as the primary interrupt source, the reconfiguration process first masks interrupts, then the operator is instructed to move the switch, and finally the system unmasks interrupts after the operator responds to indicate that the control processor switch has been moved. In the second case, to change the control processor switch of the primary interrupt source, the reconfiguration process signals the idle process to do the masking, and the idle process returns a signal when interrupts are masked. (For convenience, all "signalling" to and from idle processes is done via changes in a system wide variable.) The operator then moves the switch, and responds to indicate that the switch has been moved. If this memory module generates only process interrupts (viz., cannot generate

system interrupts), it is then unmasked by the reconfiguration process. On the other hand, a memory module that generates system interrupts is treated as a special case, as described below.

An awkward problem arises in Multics, because system interrupts (which includes all I/O interrupts) are redirected to another CPU by the very slow operator manipulated control processor switch, rather than by a rapid change of the "signal target table" as in the general model. For a memory module which relays system interrupts, severe difficulties (for example, overflowing of I/O buffers) could result from delayed interrupts while the operator is moving the switch. As part of the design to cope with this problem, Multics ignores "extra" interrupts, and while waiting for the operator, a stream of artificial, CPU-generated interrupts is sent by the idle process to another processor so that events signaled by the true, masked interrupts will not go unnoticed. After the operator moves the control processor switch to a remaining CPU, the reconfiguration process forces its idle process to also run by giving it high priority and sending a preempt interrupt (again the reconfiguration process is unmasked, since it may be running on the preempted processor). The idle process of the remaining CPU changes the remaining CPU's primary interrupt source to be the memory that was associated with the CPU being removed -- this idle process updates the "signal address table", unmasks the module to allow system

interrupts, and signals the idle process for the processor being removed that interrupts are unmasked. Then this idle process for a remaining processor restores itself to normal priority. Note that forcing an idle process to change the interrupt source ties up a remaining processor for only a short time and therefore does not disrupt service to users. After receiving a signal that system interrupts are unmasked, the idle process for the processor being removed stops sending the artificial system interrupts.

After all the control processor switches are moved, the idle process for the processor being removed is signaled. This signifies that no more interrupts will be received by the processor, so the idle process returns a signal that it is about to halt, and halts. Note that although the idle process is given exclusive use of its processor for a long time, this processor is about to be removed anyway, so this causes no unexpected disruption. On the other hand the reconfiguration process is given no special scheduling consideration, and therefore in no way disrupts the other users of the system.

After the processor is halted, the idle process is destroyed, and the call stack and descriptor segment for the process are deleted. The processor's data base (used to save the processor state at interrupt time) is also deleted. The physical processor module is made "free" by updating the reconfiguration data base. The GE-645 memory modules include program controlled port enable registers, so the system

disables all its links to memory modules. Finally the reconfiguration data base is unlocked, and the operator is informed that the CPU has been removed. Before using this CPU he must manually disable its "port enable register" for the service system clock module(s); otherwise, the removed CPU can access and stop a service system clock, because links to clocks cannot be disabled at the port on the clock (since the clock modules have no port enable registers of any sort).

In summary we can say that the Multics implementation for removing a CPU is made quite cumbersome by a hardware design that uses a manual switch to determine the CPU receiving interrupts from a memory module, and by the clock port organization. The Multics reconfiguration implementation takes advantage of the Multics convention that an idle process can only run on a specific processor: in order to remove a CPU the idle process is augmented with procedures to perform tasks that must be executed on a specific processor.

5.2 Adding a CPU

With the idealized reconfiguration design a processor can be added by an operator making the module "available" (using the "operator interface matrix", which Multics does not have), and a single operator request to the system. With the Multics implementation, processor module initialization usually requires moving five "address interval" switches; the

the CPU (see Appendix I for an example).

With a manual switch the operator initializes the CPU in a halted state, since Multics does not have a program controlled "initialize register" (as defined in chapter four) that can be used to intialize a processor. Instead of a program controlled "signal address register", as defined in chapter four, each processing module (whether in the service or development system) has a "permanent" switch-assigned "base address" which the hardware design restricts to a limited range of absolute addresses. Since this fixed base address requires a specific absolute address range in every configuration, the "address interval registers" cannot have fixed values but are set (in manual switches) by the operator when initializing the processor. Since the hardware design does not allow these address interval switches to be changed while the CPU is running, address intervals must be assigned to, and ports enabled for, all memories in the installation. A "processor number" is by Multics convention permanently assigned to each physical CPU module. After completing all this manual initialization of the CPU, the operator responds to the system to indicate that the CPU is initialized.

The system then enables "port enable registers" for the processor in each primary memory module of the service system to enable all links to the physical processor. The memory module selected as the primary interrupt source is masked. Then the operator is instructed to direct the "control

processor" switch to the CPU being added, and he responds when done. The "signal address table" is updated to reflect that the physical processor is now accessible for processing interrupts. The reconfiguration data is updated to reflect that the physical processor is now accessible to the system.

The Multics implementation of the Make_available primitive creates a processor data segment for the processor, and creates an idle process with its call stack and descriptor segment. The idle process is created with the normal (lowest) priority but is in a "ready" state (rather than a "running" state) so there will be no attempt to preempt it. Next the processor is forced to start executing the idle process. Since Multics uses segmentation, the descriptor base register (refer to Figure 3.4 for illustration) must be loaded with the absolute address of the idle process descriptor segment. However, in response to any interrupt, Multics executes an interrupt vector in "absolute mode", at an absolute address determined from the processor "base address". The interrupt vector for the reconfiguration interrupt is constructed so it will save the processor state, load the descriptor base register, then leave absolute mode, and transfer to the procedure for the idle processes.

The reconfiguration interrupt is sent, and the newly executing idle procedure first checks for errors -- if the operator assigned the wrong processor number the processor halts, or if the operator directed the "control processor"

switch to the wrong processor the saved machine conditions are restored. If no error is detected, the idle process sets itself to the "running" state, unmasks the primary interrupt source, and sends itself a preempt interrupt so the traffic controller will give the processor to the highest priority process. Finally, the operator is informed that the processor has been added, and the reconfiguration data base is unlocked; if there is an error, part of the sequence for removing a processor is followed to reverse the processing already done.

The Multics implementation to add a CPU is also made cumbersome by the manual "control processor" switch. In addition, the lack of a permanent address intervals for each memory module makes it tedious for an operator to initialize the CPU, and there is no practical way to initialize the port enable registers to reflect the configuration. However, the more easily made operator errors can be detected -- the operator is told of his error and can repeat his attempt to add a CPU.

5.3 Removing a Memory

The implementation to remove a memory from Multics approaches the ideal operator interface -- for the usual configuration the operator merely issues the request to remove a specific physical memory module, and the system automatically removes the memory and informs the operator it

has been done. However, the GE-645 hardware design leads to the restriction that in any Multics configuration there is always one memory module that cannot be removed. The major events in the sequence to remove a memory module are as follows:

1. The system locks the reconfiguration data base and checks the operator request for validity.

2. If this memory is used as a primary interrupt source, a remaining memory module assumes this role.

3. All information in the module is either copied to another memory module or paged out of primary memory.

4. The system updates the reconfiguration data to show that the module is not being used, and makes the memory module inaccessible to the system.

5. The system unlocks the reconfiguration data base and advises the operator that the memory has been removed.

After locking the reconfiguration data base, the system verifies that the operator request is valid -- not only must the memory be in the current configuration but also there must remain at least one memory module for each processor to act as its relay point for interrupts. If a memory module to be removed is interlaced, then all the interlaced modules are removed as if they were one larger module. Recall that every processing module has permanently set "base address" switches specifying the address of the instruction (or channel command) executed in response to an interrupt (or channel connect)

signal -- by Multics convention all base addresses are in a single module called the bootload memory. The "base address" switches can only be changed while the processing modules are stopped, which means service to users must be interrupted while an operator changes the switches on all processing modules. Because of this service interruption and the likelihood of operator errors, the choice was made not to implement the (less than dynamic) bootload memory removal.

If the memory module to be removed serves as the primary interrupt source for some CPU, then the system selects some remaining memory as a replacement. If no other memory in the configuration directs interrupts to that CPU, then an operator changes a "control processor" switch using the same sequence as for CPU reconfiguration. Using the same techniques as when removing a CPU, the CPU's idle process is forced to run and change the "signal address table" to reflect the new primary interrupt source.

The crux of memory removal is moving the information stored in the module. First an implementation of the Make_unavailable primitive threads the storage blocks in the module out of the "available list" and onto a "removing list". Then an implementation of the Unbind primitive invokes the normal paging mechanism to "page out" (i.e., move to secondary storage) all the demand paged information. Unbind copies the wired down storage to another memory using the "trial copy" method -- copy, and then if modified stop all other CPU's and

copy again. It is noted that the Multics implementation is simplified by taking advantage of fact that the "bootload memory" module cannot be removed: the problem of moving page tables and the similar problem of moving unpaged information are avoided by placing this information in the bootload memory. All external input/output is done using buffers in the bootload memory only: this avoids the difficult relocation problems resulting from the fact that the general I/O controller uses absolute addresses rather than paging for all memory references. These implementation short cuts introduce no additional restrictions since all this stationary information easily fits into the bootload memory module.

After the memory is no longer storing information for the system, the reconfiguration data is updated to show the module is not being used. Since the port enable registers in processor modules cannot be changed while the system is running, the system tries to make the memory inaccessible to the system by disabling the port enable register in the memory module being removed. Then the reconfiguration data base is unlocked, and the operator is advised that the memory has been removed.

The hardware design of the port enable registers imposes some significant restrictions. Not only can a program in the development partition enable a removed memory to allow access from service system processors, but also initializing the development system will enable <u>all</u> memory module port enable

registers and then initialize (viz., halt) all processors on the enabled links; this problem results from the fact that the initialize signals ignore the program accessible port enable registers in the memory modules. Another design problem is that even though a port is disabled at the memory module, the module will still relay (interrupt) signals through that port as indicated by the "control processor" switch -- the target CPU will "hang up" trying to reference the inaccessible memory to determine exactly what type of signal was sent. To alleviate these problems, after removing a memory an operator must redirect the "control processor" switch to the development partition, and he must use manual switches on the memory module to disable links to the service system processor modules.

In summary, the Multics reconfiguration implementation is restricted in that the "bootload memory" module cannot be removed because of the fixed absolute address contained in the manual "base address" switches of each processor module. Memory removal has a convenient operator interface -- he usually just types one command. On the other hand, before the removed module can be used in another partition, the operator must typically set about five manual switches: this is because processor modules have no program accessible port enable registers and because the port enable registers in memory modules are not completely effective, viz., do not disable interrupt signals.

5.4 Adding a Memory

To initialize a primary memory module for adding to Multics the operator must manipulate about eleven switches -- eight port enable register switches, two initialize switches, and the "control processor" switch. The primary events in the sequence to add a memory module are:

1. The system locks the reconfiguration data base and checks the operator request for validity.

2. The operator initializes the memory -- the system gives detailed instructions to the operator.

3. The system updates the reconfiguration data.

4. The system makes the storage in the module available for demand paging.

5. The system unlocks the reconfiguration data and informs the operator that the memory module has been added.

With the reconfiguration data base locked, the operator request is checked to make certain that the memory is part of the installation but not already in the configuration. Again interlaced memory modules are treated as a single "module". In response to a valid request the operator is given detailed instructions for initializing the module (see Appendix I for an example). The operator manually disables all port enable registers (primarily so no interrupt signals can be received). Then he initializes the module in order to remove all pending interrupts and enable all program accessible port enable

registers. The operator directs the "control processor" switch to a CPU in the configuration -- this is safe since there are no interrupt signals pending. The program accessible (and now enabled) port enable registers are by manual switches allowed to control the links to all processor modules in the configuration -- since the ports in the processor module are always enabled for all memories, the physical memory module is "available" and "bound" to an address interval at this point.

The reconfiguration data base is updated to reflect that the physical memory module is now accessible to the system. Then the program accessible port enable registers are enabled for just those processor modules in the configuration (this has no immediate effect since links to processor modules not in the configuration are still disabled by the manual switch). Finally an implementation of the Make_available primitive makes the storage in the module available for demand paging by threading the storage blocks into the "available list". Then the reconfiguration data is unlocked and the operator is informed that the memory has been added. The operator can then manually change the switch that allows all links to the added module to be controlled by the program accessible port enable registers.

The lack of program accessible port enable registers in processor modules (for links to memory module) results in an intricate operator sequence to initialize the memory module to

be added. In addition the operator must manually initialize the port enable registers in the memory module to reflect the configuration.

5.5 Experience With the Multics Implementation

The Multics dynamic reconfiguration capability is used on a daily basis for the operation of the Multics system at M.I.T. The primary motivation is to increase the system capacity during hours of peak usage, and otherwise partition the equipment into independent "service" and "development" systems. Occasionally dynamic reconfiguration is also used to remove for repair a CPU with intermittent faults or a memory modules with parity errors. The author considered it a significant achievement that reconfiguration is always invisible to users of the system: regardless of what a user is doing he has absolutely no disruption when reconfiguration occurs, not even a noticeable variation in response time.

Since dynamic reconfiguration was one of the initial goals of Multics, the initial operating system was designed with a generality that made it relatively easy to add this capability. Still for the implementation of dynamic reconfiguration, the author added to Multics more than 3000 lines of PL/1 source code and 700 lines of assembly code that give rise to more than 12,000 words of instructions and data. The scope of this programming is indicated in Appendix II

which lists the Multics modules modified or added for reconfiguration. In addition to the reconfiguration programming identified in Appendix II, the author expended substantial effort fixing previously undetected multiple processor software bugs in the initial Multics operating system.

The Multics implementation was developed for hardware not designed with dynamic reconfiguration as a primary goal. However, useful dynamic reconfiguration is still possible, because the hardware was designed for operation with a wide variety of configurations. Although no changes in the basic design were made, the reconfiguration development stimulated corrections to minor hardware design errors that prevented operation with more than two memory modules and a design error that allowed system initialization signals thru disabled ports.

The limitations of the hardware design have two significant results: there is always one memory module in the configuration that cannot be removed, and there are numerous opportunities for an operator to make fatal errors when manipulating manual switches. A fatal error currently interrupts service to users for approximately 20 minutes while a "salvager" makes secondary storage self-consistent and while the system is reinitialized. In an effort to reduce operator errors, detailed instructions have been prepared (Appendix I), summary checklists are provided the operators, system

programmers provide training, the system prints explicit operator instructions for each reconfiguration request, and the system asks the operator for his personal initials in order to encourage a sense of responsibility. The current experience at M.I.T. is that an operator makes a fatal error in about one of every 100 reconfiguration requests, that is, about twice a month, but the need for operator accuracy is very high.

The basic architecture of the GE-645 is maintained in the newly announced and upward compatible Honeywell 6000 series computers -- Appendix III is an analysis of the deficiencies of this architecture and Appendix IV is a case study of how this architecture could be improved to provide a complete and convenient dynamic reconfiguration capability for Multics.

CHAPTER SIX

CONCLUSIONS

6.0 Summary of Results

We have developed an orderly design approach for dynamically changing the configuration of constituent physical units in a modular computer system. This design allows the modules in an installation to be partitioned into separate, noninterfering partition-systems in order to permit preventive maintenance, allow development of new operating systems, and change system capacity in response to fluctuations in the computational load. This design included the operating system primitives and hardware architecture to allow any primary or secondary storage module and any processing module to be added and removed while the system is running. We have considered reconfiguration as externally initiated by a human operator and accomplished automatically without disruption to any users of the system.

This thesis has developed reconfiguration as basically an extension of resource management. To make the reconfiguration operations clear, we have introduced three distinct views of resources:

1. Physical resources represent the actual hardware devices.

2. Logical resources represent the abstract processing

and memory capacity derived from the hardware devices.

3. Virtual resources represent the apparent processing and memory capacity of a process; virtual resources thus represent demands for logical resources.

This thesis uses the concept of binding to model the functions that manage a system's resources. With this model, hardware capabilities being applied to a process are represented by physical resources bound to logical resources which are in turn bound to virtual resources of the process. The problem of removing physical resources from the configuration concerns the reversibility of binding, and adding physical resources to the configuration is an example of delayed binding. This reconfiguration model is developed in terms of a modular structure with an interconnection of processing and storage modules, such as is common in large contemporary computer systems.

6.1 Implications

Reconfiguration is viewed as changes in binding. From this model ground rules which permit reconfiguration have been identified, and we will now review their major implications. The basic ground rules are summarized below:

1. Each physical module must be interchangeable with any other similar module.

2. A human operator must select the particular modules

available for use by a system.

3. The system must be able to automatically change the configuration (viz., binding to logical resources) of its physical modules.

4. The system must be able to dynamically change the set of logical resources it is using.

These rules are intended to provide a basis for the orderly design of future systems that require dynamic reconfiguration: these rules influence the hardware architecture, operating system, and operation of a computer system. The usefulness of these rules has already been demonstrated by using them to provide a practical dynamic reconfiguration capability for the Multics system at M.I.T., where reconfiguration has had a significant impact on the daily operation of the computer installation.

6.1.1 Module Interchangeability

To be effective, dynamic reconfiguration must be equally applicable to all individual modules of a given type (e.g., all central processing units). This implies that the hardware design avoids any implicit or explicit relationship between individual modules (for example, a processor module must not require the use of some specific memory module). That is, the hardware design allows each module to be used interchangeably with any other module of the same type.

6.1.2 Operator Participation

The dynamic nature of reconfiguration implies that operator participation in reconfiguration operations should be limited. This implication is primarily motivated by the need for reconfiguration operations to be reliably performed: program controlled operations require human perfection only once (when the reconfiguration program is implemented), but operator controlled operations require perfection for every reconfiguration. Still, having coexistent, independent partition-systems within a single installation implies that an operator must intervene to specify the modules to be included in each partition-system. This is a simple selection operation (completely decoupled from the complex operations that control the configuration of the modules), and it should be implemented with a simple human interface.

6.1.3 Automatic Configuration Control

Although nearly all contemporary computer systems manually determine the configuration of modules (for example, the assignment of addresses to memory modules), dynamic reconfiguration implies that the configuration should be controlled automatically. There are two techniques that can be used: a particular configuration control mechanism either has a permanently assigned value or else is under the explicit (program) control of the operating system.

6.1.4 Resource Utilization

Dynamic reconfiguration implies that the system must be able to change the (logical) resources it is using to meet any (virtual) resource demands. The operations that make those changes require the speed and accuracy of a computer program -- in particular, the operation must be fast enough that the system can still meet its response time constraints, and thus can be considered to be "continuously operating".

For processing modules, the primary implications are that no particular processor must be indispensable to the continued progress of any process, and it must be possible to "preempt" the execution of any specific processor.

For storage modules, the primary implication is that it must be possible to relocate any information, including such things as the "resident supervisor". The intrinsic problem is being able to locate and update all absolute addresses: by using the hardware mechanisms of paging, the relocation operations are basically independent of the information being relocated. An obvious corollary of this implication is that the system must in no way require the use of any fixed absolute address.

6.2 Additional Research

The research reported in this thesis provides a basis for additional investigation in three areas -- spontaneously initiated reconfiguration, reconfiguration of modules involved in external input/output, and reconfiguration of very large capacity secondary storage modules.

As computer systems try to approach the goal of continuous availability, one obvious approach is spontaneous error detection and error recovery. A dynamic reconfiguration capability for replacing faulty hardware modules is an essential element of such ultra-reliable computation capabilities. One approach indicated by our general model is to consider all the modules of the installation as part of one "super system" and have this "super system" spontaneously perform the (manual) functions of the operator interface matrix (of chapter four). The reconfiguration design is complicated by the need to interface with procedures which retry the failed operation, attempt to salvage damaged portions of the computation, and identify the source of the errors. Automatic error recovery, of course, introduces the additional problem that errors may be present in the system performing the reconfiguration.

External input/output, including such synchronous applications as real time processing, does not directly fit into our generally asynchronous model. One difficulty is that

during processor reconfiguration we must "preempt" the processor to stop the execution of the currently running process; however, it is not in general acceptable to stop the I/O channel running an external "I/O process". What seems to be needed is a way to move the "I/O process" from one channel to another without any interruption. In terms of our binding model, external I/O seems to imply that the virtual channel resource must always have a valid binding to some logical channel. The problem is somewhat analogous to the problem of "wired" information which must always have a valid binding to some (logical) storage resource.

Although our reconfiguration ground rules apply equally well to primary and secondary storage, very large modules are somewhat intractable. Our approach to removing a storage module is basically to copy the stored information to a new location. However, with a very large capacity module, such a copy operation requires a large amount of processing. Additional research might consider such alternatives as maintaining duplicate copies of information, providing direct transfer of data from one secondary storage module to another secondary storage module, or using a movable storage medium (e.g., moving a "disk pack" from one drive to another).

6.3 Remarks

We have seen that it is possible to, in an orderly fashion, design a modular computer system with the ability to dynamically change the configuration of constituent modules. One final point is emphasized: although the system designer can (by careful attention to the ground rules given in this thesis) provide dynamic reconfiguration, he can also (by ignoring these ground rules) make it difficult, if not impossible, to later include dynamic reconfiguration as part of the operating system. Hopefully the reconfiguration ground rules developed in this thesis are a significant contribution towards a systematic engineering approach for building a large computer system for use as a computer utility.

BIBLIOGRAPHY

[1] Corbató, F. J. and Vyssotsky, V. A. Introduction and Overview of the Multics System. Proc AFIPS 1965 Fall Joint Computer Conference., Vol 27, Part 1. Spartan Books, New York, pp 185-197.

[2] Stanga, D. C. Univac 1108 Multiprocessor System. Proc AFIPS 1967 Spring Joint Computer Conference, Vol. 30, Thompson Books, Washington D. C., pp 67-74.

[3] Thompson, R.N. and Wilkinson, J.A., The D825 Automatic Operating and Scheduling Program, Proc AFIPS 1963 Spring Joint Computer Conference, Vol 23, Spartan Books, Washington D.C., pp 41-49.

[4] GE-635 System Manual, CPB-371A, General Electric Company, Phoenix, Arizona, July 1964.

[5] Witt, Bernard I., M65MP: an Experiment in OS/360 Multiprocessing, Proceedings of 23rd ACM National Conference, ACM Publication P-68, Brandon/Systems Press, Inc., Princeton, N.J., 1968, pp 691-703.

[6] Glaser, E. L., Couleur, J. G., and Oliver, G. A., System Design of a Computer for Time Sharing Applications. Proc AFIPS 1965 Fall Joint Computer Conference, Vol. 27, Part 1 Spartan Books, New York, pp 197-202.

[7] Vyssotsky, V. A., Corbató, F. J. and Graham, R. M., Structure of the Multics Supervisor, Proc AFIPS 1965, Fall Joint Computer Conference, Vol. 27, Part 1, Spartan Books, New York, pp 203-212.

[8] Ossanna, J. P., Mikus, L. E., Dunten, S. D., Communications and Input/Output Switching in a Multiplexed Computing System, Proc AFIPS 1965 Fall Joint Computer Conference, Vol. 27, Part 1, Spartan Books, New York, pp 231-241.

[9] Dennis, Jack B., Programming Generality, Parallelism and Computer Architecture, Computation Structures Group Memo No. 32, Project MAC, MIT, Cambridge, Mass., 1968.

[10] IBM System/360 Model 67 Functional Characteristics, GA27-2719-1, International Business Machines Corporation, Kingston, N.Y., January 1970.

[11] Burroughs B6500 Information Processing Systems Reference Manual, Burroughs Corporation, Detroit, Mich., 1969.

[12] Harr, J. A., Taylor, F. F., and Ulrich, W., Organization of No. 1 ESS Central Processor, The Bell System Technical Journal, Volume XLIII, Number 5, Part 1, September 1964.

[13] R. W. Parker, The Sabre System, Datamation, September 1965, pp 49-52.

[14] Sackman, Harold, Computers, System Science, and Evolving Society, John Wiley & Sons, N.Y., 1967, pp 91-167.

[15] Pokorney, Joseph L. and Mitchell, Wallace E., A Systems Approach to Computer Programs, ESD-TR-67-205, Technical Requirements and Standards Office, ESD, L. G. Hanscom Field, Bedford, Mass, February 1967.

[16] Keely, J. F., et al., An Application-oriented Multiprocessing System, IBM Systems Journal, Vol. 6, No. 2, 1967.

[17] Dijkstra, Edsger W., The Structure of the "THE" Multiprogramming System, Communications of the ACM, Vol. II, No. 5 (May 1968), pp 341-346.

[18] IBM System/360 Principles of Operation, A22-6821-6, International Business Machines Corporation, Poughkeepsie, N.Y., January 1967.

[19] Andrews, J., et al., GE-645 Processor Reference Manual, G0098, General Electric Company, Cambridge Information Systems Laboratory, Cambridge, Mass., August 1970.

[20] Saltzer, Jerome H., Traffic Control in a Multiplexed Computer System, MAC-TR-30 (Sc.D. Thesis), Project MAC, MIT, Cambridge, Mass., July 1966.

[21] Bensousan, A., et al., The Multics Virtual Memory, Second ACM Symposium on Operating System Principles (October 1969), Princeton University, pp 30-42.

[22] Daley, Robert C., and Dennis, Jack B., Virtual Memory, Processes, and Sharing In Multics, Communications of the ACM, Vol. II, No. 5, (May 1968), pp 306-312.

[23] Schroeder, Michael D., Performance of the GE-645 Associative Memory While Multics is in Operation, Proceedings of the ACM SIGOPS Workshop on System Performance Evaluation (April 1971), Harvard University, pp 227-245.

APPENDIX I

MULTICS OPERATOR INSTRUCTIONS

This appendix is a copy of the dynamic reconfiguration instructions available to the Multics computer operators. These instructions refer to "MBOS configuration cards": this is a reference to a set of operator supplied punched cards specifying the initial hardware configuration. These cards are interpreted by the Multics Bootload Operating System (MBOS), and during system intitialization MBOS places in core memory a copy of the configuration information for use by Multics itself.

TO: Distribution

FROM: Roger R. Schell

DATE: October 23, 1970

SUBJECT: Use of Dynamic Reconfiguration in Multics

1. Multics now has a full capability for dynamic reconfiguration of processors and memories. This will make it possible for operations to add or remove any processor and any memory except the bootload (i.e., the low order) memory while the system is running. There must, of course, be a minimum of one processor, and one memory for each processor. The reconfiguration commands can be issued from the initializer console or any daemon process at Multics command level (or in "admin" mode).

2. The following is a sample console output for adding a processor (underlined portions are typed by the operator):

- -

addcpu b 5

Check that the following has been done (if not, do it in the following order):

cpu b must be initialized (depress INITIALIZE switch on processor)

All memories: PORT ENABLE (port 5) set to MASK
cpu b must have the following switch setting:

clock a:	Port Block 5	Interlace OFF
memory c:	Port Block 0	Interlace ON
memory d:	Port Block 1	Interlace ON
memory e:	Port Block 2	Interlace OFF

CPU NO: 2

Have all the above been done?(yes/no and initials): yes rrs

You will change CONTROL PROCESSOR switch on memory d to port 5 (cpu b).
Wait until instructed to change it. Are you ready? (yes/no): yes
Change switch now. Type 'yes' when done: yes

cpu b is now running.

- -

The arguments to the addcpu command are the processor name (e.g., "b") and the processor port (e.g., "5") as on the MBOS configuration card. An additional argument can optionally be provided to specify that a particular memory controller (e.g., "b") is to be given the added CPU as its "control processor". Be certain that when the new CPU is being readied, the various switches are set in the order listed: check very carefully, as wrong switches can crash the system.

3. The command for removing a processor is of the form delcpu b. The sequence of instructions (as for adding a CPU) for changing the control processor switch will be given for each controller assigned to the CPU being removed.

4. The following is a sample console output for adding a memory:

- -

addmem e

Perform the following (in the order given) on memory controller e.

PORT ENABLE set to OFF for all ports.
Initialize controller at its maintenance panel.
Change CONTROL PROCESSOR switch to port 4 (cpu a).
PORT ENABLE set to MASK for ports 0, 3 and 4.

Have all the above been done?(yes/no and initials): yes rrs

You have added memory controller e.

- -

The argument to the addmem command is the memory name (e.g., c) as on the MBOS configuration card. An additional argument can optionally be provided to specify a particular cpu (e.g., a) as the control processor for the memory being added. Be certain that when the new memory is being readied, the various switches are set in the order listed. In order to initialize the controller at its maintenance panel, the controller must

be in the test mode using the test switch. While in the test mode, press the initialize button. Then set the controller from the test mode to the operate mode.

5. The command for removing a memory is of the form delmem e. If this is the only memory whose control processor switch is directed to some processor, then the operator will be given the sequence of instructions (as for adding a processor) for changing the control processor switch on some other memory to this processor.

6. When the reconfiguration commands request the operator to give a "yes/no" answer, then any answer other than yes will properly terminate the reconfiguration. When the operator is asked for a "yes" answer, he must do as directed and answer "yes" or risk crashing the system. If the error message "program error--notify programmer" ever occurs, DO NOT try to use ANY reconfiguration command again until cleared by a programmer, or until the next bootload of the system.

7. Note that when the configuration is changed dynamically, the MBOS cards for processors and memories may no longer reflect the true configuration. Therefore, before the next bootload or salvage, the hardware configuration and the MBOS configuration deck must be made consistent with each other.

8. Only memories that are defined by MBOS cards at bootload time can be added. The configuration card for a memory has a

9. The PORT ENABLE switches for all memory controllers running on the service system should be set in the "MASK", i.e., "IN LINE", position for all ports at all times. For the development system, all ports should be "ON" or "OFF" to reflect the running configuration. Note that after adding a memory, unused ports are left in the "OFF" position. The operator has the option of either leaving them "OFF" or putting them in the "MASK" position after the adding of the memory is completed.

10. When moving the processor and memory from the development to the service system, the changing of switches by the operator can be minimized by using the following sequence. If the development system has a processor P with port X and a memory M, then to move these to the service system give the following commands in this order:

```
addmem M
addcpu P X M
```

When removing a processor and memory from the service system to form a development system, changing of switches is minimized by removing the non-bootload processor and the high order memory using the following commands in this order:

```
delcpu P
delmem M
```

APPENDIX II

SUMMARY OF MULTICS SUPERVISOR CHANGES FOR RECONFIGURATION

Module Name	Source Language	Reason For Addition or Change	Number of Lines (changed or added) Source	Object
add_memory	PL/I	Set up memory controller for interrupts and interrupt masks	100	460
bootstrap1	alm(1)	Determine initial configuration	10	10
bootstrap2	alm	Save hardware provided data	2	2
cl_dcm	PL/I	Make call to keep I/O buffers at fixed absolute address	4	40
emergency_shutdown	alm	Remove multiple-cpu conflict	4	4
fault_init	alm	Provide for multiple-cpu clear associative memory	8	8
freecore	PL/I	Primitive to add core to pool for paging	50	245
hphcs_	alm	Highly priviledged gate for reconfiguration commands	9	9

(1)

"alm" is the assembly language for Multics.

ii	alm	System stop/restart for multiple processors	130	130
init_processor	alm	Start execution of a halted processor	260	260
init_sst	PL/I	Initialize a map of core usage	60	180
initialize_faults	PL/I	Set up fault and interrupt vector for reconfiguration	200	925
make_fv_code	alm	Create code for loading DBR of stopped cpu	20	20
make_sdw	PL/I	Set up processor data base so that processor can be removed	5	25
master_mode_init	alm	Experiment to determine if memory controllers are interlaced	40	40
master_mode_ut	alm	Clear associative memory; move wired-down pages	120	120
master_pxss_page	alm	Recognize multiple cpu error condition	10	10
mini_gim_init	PL/I	Provide for error messages for wrong configurations	1	10
page	alm	Add primitives for using absolute core address	40	40
pc	PL/I	Prevent use of pages being removed	10	60
pc_abs	PL/I	Primitive to remove pages from paging pool	530	1680

pc_wired	PL/I	Primitive to wire down pages	50	150
prds	alm	Per processor reconfiguration data	5	5
prds_init	PL/I	Set up per processor reconfiguration data	5	35
pxss	alm	Create idle process for processor to be added	20	20
reconfig	PL/I	Maintain reconfiguration data base	940	2488
reconfigure	PL/I	Commands for reconfiguration	620	2185
scas_init	PL/I	Initialize reconfiguration data for memory	175	954
scs	alm	Reconfiguration data base	50	50
scs_init	PL/I	Set up reconfiguration data	50	250
shutdown	PL/I	Allow varying number of processors	3	20
signal_0	PL/I	Allow varying number of processors	3	15
start_cpu	PL/I	Set up traffic controller data for new processor	200	810
stop_cpu	PL/I	Clean up traffic controller data for removed processor	180	650
syserr	PL/I	Allow error message for wrong configuration	4	20
system_control_	PL/I	Provide command for operator	15	40

system_meter	alm	Allow varying number of processors	5	5
tc_init	PL/I	Allow varying number of processors	4	20
update_sst_pll	PL/I	Update map of core usage	20	100
wire_proc	PL/I	Primitive to temporarily wire down reconfiguration procedures	150	600
wired_fim	alm	Stop processor while updating address for moved wired page	15	15

SUMMARY OF CHANGES AND ADDITIONS

	alm	PL/I	Total
Number of modules modified	16	16	32
Number of new modules added	1	8	9
Total lines of source changed or added	748	3379	4127
Total lines of object changed or added	748	11962	12710
Total lines of object in Multics supervisor before reconfiguration added (approximate)	9000	171000	180000

APPENDIX III

ANALYSIS OF CURRENT MULTICS HARDWARE FOR RECONFIGURATION

INTRODUCTION

The discussion of the Multics reconfiguration capabilities in chapter five pointed out a number of problems resulting from the hardware design. This appendix points out the features of our general design approach not included in the current Multics hardware, and shows how these omissions lead to the observed operational restrictions.

The primary hardware design ground rules presented in the body of this thesis are summarized below:

1. Processing modules require relocation hardware so that the operating system can locate and update all absolute storage addresses during reconfiguration.

2. The intermodule connection network is required to treat modules of like type homogeneously.

3. It should be convenient to permit an operator to partition the modules into independent partition-systems.

4. The operating system should be able to automatically (viz., without operator intervention) control the configuration of its hardware modules.

Since these ground rules are really design constraints on basic system capabilities, we can readily make a comparison between these features and the existing Multics design. We

will now separately consider each of these four design areas.

STORAGE RELOCATION HARDWARE

Storage reconfiguration primitives of chapter three require that only selected supervisor routines explicitly use absolute addresses for storage references; all other programs use only relocatable virtual addresses. The system must be able to dynamically update any absolute addresses it does use.

Multics comes very close to the desired design in that a (primary or secondary) storage absolute address is never explicitly used, except by the supervisor. However, the central processing unit (CPU), general I/O controller, and drum controller all have a "base address" set by manual switches: this "base address" implements the concept of a "signal address register" introduced in chapter four. The problem is that the system cannot update this (switch controlled) absolute address, except by stopping the system and having an operator change the switches. The operational effect is that any memory module containing one of these "base addresses" cannot be dynamically removed.

In addition the Multics I/O controllers have no general relocation mechanism, but rather directly use absolute addresses for all memory references. However, since only the supervisor does I/O, it is possible (although not included in the current Multics) to locate and update all absolute

addresses; any relocation design must essentially make a special case for each place where an absolute address may be used. The practical result is that an unduly complex design would be needed in order to relocate primary storage used for I/O.

UNIFORM INTERMODULE COMMUNICATION

A basic ground rule in our reconfiguration design (as introduced in chapter two) is that the hardware allows each module of a given type (e.g., each CPU) to be used interchangeably with any other module of that type. This rule means that the hardware must permit the same types of signals over all links from a module to a given type of module; the operating system will determine which signals are actually used.

The Multics hardware generally provides a uniform treatment of similar modules, with one notable exception. Of all the links from a memory module to CPU's, only one link can be used for relaying interrupt signals to a CPU. This "control processor" is specified by a manual switch. To change the CPU receiving interrupts, all interrupts from the memory must be stopped while an operator moves the switch. The operational result is that, when adding and removing CPU's or memories, the operator must frequently move the "control processor" switch to make certain that every CPU is the

"control processor" for at least one memory module. Not only is there opportunity for operator error, but also the related design is complicated by such things as artificially generating interrupts in place of those interrupts stopped while the switch is moved.

An additional communications network deficiency is that the current clock module (treated by Multics as a special "memory" module) can only have interface links to a maximum of two CPU's, although a Multics installation could include up to about five CPU's. The operational impact is that any installation with more than two CPU's must restrict the possible CPU/clock combinations. These restrictions increase the number of clocks required and reduce the system availability.

Finally, the "base address" of each processing module is restricted to addresses less than 256K words (viz., an 18 bit address), although the system in general uses 24 bit absolute addresses. Operationally this means that every configuration must include a memory module with absolute addresses in the range 0-256K words.

OPERATOR SELECTION OF CONSTITUENT MODULES

Totally independent partition-systems within one installation is a major goal of reconfiguration. If such partition-systems are truly independent, an operator must

intervene to add or remove a module from a partition-system. Chapter four introduced the concept of an "operator interface matrix" to emphasize that this simple operator selection is a function distinct from the much more complex operations needed to control the configuration of the modules in a partition-system.

The Multics hardware has no explicit mechanism to implement the operator's selection functions provided by the operator interface matrix. Rather, the partitioning of modules is merely implicit in various manual configurations control mechanisms. These mechanisms require tedious and precise operator switch manipulations in order for an operator to assign a module to a different partition-system. The primary operational result is frequent operator errors. Errors that are made usually result in a major interruption of service, since the operator must manipulate switches on modules being actively used by the partition-system.

AUTOMATIC CONFIGURATION CONTROL

A crucial hardware feature for dynamic reconfiguration is that only the system (not an operator) controls the mechanisms that determine configuration of the modules assigned to a system. With this automatic control, configuration changes are made accurately and rapidly: to be automatic, configuration control mechanisms must either be permanently

set, or else program accessible. In chapter four we explicitly identified the configuration control features needed for a modular computer system. These are listed below:

1. "Port enable registers" in every module.
2. "Initialize registers" in every module.
3. "Address interval registers" in processing modules.
4. "Processor number registers" in processing modules.
5. "Signal address registers" in processing modules.

We will now evaluate how the Multics hardware implements each of these configuration control functions.

Port Enable Registers

Port enable registers are needed to control all intermodule communication, except that required for accessing the configuration control mechanisms themselves. In Multics a module's port enable features are primarily under the control of manual switches which must be changed by an operator whenever a module is added to or removed from a system. This aspect of the current Multics design has the greatest adverse impact on system operation. Listed below are some of the more significant operational problems that have actually been experienced:

1. The clock module of the current Multics hardware has no mechanism for disabling a link to a processor; therefore, references to the clock by a processor not in the configuration can stop the clock, and without a working clock

inconvenience, and errors can lead to adding a "dangerous" module (e.g., a memory module with pending undefined interrupts).

Address Interval Registers

Address interval registers are needed in processing modules, in order to assign address ranges to memory modules. Multics uses manual, multiple-position switches (awkwardly located behind closed panels) to control address assignment. For some positions the switch cannot be moved without momentarily "crossing" the address range assigned to another memory: this momentary assignment of the same address range to two different memory module leads to undefined behavior. The impact is that before starting the system the operator must carefully (at each processing module) assign addresses to all the memories that might ever be dynamically added.

Processor Number Registers

A processor number register is used to uniquely identify each physical processor. In a Multics CPU this is implemented as a manual switch (behind a closed panel), and I/O channels have channel numbers assigned by pluggable circuit boards. Since Multics uses a processor number as a serial number, this implementation creates little difficulty, because it is seldom changed. However, one problem is that test and diagnostic programs require specific CPU and channel numbers, and a

maintenance technician may leave the same number assigned to more than one CPU or leave an invalid channel number.

Signal Address Register

A signal address register is needed to specify the location of the instructions executed when a processor receives signals from other processors. Multics implements this function with manual "base address" switches. The inability to automatically change this address restricts memory reconfiguration, as already discussed above in the "Storage Relocation Hardware" section.

CONCLUSIONS

The current Multics hardware design not only limits the reconfiguration capability, but also introduces many opportunities for serious operator errors. The primary cause of these problems is the use of manual switches to perform functions requiring the speed and accuracy of program accessible registers. Although many of the problems of operator errors are present when starting a system on a static configuration, the impact of a few minutes delay while repeating an unsuccessful attempt to start the system is dramatically less than the impact of "crashing" the system during dynamic reconfiguration while 50 users are in the midst of interactive sessions.

APPENDIX IV

MULTICS HARDWARE DESIGN PROPOSAL FOR RECONFIGURATION

INTRODUCTION

The body of this thesis provides general ground rules for designing hardware and operating systems in order to provide dynamic reconfiguration. These ground rules have been applied to the Multics operating system (as described in chapter five) to implement a dynamic reconfiguration capability. However, the current hardware design limits this initial reconfiguration implementation, as discussed in Appendix III. Since major hardware changes were not permitted as part of the initial reconfiguration implementation, this appendix is included to propose a hardware design that would allow an unrestricted dynamic reconfiguration capability.

This appendix will concentrate on the features essential to reconfiguration, although a few "nice to have" improvements are also noted. It is noted in passing that Project MAC at M.I.T. is currently considering a "follow-on" Multics implementation on an upward compatible Honeywell 6000 series computer in order to benefit from more advanced circuit technology. Cases where this proposed "follow-on" design already includes adequate solutions to reconfiguration problems will be pointed out.

As noted in Appendix III, there are four major hardware

3. The current address assignment switches in processing modules are replaced by "address interval registers". For any link with the port disabled, the address assigned is immaterial--for example, the same address may be assigned to more than one memory module as long as only one memory has an enabled link.

4. The current CPU number switch and channel number plug boards are replaced by "processor number registers". (Permanently assigned manual settings would be an equally satisfactory alternate design approach.)

5. The current base address switches of each processing module are replaced by "signal address registers". The signal address register specifies a full 24 bit address: any operations (such as the CPU "absolute mode") restricted to 18 bit addresses will append the high-order 6 bits of the signal address register to all absolute addresses generated.

The above configuration control registers are normally only set and read by a special "configuration channel" contained in each general I/O controller (although a "maintenance mode" may allow the registers to be switch-controlled). Each module is individually accessed by this configuration channel, and difficulties (e.g. power not on) are reflected in the status returned by the channel.

The current bootstrap channel in the general I/O controller is modified to use the configuration control registers during system initialization, as described in

chapter four. The configuration channel is used to perform the functions of the current "system initialization signals".

OPERATOR SELECTION OF CONSTITUENT MODULES

To prevent one independent partition-system's configuration channel from interfering with another partition-system, an "operator interface matrix" (as defined in chapter four) is used to control which configuration control registers each partition-system can set. If a configuration channel attempts to set registers in another partition-system, a distinctive status is returned so that the operating system knows that no registers were actually set. The configuration channel can also read the operator interface matrix information for each module. Each general I/O controller module has a "send register" in the matrix itself, which can of course be set (as described in chapter four) to control which configuration channels a partition-system can use.

It would be convenient if the operator interface matrix were part of a general purpose operator's console that included an interactive terminal (one for each potential partition-system) for issuing reconfiguration requests and other partition-system control requests. To minimize down time, there should be two copies of the operator interface matrix, each with a simple on/off-line switch for specifying

the matrix to be used -- changing from one matrix to the other should not disrupt any partition-system (it should be quite easy to avoid disruption, since the operator interface matrix affects the operation of a partition-system only during reconfiguration and initialization). It would be nice if each matrix were itself included as just another module on the operator interface matrices. For each matrix, the configuration channel would then be able to read the setting of the on/off-line switch and all other operator switches -- a partition-system could then guide and monitor an operator's actions, for example when changing the on-line matrix.

Errors using the operator interface matrix can usually be detected before harm results; however, a convenient safeguard is an alarm (and manual "override" for ignoring the alarm) when attempting to make unavailable a module currently being "used" by a system-partition. To implement such an alarm, program accessible registers are included in the matrix, so that a partition-system can indicate which modules it is using.

A primary emphasis in designing the operator interface matrix should be making it convenient for the operator, since this is the only place where the operator manipulates switches during reconfiguration.

UNIFORM INTERMODULE COMMUNICATION

The currently contemplated Multics follow-on hardware design solves the remaining problems related to uniform treatment of similar modules. Clock modules interface with all CPU's in the installation. Furthermore, a memory module can send interrupts to any of its interfacing CPU's. The system can control the interrupts by using a separated mask for each CPU.

STORAGE RELOCATION HARDWARE

The troublesome "base address" switches should be replaced with a register, as discussed above. However, although not essential to reconfiguration, it is proposed, for design simplicity, that a CPU have no "absolute mode" of operation. By reserving a fixed segment number for the fault vector, the Descriptor Base Register (DBR) serves as the "base address" for the processor. This implies of course, that the configuration channel can read and set the DBR.

Finally, we note that the memory reconfiguration design is made complex if I/O channels explicitly use absolute addresses. It would be nice if I/O modules used an appending mechanism (viz., segmentation and paging with "used" and "modified" bits) for all control and data references. For example, one approach is to have the base address register become a Descriptor Base Register, and have the "mailbox" of

the current Multics design become a Descriptor Segment with fixed segment numbers for the control and data references for each channel. Since absolute addresses would not be used directly, information could easily be relocated using the same techniques Multics uses for CPU information. If copies of absolute addresses are (for efficiency) maintained internally by a channel, then there must be a controlled way to clear these addresses (just as for the associative memory of a CPU) during reconfiguration.

CONCLUSION

The Multics hardware can support a substantially improved reconfiguration capability if the configuration control mechanism are manipulated by program rather than by an operator: such a design has been proposed. This design would essentially eliminate system failures attributable to reconfiguration.

CS-TR Scanning Project
Document Control Form

Date : 1 / 23 / 96

Report # LCS-TR-86

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: 190 (196-IMAGES)
Not to include DOD forms, printer instructions, 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): FOLLOWS LAST PAGE (189)

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP:	(1-190)
UNM'ED TITLE PAGE, BLANK	2-189, UNM'ED
SCANCONTROL, COVER, DOD, TARGETS (3)	(191-196)

Scanning Agent Signoff:
Date Received: 1 / 23 / 96 Date Scanned: 1 / 24 / 96 Date Returned: 1 / 25 / 96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Massachusetts Institute of Technology Project MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP None

3. REPORT TITLE

Dynamic Reconfiguration in a Modular Computer System

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Ph.D. Thesis, Department of Electrical Engineering, May 1971

5. AUTHOR(S) *(Last name, first name, initial)*

Schell, Roger R.

6. REPORT DATE June 1971	7a. TOTAL NO. OF PAGES 190	7b. NO. OF REFS 23
8a. CONTRACT OR GRANT NO. Nonr-4102(01) b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) MAC TR-86 (THESIS) 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES None	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

This thesis presents an orderly design approach for dynamically changing the configuration of constituent physical units in a modular computer system. Dynamic reconfiguration contributes to high system availability by allowing preventive maintenance, development of new operating systems, and changes in system capacity on a non-interference basis. The design presented includes the operating system primitives and hardware architecture for adding and removing any (primary or secondary) storage module and associated processing modules while the system is running. Reconfiguration is externally initiated by a simple request from a human operator and is accomplished automatically without disruption to users of the system. This design allows the modules in an installation to be partitioned into separate noninterfering systems. The viability of the design approach has been demonstrated by employing it for a practical implementation of processor and primary memory dynamic reconfiguration in the Multics system at M.I.T.

14. KEY WORDS

Modular Computer Systems
Computer Utility
Multics
Multiplexed Computers
Dynamic Reconfiguration
Operating Systems

DD FORM 1 NOV 68 1473 (M.I.T.)

UNCLASSIFIED

Security Classification

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: 1/24/1996
M.I.T. Libraries
Document Services