MIT/LCS/TR-171

# A MULTI-PROCESS DESIGN OF A PAGING SYSTEM

Andrew R. Huber

December 1976

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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

# A MULTI-PROCESS DESIGN OF A PAGING SYSTEM *

by

Andrew R. Huber

## ABSTRACT

This thesis presents a design for a paging system that may be used to implement a virtual memory on a large scale, demand paged computer utility. A model for such a computer system with a multi-level, hierarchical memory system is presented. The functional requirements of a paging system for such a model are discussed, with emphasis on the parallelism inherent in the algorithms used to implement the memory management functions.

A complete, multi-process design is presented for the model system. The design incorporates two system processes, each of which manages one level of the multi-level memory, being responsible for the paging system functions for that memory. These processes may execute in parallel with each other and with user processes. The multi-process design is shown to have significant advantages over conventional designs in terms of simplicity, modularity, system security, and system growth and adaptability. An actual test implementation on the Multics system was carried out to validate the proposed design.

Thesis Supervisor: David D. Clark
Title: Research Associate

---

*This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, on May 19, 1976 in partial fulfillment of the requirements for the degrees of Master of Science and Electrical Engineer.

ACKNOWLEDGEMENTS

I wish to thank my advisor, Dave Clark, for his patience in what has been a rather protracted effort. The original idea for this thesis is due to him. Three people were of great help to me in implementing the design presented in this thesis: Bernie Greenberg explained many of the mysteries of Multics page control and gladly contributed his time, knowledge and enthusiasm. Bob Mabee implemented some of the code necessary to permit page control to be implemented on Multics as parallel processes, and helped in getting the design working on Multics. Doug Wells was expert at finding my programming errors and explaining the pitfalls of PL/1. Without their help, I would still be debugging. Many other members of the Computer System Research Division contributed in ways too numerous to mention.

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

TABLE OF CONTENTS

LIST OF FIGURES

# CHAPTER 1

## Introduction

This thesis will examine a general multiple process design of a paging system. Such a design could be used in the implementation of a demand paged memory in any suitable computer operating system. As computer systems have grown in size, the operating systems have also greatly increased in size, scope, and complexity, especially so-called computer utilities and large time shared systems. The design presented here offers a method for simplifying one large component of such systems: the memory management task. The resulting system is less complex yet readily expandable to accomodate future systems growth.

There are two central concepts underlying the design presented in the following chapters. These are the concept of a process as an abstraction of a program in execution, and the concept of paging as a means of implementing a virtual memory. Before the motivation for designing a paging system as cooperating processes can be discussed, these two concepts warrant closer examination.

## 1.1 Processes

The essence of a process is the execution of a program. Numerous definitions of a process are given by various authors [Da68] [Ha70] [Di68a] but all include the notion of an execution point passing through the instructions of some program. Thus a process is an abstraction of the locus of control that passes through an executing procedure [De66].

The address space of a process, that is, the set of all memory addresses the process may reference, is an important component of a process. In fact, the address space of a process influences the computations the process can carry out to such an extent that we include the address space in our definition of a process. A process, then, consists of a pair: an execution point, or locus of control, and an address space.

The process abstraction provides a natural way of describing an operating system. Each user's work is viewed as a process, i.e. a task to be performed. The operating system itself is seen as a task or process manager. The various facilities the operating system provides, such as memory or device management, can themselves be implemented as processes. Two good examples of systems designed around the process concept in this manner are Dijkstra's THE system [Di68b] and a multiprogramming system described by Hansen in [Ha70].

In any multi-processor computer system, processes offer a straightforward technique for achieving multi-processing (the simultaneous execution of two or more programs). Any physical processor (CPU) in the system can execute any user or system process. This permits the operating

system to be multi-processed, i.e. different functions of the operating system may be executed in parallel. Parallel execution of the operating system, or one component of the operating system (the paging system) is a central theme in this thesis.

## 1.2 Paged Systems

Paging is a common strategy for solving the memory allocation problem, one of the chief tasks any operating system must perform. Examples of systems using paged memories include Multics [Da68], TENEX [Mu72], and IBM's VS systems [Wh74] [Sc73].

In a paged system the address space of a process is divided into contiguous pieces of fixed size called pages. Physical memory is partitioned in the same manner into contiguous blocks called page frames. When allocating memory to a process, any available page frame may be allocated to hold any page.

Usually the memory of a large computer utility is organized into several physical levels L1, L2, ... Ln. The access time and capacity of a level increases with n, and each level is normally a different type of memory device. For such devices, the smaller the access time the higher the cost per bit and therefore the smaller the capacity. By combining such components with widely varying speeds and size into a multi-level memory an overall memory system can be constructed whose capacity equals that of its largest component yet whose effective speed approaches that of its fastest component.

In such multi-level memories a process may reference only pages residing in the primary (level 0) memory. Referencing a page not allocated a page frame at the lowest level results in a page fault, an event which causes the necessary operating system mechanisms to be invoked to allocate a level 0 page frame to the page and cause the page to be read into that page frame. The operating system modules and the data bases these modules use to perform this task are called the paging system, or page control. Page control is a resource manager; page frames being the resource page control manages.

## 1.3 Paging Systems as Processes

There are many alternative methods for organizing and implementing the paging system functions. The most widely used is to have the user process itself perform the necessary memory management functions when needed, just as with any other system call. That is, the code that carries out the necessary operations to allocate page frames is executed in the user's address space just like a user program.

This thesis will examine several ways for organizing paging systems as processes. The paging system can be broken down into several activities, for example, removing pages from primary memory when it becomes full to make room for other pages. In such a system, each activity of the paging system can be made a separate process, with its own address space. Thus the paging system becomes a set of cooperating sequential processes, running in parallel and asynchronously. Such

systems will be called multi-process paging systems, and this thesis will argue that such systems offer significant advantages in simplicity, modularity, security and expandability over more conventional designs.

The work described in this thesis differs from a multiple process paging system proposed by Hoare [Ho73] in the number of processes used and the function assigned to each. The model developed by Saxena and Bredt [Sa75] is closer to what is described here. However Saxena and Bredt use a multi-level paging system that distinguishes user page faults from page faults caused by system processes, a distinction found unnecessary in the design presented in Chapter 3. These differences and similarities are considered in more detail in section 3.3.

## 1.4 Summary of Thesis

The remainder of this thesis will examine the design and implementation of paging systems for a large computer utility as several cooperating processes. The Multics system will be used as a model of such a computer utility. Multics was chosen because it is typical of large, sophisticated time shared systems and incorporates both of the prerequisite ideas already mentioned: a multi-level, demand paged memory, and processes. Therefore the basic concepts are already present and need not be added.

Currently a major research effort is being made to engineer a security kernel for Multics [Sc75]. Redesigning the paging system contributes to the certification of such a kernel by reducing both the

size and complexity of the code that must be verified. The original impetus for the work described in this thesis was the need for simplifying kernel mechanisms such as paging.

Chapter 2 discusses the basics of paging systems in detail. The objects page control uses to implement a large demand paged virtual memory are examined. Functions which the paging system must provide to the rest of the operating system are listed and discussed.

In Chapter 3 paging systems are classed into three groups based on their organization. User process paging systems, illustrated by Multics, are those where the paging functions are performed in the user's process. System process paging systems utilize special system processes to implement the paging functions. Combination paging systems, using features of both of the other two types, include designs appearing in the literature due to Hoare [Ho73] and Saxena and Bredt [Sa75]. The author's design for a combination multi-process paging system in presented, in which memory allocation is performed in the user's process but other page control functions are done in system processes. The significant advantages of both types of multi-process paging systems are considered at some length.

A test implementation of the design on the Multics system is presented in Chapter 4, concentrating on the difficulties arising in an actual implementation and the insights gained from such an effort. The results of this test implementation are compared with the current implementation to see how well the goals of a multi-process implementation can be realized.

Techniques for exploiting fully the parallelism available in a

multi-process paging system by eliminating global locking strategies are examined in Chapter 5.

Chapter 6 concludes the thesis by summarizing the important results and drawing some final conclusions and observations.

The three appendices present additional information on the implemented multi-process page control described in Chapter 4. Appendix A compares the design to the standard Multics page control. Appendix B lists the components of the implemented design, and Appendix C contains some of the actual PL/1 code from important portions of the design.

# CHAPTER 2

## Basic Objects and Functions of Paging Systems

In Chapter 1 the paging system, or page control, was loosely defined to be those procedures and data bases necessary to resolve page faults and provide the memory allocation task. This chapter will focus on exactly what functions and services page control must provide to the rest of the system and what objects page control must implement in providing these functions. Such a description will help suggest how the parts of page control can best be divided along functional lines into several processes.

Figure 2.1 illustrates the model of a memory system that will be assumed in the remainder of this thesis. The memory system is a hierarchical, multi-level memory consisting of three levels: 1. Primary memory, in which any data referenced by a processor must reside. 2. The paging device, or backing store (which need not be a single device) which acts as a large, high speed buffer between primary and secondary memory. 3. Secondary storage, which provides long term storage of data and programs. For example, in such a system primary memory is often high speed core memory; the paging device is often a drum (or a bulk store device in the case of Multics); and disks and perhaps tape normally

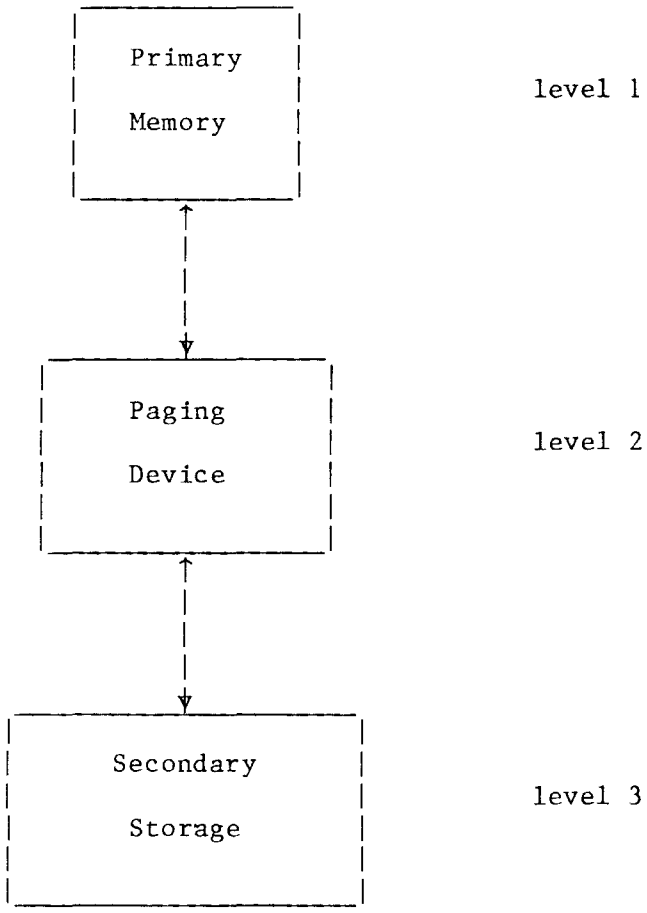



Figure 2.1

Model of a multi-level hierarchical memory system

provide secondary storage.

While the model shown incorporates three levels of memory, more or fewer are possible. The actual number of levels should not be crucial in a well designed system. Indeed, the design presented in Chapter 3 will be seen to adapt easily to a multi-level memory with any number of levels.

Pages are moved from level to level by the paging system. It is assumed that a page may reside in any or all levels of the memory at any given time; however only one copy of the page may exist in each level. If multiple copies of a page do exist in the memory hierarchy, they may not all be identical. The most up to date version of a page will be the copy in primary memory (if there is one), then the paging device copy (if there is one).

## 2.1 Page Control Objects

There are three objects of fundamental importance to page control: pages, the basic allocatable unit of virtual memory; page frames, the corresponding unit of physical memory; and address translation registers, which translate virtual addresses into absolute physical memory addresses.

### 2.1.1 Pages

In paged systems, the address space of a process is divided into units called pages, or sometimes virtual pages. A page is an abstraction of a portion of a process's address space, a set of consecutive virtual

addresses (hence the term "virtual page"). Procedures and data are both broken into pages, although this division into pages is invisible to the programmer.

The number of consecutive virtual addresses (locations) in a page is the page size. The page size is typically fixed at a power of two, and generally ranges from 128 to 4,096. The page size is usually determined by characteristics of the hardware in order to optimize performance of secondary memory. The virtual address space of a process is restricted only by the hardware's limits on the number of pages the process may reference.

### 2.1.2 Page Frames

The physical counterpart of a page is a page frame. Just as the address space of a process is divided into pages, the physical memory in the system is broken into page frames. A page frame is a contiguous area of fixed size in some physical memory device. Each page frame can store a number of bits of information, namely the same number of bits as in a page (which depends upon the page size and the word size).

Page frames are the raw memory resource of the system. The number of page frames is strictly limited by the capacities of the various devices in the memory system. Often it is useful to distinguish among the page frames of each level, hence the terms "paging device page frame" or "core page frame" may be used.

Memory allocation is done by assigning page frames to hold the pages

needed by a process. A process may only reference pages which reside in a primary memory page frame. Since the number of primary memory page frames is quite small (on the order of hundreds) while the number of pages the processes in the system can address is much larger (by at least an order of magnitude) only a fraction of the pages can be in main memory at any time. The purpose of the paging system is to multiplex the page frames among the pages to give the appearance of a much larger primary memory.

The paging system must keep track of the status of each page frame, whether allocated or available, at each level of the memory under its control. While there are many ways to organize the required information, we assume lists are used. There is nothing fundamental about using a list structure for this purpose, the choice is largely for convenience. Thus we assume that page control maintains two lists of page frames for each level of memory it manages (primary or core memory, and the paging device -- secondary storage is assumed to be managed by the file system, see section 2.1.4.). These lists are a "used list" containing those page frames currently allocated, and a "free list" consisting of those page frames not currently allocated. We further identify each list by its level, hence there will be a "core used list" and a "core free list", and corresponding paging device free and used lists. Note page control may want to keep certain information about the page frames on these various lists. For example, for every frame on the core used list, page control will want to record the identity of the page using that frame. We assume the page frames in a list may be ordered in an arbitrary manner. (For example, the lists might be structured as linked lists.) The reason for wishing to order the lists is made clear in section 2.2.2.

These four lists, along with the page tables described in the next section, are the fundamental data bases of page control, for they define the state of the memory.

### 2.1.3 Address Translation Registers

Since processes make references to virtual addresses of the form (page, word) while the physical processors executing the instructions of the process must reference real memory using physical addresses, there must be a mechanism for translating virtual addresses (references to virtual pages) to physical addresses (references to page frames). This is done by associating with each virtual page an address translation register. The address translation register contains the address at which the contents of the virtual page may be found (i.e. the absolute address of the page frame bound to the page). All references to pages are made through the address translation registers. If the page has not been allocated a page frame a special tag indicates the fact and causes a special hardware fault when a reference is made to the address translation register.

The address translation registers for all the pages in the address space of a process may be collected together into a page table. Typically the virtual pages in the address space are identified by a number: 0, 1, ..., n. The page table then is an array of address translation registers; the ith page table entry is the address translation register for virtual page i. Because the address translation registers are grouped into a page

table, they are often also called page table words, since each is essentially a word in the page table. Hence we will use the term page table word to refer to these page address translation registers (and to distinguish them from address translation registers used for segments; see the following section). The page table may be contained in special hardware registers, or reside in memory as any other data. Of course, the physical processor must know the physical address of the page table. If the page table is maintained in memory, a special register, the page table base register, indicates where. This translation mechanism for paging is illustrated in Figure 2.2.

Besides containing the physical address of the page, the page table word often contains some additional items, such as whether the page has been referenced recently or modified. The reason for recording these facts is usually to provide information to various page control algorithms. More will be said about the function of such additional information below.

### 2.1.4 Segments and the File system

At this point a brief digression is in order. Although this thesis is concerned with paging systems and deals with pages as the basic component of a process's address space, it is necessary to also consider a higher level organization of the address space, namely segmentation. Segmentation has a profound influence on a paged system.

Until now the address space of a process has been treated as strictly

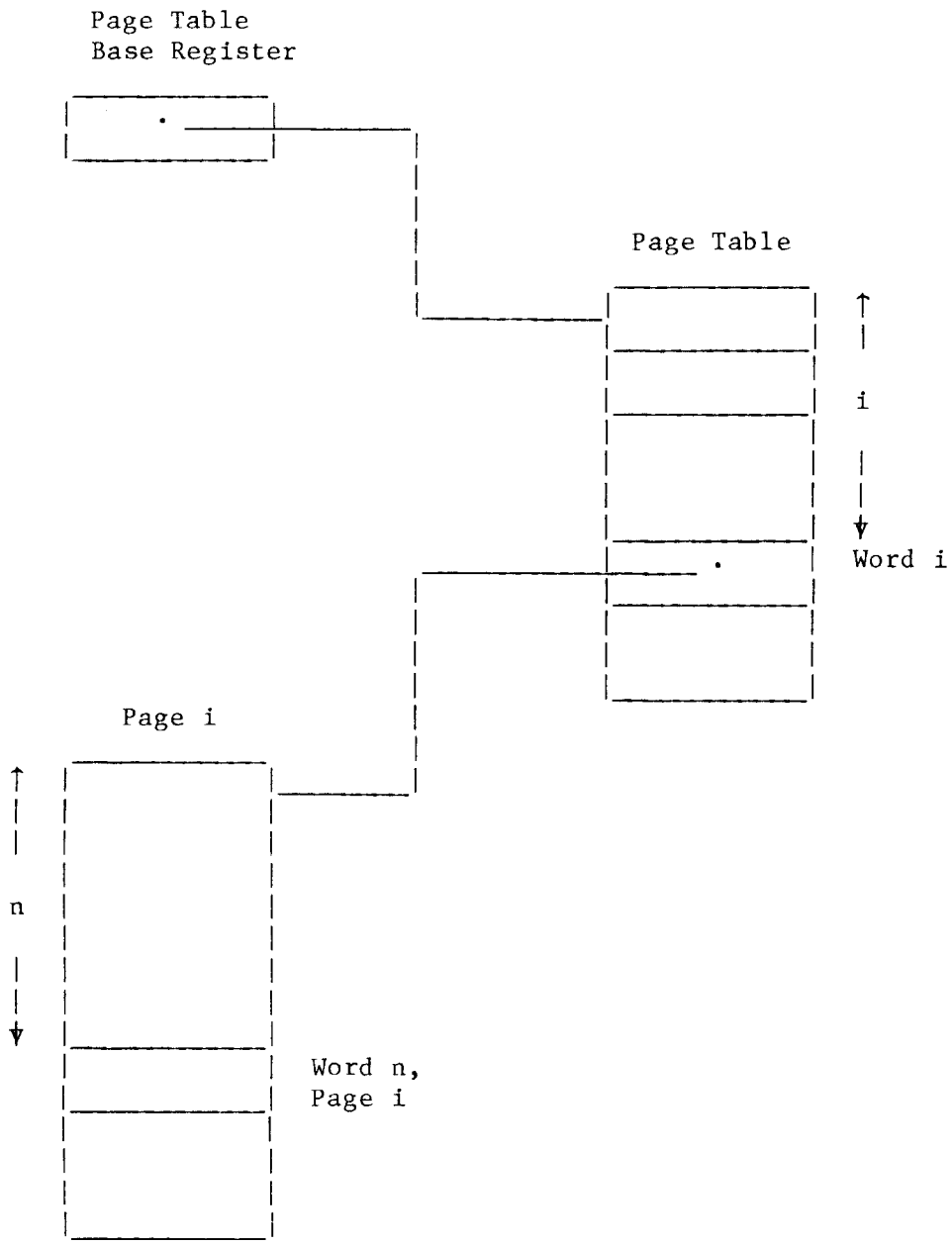



Figure 2.2

Translation of virtual address page i, word n

linear, a one dimensional array of words. In Multics and other segmented systems this is not the case. The address space in a segmented system is two dimensional, containing multiple segments, each of which is itself a linear address space. Thus a virtual address in a segmented system consists of a segment number and a word number (offset) within the segment. Each segment is paged, so the offset within the segment is in two parts, as before: a page number and a word within the page.

Instead of having a single page table, the address space of the process is now defined by a page table for each segment. There must be a page table base register for each page table; these will be called segment descriptor words and collected into a descriptor segment. The jth segment descriptor word contains the absolute address of the page table for segment j. The descriptor segment of a process completely defines the address space of the process. The physical processor executing the instructions of the process must know the location of the descriptor segment for that process. A register called the descriptor segment base register is used for this purpose. The translation of a virtual address in a segmented, paged memory is shown in Figure 2.3.

Segments may be shared, i.e. in the address space of more than one process. In this case there will be a segment descriptor word for the shared segment in the descriptor segment of each process sharing the segment. These segment descriptor words will all point to the same page table.

While the paging system bears the responsibility for maintaining the page table words, the job of assigning a page table to a segment will be assigned to a different module, the segment manager. Since the number of

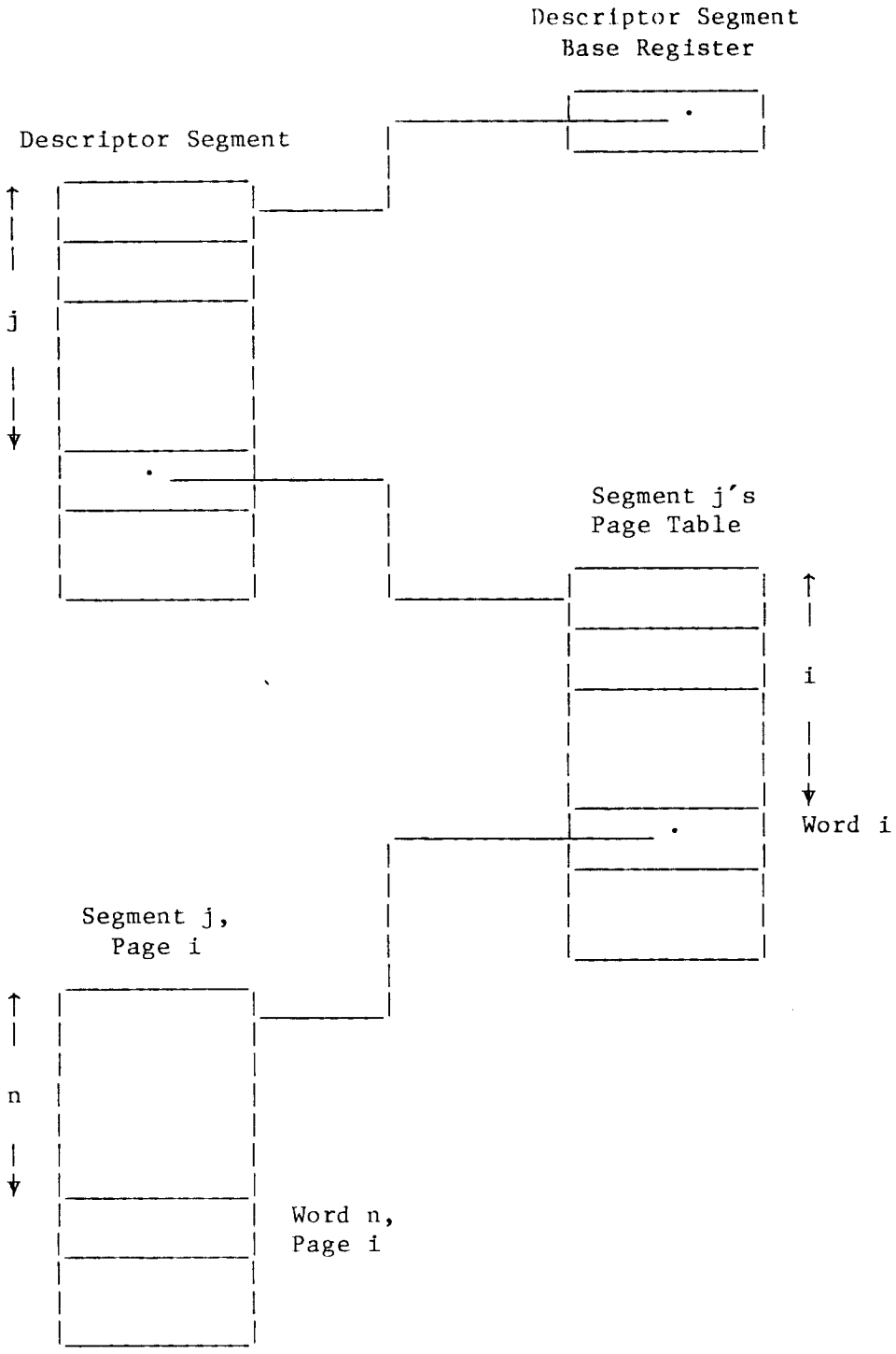



Figure 2.3

Virtual address translation with segmentation

segments in a process's address space is unlimited for most practical purposes, a page table cannot be given to every segment. Instead, the available page tables are multiplexed, just as page frames are multiplexed among a large number of virtual pages. That is, segmentation implies dynamic page table word allocation. Allocation of page tables to segments is a task very similar to allocating page frames to pages. This job is performed by the segment manager and will not be discussed further here. Activating a segment (corresponding roughly to opening a file in many systems) results in the segment being assigned a page table.

The paging system can deal only with segments that are active, i.e. have page tables. Deactivated segments, those not assigned page tables, are manipulated by the segment manager and the file system.

Thus the page tables, though indispensable to the paging system, are not completely implemented by the paging system. Rather the task is shared with the segment manager (or segment control, as it is often called). And although segments per se are not really page control objects, page control is aware of their existence and has some knowledge of their implementation. As a consequence, there is interaction between segment control and page control. This interaction is undesirable as it complicates both segment control and page control, and we would like to minimize the interface between segment control and page control. This interface will be examined in detail at a later time. (1)

Similarly, page control interacts with the file system and knows

---

(1) Research in progress at the Computer Systems Research Division is attempting to eliminate from page control this knowledge of segment control and the implementation of segments.

about the file system's organization. Such knowledge complicates page control, and minimizing the influence of the file system on page control is highly desirable. By the file system we mean the operating system modules which manage the permanent storage of segments on secondary memory. The file system is responsible for knowing where a segment is stored in secondary memory so that the paging system may bring the segment's pages into primary memory when needed. Secondary storage page frames, or "records", are allocated to segments by the file system when the segment is created. Thus, the file system must remember the location of each page, and a "file map" analogous to a page table is kept for each segment to retain this information. The file map itself can be stored in the file system.

The structure of the file system may vary widely; however we will not be concerned here with the specific organization. The file system may be hierarchical as in Multics or flat (one-level).

## 2.2 Page Control Functions

Having examined the basic objects the paging system manipulates we turn to the operations that page control must perform on these objects. The most important job of page control is allocating memory, that is, assigning free page frames to hold pages. When all available memory has been allocated, memory deallocation must occur to enable reuse of page frames. Memory deallocation removes pages from page frames thereby freeing the page frame for further use. Note that in a multi-level memory

system a page may be allocated memory in one, several, or none of the levels.

Hence the two major functions of page control are:

1. Memory allocation
2. Memory deallocation

Two other minor functions that a paging system may optionally provide are:

1. Reconfiguration
2. Wiring or Locking

The following sections will consider all four of these in turn.

### 2.2.1 Memory Allocation

Memory allocation is the primary task of the paging system. Recall that a processor may only reference pages which are allocated main memory page frames. A reference to a page not allocated a main memory page frame causes a page fault. Assuming a free list is kept, as mentioned in section 2.1.2, the steps involved in allocating memory and thereby resolving the page fault are the following:

1. A reference is made to the page, whose page table word contains a special tag, causing a hardware fault which results in the invocation of the paging system's main memory allocator.

2. A free page frame is obtained from the core free list.

3. The identities of the page to be read in and the frame the page is read from are saved in the collection of information associated with the

main memory page frame. (This information is needed when deallocation occurs.)

4. A read operation is performed to copy the contents of the page into the main memory page frame.

5. The absolute address of the page frame is placed in the page's page table word, replacing the special fault tag. (Note the fault tag must remain until the read operation is completed.)

Control may now be returned to the process that made the reference to the page.

An important complication arises in a multiprocessing environment with sharing. Care must be taken so that while the sequence of steps described above is in progress, other processes sharing the page are prohibited from repeating the steps. That is, two processes may not allocate page frames for the same page simultaneously. This would lead to several possibly inconsistent copies of the same page. There must be some inhibiting mechanism which prevents a process from beginning the allocation procedure for a page if some other process has already started the allocation algorithm for that page.

There are many ways of implementing such a mechanism. One is to permit only a single page to be involved in the allocation procedure at any given moment. For example, the allocation code could employ a lock, which any process executing the allocation algorithm must set. Since there may be a considerable delay involved during the read operation, this scheme may result in an impractically inefficient paging system. A per page mechanism, rather than a global mechanism which inhibits all allocation, seems desirable. There is much more to be said on this topic;

the mechanism used to prevent multiple allocations for a single page is very influential in determining the efficiency of the overall page control design. A closer examination of this issue is postponed until Chapter 5.

Memory allocation must be performed at each level in the memory system. Thus memory allocation must also occur for the paging device. The only difference from main memory allocation is the manner in which allocation is initiated. Main memory allocation takes place in response to a page fault; paging device memory allocation is done in response to an explicit request made by the main memory deallocation algorithm as explained in the next section. Otherwise, the steps in allocating paging device memory to a page are identical to those for allocating main memory:

1. A request is made to the paging device allocator for a paging device page frame.

2. A free paging device page frame is chosen from the paging device free list.

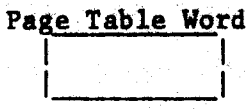
3. The identity of the page is stored in the collection of information associated with the paging device page frame.

4. The contents of the page are copied into the page frame.

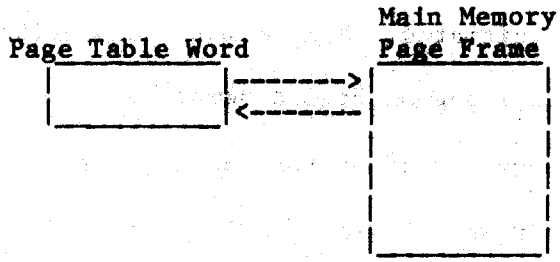
5. If the page has a main memory page frame allocated, the identity of the paging device page frame is saved in the information associated with the main memory page frame, and vice versa (see Figure 2.4). Otherwise, the identity of the paging device page frame is placed in the page's page table word so that when a fault occurs the location of the page on the paging device is known.

As was the case with main memory allocation, once allocation of a paging device page frame to a page has begun, the system must insure some

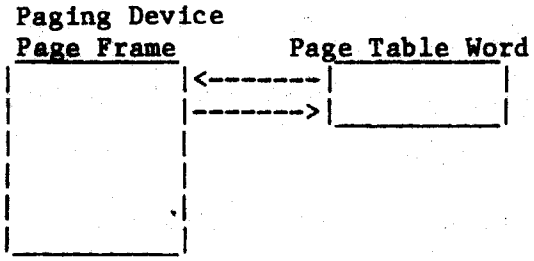
Case 1: Unallocated page




Case 2: Page allocated a main memory page frame




Case 3: Page allocated a paging device page frame




Case 4: Page allocated both a main memory page frame and a paging device page frame

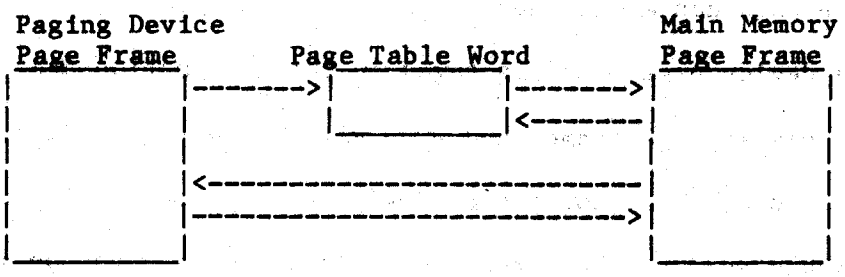



Figure 2.4

Allocating memory to pages

other process does not duplicate the effort. The same mechanism used to prohibit multiple main memory allocations may be employed.

Memory allocation at the final level of the memory system is the duty of the file system, since the file system bears the responsibility for permanent storage of segments.

### 2.2.2 Memory Deallocation

The second step in allocating main memory listed in the preceeding section is to obtain a free page frame from the core free list. This list can be maintained only by deallocating main memory; i.e. reversing the steps of the allocation algorithm and thereby freeing page frames. This operation is commonly termed "page replacement" in paged systems. Page replacement, or memory deallocation, is nothing more than removing pages from the page frames in which they reside.

The steps taken in deallocating a main memory page frame from its page are summarized below:

1. A used page frame is selected from the core used list.

2. The page contained in the page frame (which can be determined by looking at the information associated with the page frame -- see step 3 in the allocation procedure) is copied to some other page frame in the memory hierarchy (more on this shortly).

3. The physical address of the page frame stored in the page table is replaced by the address of the page frame copied to in step 2, and the fault tag is set.

4. The page frame is added to the main memory free list. (For security reasons, the contents of the page frame should be cleared to all zeroes.)

Several comments are necessary to explain these steps further. First, nothing has been said about how the deallocation algorithm is started. The allocation process might note when performing step 2 that the free list was empty and thus issue a call to the deallocation routine. This has the undesirable effect of delaying the allocation. The approach taken in the design presented in Chapter 3 is to maintain the free list at some minimum size; whenever the supply of free page frames is depleted below the system determined limit, deallocation begins until the free list is sufficiently replenished. There is, of course, a significant tradeoff involved here: time spent in allocating memory versus effective memory utilization. Page frames on the free list represent unused physical memory. It is possible to utilize memory completely by allowing the free list to become or remain empty. But then allocating memory is slowed due to the necessity of first deallocating some other page frame so that a page frame is free. Although a delay in allocating memory to any one process should not lower throughput in a multiprogrammed system, two costs are involved: a process that presumably already has pages in memory is prevented from running, and response time for any one process is lengthened.

Second, nothing has been said about the criteria to be used in choosing from the used list the page frame that is to be replaced. The method for making this decision is commonly called the "page replacement algorithm" and usually involves usage characteristics of the page

contained in each page frame. For example, the First in, First out (FIFO) page replacement strategy chooses whichever page frame has been allocated to a page for the longest time. Note this implies it is possible to order the page frames by the length of time they have been allocated. One way to do this alluded to earlier is to maintain the used list as a linked list of page frames; the head of the list being the page frame in use for the longest period of time. Newly allocated page frames are added at the end of the list. We will not be concerned with the details of specific page replacement algorithms; the discussion of paging systems here is intended to be general enough to permit almost any page replacement algorithm. It is worth noting however that some algorithms require special information be kept on each page. For example, a "used" bit is often associated with each page. This bit is set by the hardware when a reference is made to the page. The replacement algorithm may examine the bit, and reset the bit, in deciding what page should be deallocated. The details of one such scheme are given by Corbato [Co69].

A third comment with respect to memory deallocation pertains to copying the contents of the page to some other page frame in the hierarchy. There are two points of interest: what other page frame to use, and when the copying is necessary.

The question of where the page is to go when ejected from main memory is answered by looking in the data associated with the page frame. Recall that step 3 of the main memory allocation algorithm given above remembers the page frame a page is read from when allocated main memory. If the page was read from a paging device page frame, it may be written back to that same frame by an appropriate output routine. Otherwise, the page was

read from disk, and the paging device memory allocation mechanism is invoked (as discussed in the previous section) to obtain a paging device page frame to allocate to the page and serve as the destination of the page. Under certain circumstances, or if the paging device itself is not part of the current memory configuration, the page's contents may instead be returned to their permanent file system location.

The copying is necessary only under two circumstances: 1. The page has not yet been written into the paging device page frame. 2. The page has been altered by a write operation, and hence the copy in main memory differs from the paging device copy. The first situation is readily recognized; to aid in detecting the second situation many paged systems include special hardware which associates a "modified" bit with each page. This is similar to the used bit mentioned in conjunction with page replacement, but the modified bit is set only when a write reference is made to the page, e.g. a store instruction. This bit is examined by the deallocation algorithm; if it has been set then the page has been modified while in main memory and must be copied.

Deallocation of paging device memory is analogous. The steps involved are as listed above for deallocating a page frame from its page. The comments apply equally well with only the following alterations:

Utilization of memory on the paging device is less critical than with main memory. This is because there is assumed to be a much larger amount of memory on the paging device. Hence paging device page frames are a less critical resource; therefore it is feasible to maintain a larger number of page frames on the paging device free list than might be the case for main memory.

Used and modified flags may also be associated with each paging device page frame. The used flag may provide information to the paging device page replacement algorithm for determining which paging device page frame should next be deallocated. The modified flag determines when copying the contents of a page is necessary at deallocation time.

### 2.2.3 Memory Reconfiguration

The memory configuration is defined by the page frames available to page control for allocation. Memory reconfiguration consists of dynamically adding or removing page frames to the supply available to page control. To add memory to the system dynamically, the page frames of the memory unit must be added to the pool of page frames controlled by the paging system. The inverse operation of removing memory is slightly more complex. The page frames of the device being removed must be freed before they may be removed from the memory configuration.

Reconfiguration is not, strictly speaking, a page control function. It is included here because page control must cooperate in reconfiguring memory, and any paging system should be designed with an awareness of the problems of reconfiguration. Thus to assist in removing memory, a removing flag might be associated with each page frame. This flag is turned on by the reconfiguration algorithm. The allocation algorithm should be designed to ignore any page frames on the free list with the removing flag on. This prevents allocating to a page a page frame that will only have to be deallocated shortly.

Newly added memory may be treated simply as free page frames and added to the free list for future use. Schell [Sc71] provides an extensive examination of dynamic reconfiguration. The desire to perform dynamic reconfiguration can complicate other page control functions severely, as the next section will demonstrate.

### 2.2.4 Memory Wiring

A useful function for the paging system to provide is that of "wiring" or "locking" memory. A "wired" page is simply a page that must always be allocated a page frame, thereby always remaining referenceable by a physical processor. There is a second, more restricted type of wiring which will be called "absolute wiring"; an "absolute wired" (or "abs wired") page not only must be allocated a page frame at all times, but the same page frame at all times. This means that the absolute physical address of the page will not be changed.

Some system functions must be wired, at least in part, in order to operate properly. The pages of page control and page control's data bases are an excellent example of this. In order to avoid an infinite recursive loop of repeatedly taking page faults while handling a page fault, at least a portion of page control's procedures and data must be wired.

Absolute wiring is necessary only if absolute physical addresses are used by parts of the system. The most likely place for this to occur is in the input/output programs. Channel or i/o programs may require absolute memory addresses; if this is the case pages used as buffers for

doing i/o to terminals, etc., once allocated a particular page frame, must remain there. The only alternative, to somehow keep track of all the instructions that use the absolute address and alter these instructions every time the page is allocated a different page frame, is generally impractical.

Providing for wired pages is fairly straightforward. An additional flag may be associated with each page frame. When a page must be wired, it is allocated a page frame and the wired flag is turned on, indicating the page frame may not be deallocated. In searching for a page frame to replace, the replacement algorithm must skip over any page frame whose wired flag is on. A page may be unwired at any time if it no longer must remain referenceable, by merely turning off the wired flag.

Absolute wiring may be provided in a similar fashion. An extra complication arises if in setting up a buffer a contiguous area of memory greater than the size of a page is required. In such a case the paging system must contrive to allocate some number of page frames which have consecutive absolute physical addresses. It may not always be possible to guarantee this can be accomplished.

The chief difficulties involved in both wiring and abs wiring virtual pages are due to two sources: sharing of virtual pages, and reconfiguration. Since the same virtual page may be in the address space of several processes, two or more processes may desire that a particluar page be wired. In such a case, a simple flag is inadequate; a counter of the number of processes wiring the page is needed instead. Where security is an issue, additional mechanisms are needed to insure pages may be unwired only by a process that previously wired them.

Reconfiguration poses a more difficult problem. Adding memory, of course, presents no difficulty. But consider what happens if an attempt is made to remove from the memory configuration page frames which have been wired or absolute wired. The reconfiguration must fail if an absolute wired page is encountered, for by definition its physical address cannot be changed. Simple wired pages can be handled, though not without some awkwardness. Remember a wired page must remain referenceable (allocated a page frame) at all times. Thus the page may be moved by allocating a new page frame, copying the contents of the page into that new page frame (meanwhile the page is still allocated the page frame being deconfigured), and then replacing the address in the page table word of the page with the physical address of the new page frame. Additional complications occur if the virtual page is modified during the copy operation. This problem is discussed fully by Schell [Sc71].

## 2.3 Summary

The job of page control is to implement a large virtual memory for processes by multiplexing the limited amount of physical memory. Page control deals with four objects: Pages are the basic unit of a process's address space. Page frames are the basic unit of allocatable physical memory. Page table words are used to map pages into page frames by translating virtual addresses referenced by processes into absolute physical addresses usable by hardware processors. Segments are logical units of information, either programs or data, consisting of one or more

virtual pages. Each segment has a page table containing all the page table words for the virtual pages of the segment.

The chief functions of a paging system were seen to be memory allocation (assigning a page frame to hold the contents of a referenced page) and memory deallocation (removing the contents of a page from a page frame, freeing the page frame for allocation). Other functions related to page control discussed were memory reconfiguration (changing the pool of page frames available to page control), and memory wiring (prohibiting the breaking of a page frame-page binding).

# CHAPTER 3

## Designs for Paging Systems

Now that the underlying concepts of paging systems have been introduced and the functions required of such systems examined, we turn to the question of how to structure a paging system for a large computer utility. The Multics system will be used as the basis for the general computer system model for which such a design is intended.

Contemporary paging systems such as the Multics page control have not been implemented taking full advantage of the process concept even though the operating system itself implements and makes extensive use of processes. Rather each user process performs the functions of page control, using shared supervisor code and data.

The first part of this chapter will present a method for classifying paging systems based on whether user or system processes implement the paging system. Multics will be used as an example of a paging system where the paging functions are performed by the user's own process. A simple change to convert the Multics design to one using a system process to perform the page control operations is then considered. Next a design splitting the paging functions among several processes is presented. This design was actually implemented and tested on the Multics system.

(Chapter 4 discusses the details of this implementation.) Two other similar designs appearing in the literature are contrasted to the proposed design. The advantages of these multi-process paging systems are demonstrated by comparisons with the current Multics page control.

## 3.1 Paging System Structures

We will divide paging systems into three broad categories depending upon the answer to the following question: Where, i.e. in what process, are the paging functions performed? The categories are:

1. User-process paging systems, in which the page control functions described in Chapter 2 are performed by the users' processes.

2. System-process paging systems, utilizing special system processes whose exclusive job is to carry out page control operations exclusively.

3. Combination paging systems, where some page control operations are done in the users' processes, others by system processes.

A further division of paging systems can be made based on how many processes implement the paging system. (Clearly this is not meaningful for user process paging systems, since all the processes in such a system implement the paging functions.) Thus we might consider system process paging systems or combination paging systems utilizing only a single system process as opposed to multiple processes. As will be seen, however, the advantages of multiple processes are so compelling that once the concept of using a system process to perform paging functions is accepted, multiple processes seem a natural and obvious extension.

In examining each of the different organizations for paging systems, we will be particularly interested in the solution the design uses for two crucial problems inherent in a multi-process environment allowing sharing of pages among users. These two problems are data base contention and page fault contention.

By data base contention we mean the interference caused by two or more processes attempting to access a common data base simultaneously. Hence data base contention is a direct consequence of multi-processing. Data base contention is only a problem, of course, when the data base may be written as well as read. When a process may alter a data base, unless all alterations can be performed in a single, uninterruptible operation, there is the danger that another process may find the data base in an inconsistent or outdated state. This is not a problem unique to paging systems, arising here due to the fact a central accounting of all memory resources must be kept by page control. As a simple but important example, if two processes wish to obtain free page frames simultaneously, the paging system must insure the same page frame is not allocated to both.

Thus we wish to know what mechanisms the paging system design offers to provide exclusive access to essential data bases. Ideally the mechanism should be easy to understand and use as well as guaranteeing data base integrity and prevention of system deadlock. Usually some form of semaphore or lock is involved.

Page fault contention, or more simply page contention, is caused by the sharing of information among users in a multi-processing environment. When users share information, the pages containing that information are in

the address space of each user's process, and may be faulted on by any referencing process. If users were not allowed to share pages, e.g. if users executing the same program were always given their own copy of the program, page contention would be non-existent. By page contention, we mean the problem already mentioned in section 2.2.1. That is, two processes may not be allowed to allocate a page frame to the same page simultaneously, or multiple copies of the page in primary memory may result.

In some sense page contention is really data base contention in a different guise, for after all a page may be considered a data base. We differentiate between page contention and data base contention because separate mechanisms are normally employed to resolve each. While conceivably careful data base design can minimize data base contention, page contention can not be avoided as long as the time required to read or write a page between memory levels is long relative to instruction speeds.

The following sections present several designs for paging systems. Attention will be given to the techniques inherent to each for dealing with page contention and data base contention.

## 3.2 Multics' User Process Page Control

We begin our investigation of paging system designs with a typical, contemporary paging system, namely the Multics page control (as it existed in fall, 1975). The procedures of Multics page control execute in the users' processes, qualifying it as a user process paging system under the

definition of the previous section.

### 3.2.1 The Current Multics Page Control

A process taking a page fault in the Multics system begins all the required paging functions at the time of the fault. Thus allocation and deallocation of page frames in both levels of the memory must be done at page fault time. The complexity that this results in is well illustrated by Figure 3.1, which represents diagrammatically the Multics page control. The diagram is necessarily at a rather high level, omitting much detail. The boxes represent program modules (procedures) carrying out specified functions; the solid arrows depict procedure calls and the dashed arrows indicate interprocess messages. The following paragraphs describe the sequence of events represented by Figure 3.1 happening after a page fault.

When the page fault code is invoked, the first thing done is to run the paging device page removal algorithm, as depicted by the call to the routine labeled "get free pd record" in Figure 3.1. This procedure checks to see if there are ten free paging device page frames. If there are less than ten, enough paging device page frames are selected, one at a time, to increase the number to ten, and the necessary i/o to remove their pages from the paging device is begun.

At this point two complications arise. The first is due to hardware limitations of the Multics system. It is not possible to perform read or write operations directly between the paging device and the disks in Multics, only between main memory and the paging device or between main

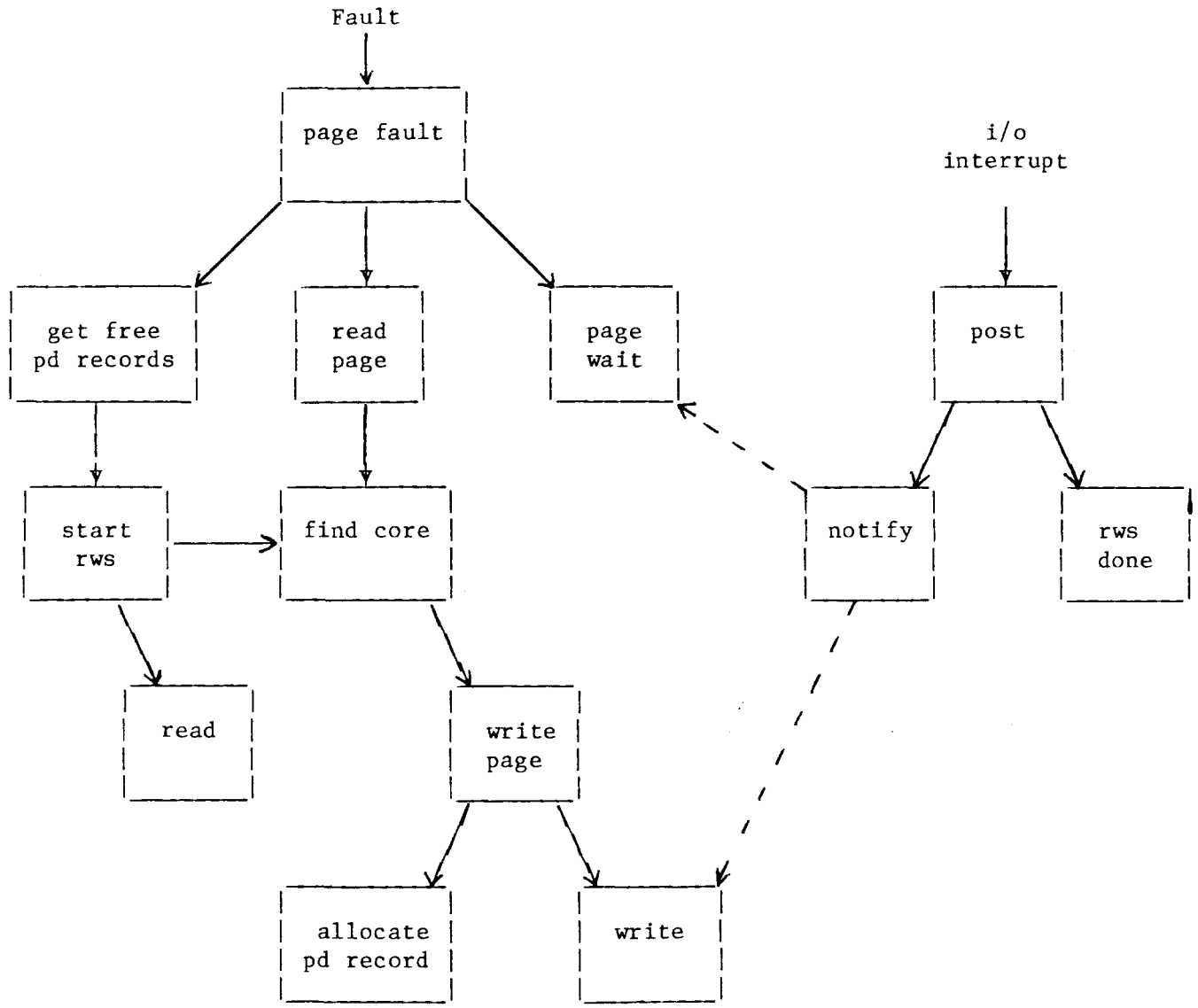



Figure 3.1 -- Multics page control

memory and the disks. Thus, the operation of writing a page from the paging device to the disks must be done in two steps: first a read operation, reading the page from the paging device to main memory; second, a write operation, writing the page from main memory to disk. This two step operation, a read followed by a write, is called a "read write sequence", or "rws". Note that performing a read write sequence requires a free main memory page frame. This is indicated in the diagram by the call made by the module "start rws" to the "find core" routine.

The second complication results from the relatively long time required to perform a read or write operation on a page. To require that the faulting process wait until the i/o operations it may start as part of read write sequences are completed would intolerably delay the faulting process, causing poor response. Thus the i/o necessary to evict pages from the paging device is not waited on, but only started. When the completion of this i/o is signalled via a hardware interrupt, whatever process is currently executing must deal with the interrupt. Thus the task of deallocating paging device page frames, though begun by the process taking the page fault, is finished by whichever process happens to be running at the completion of the disk write operation.

Returning to the discussion of Figure 3.1, we are now ready to resolve the page fault by calling the procedure named "read page", which must first allocate main memory space. This is done by a call to "find core" which is the main memory page replacement algorithm. When a free page frame has been created by evicting a page, it is returned to read page, which then may start a read operation to copy the contents of the faulted-on page into main memory. The faulting process must then wait

until the read is completed, as indicated by the call to the procedure "page wait". The completion is signalled via a hardware interrupt, which is converted to a software notify.

Multics uses a single semaphore, called the global page table lock, to solve the data base contention problem. This lock must be set by a process before it may begin processing a page fault. The lock is released just before the process blocks itself by calling "page wait". In between these times, another process attempting to resolve a page fault must wait until the lock is released.

Waiting on the lock is done by repeatedly trying to set the lock until one succeeds in doing so. This "busy" waiting has two major implications: 1. A process may not block itself, giving up the processor, while it has the page table lock set. If this were done, all page control functions would be prevented until the process were awakened and run again. 2. For efficiency reasons, the time spent with the lock set should be minimized, as this in turn minimizes the interference among processes due to the lock which results in wasted processor time.

Measurements show that when running the standard Multics system in a configuration with two processors, under a moderate to heavy load the processor time spent looping while waiting to lock the global page table lock can amount to 10% of the total system processor time. In certain extreme conditions this overhead can go as high as 20%. This effect would be even worse in a system with three or more processors. Hence the global locking strategy can have a severe impact on system performance. (1)

---

(1) A recent experiment has shown that abandoning the processor rather than

The global page table lock is not used to protect against page contention. To do so would prevent any process from resolving a page fault until all read and write operations caused by a previous page fault had completed (including read write sequences). Instead, a per page lock (implemented as a bit in the page table word of each page) is used. This per page lock is set whenever i/o is begun on a page (which can only happen with the global page table lock set) and remains set until the i/o completes. Thus a process faulting on a locked page, even though it gains control of the global page table lock, cannot start i/o to bring in the page (or to throw it out). The process must wait until the lock is released. Hence the per page lock protects the page while in transition.

### 3.2.2 A System Process Page Control Based on Multics

To introduce how a paging system implemented as a system process might work and to see some of the potential advantages of such a design, consider the following simple yet radical change to the design just described: When a page fault occurs, instead of having the user process execute the programs to resolve the fault, simply send a message to a system page control process, and wait for a return message saying the desired page has been bound to a page frame in main memory. Nothing else is changed; the algorithms described previously and illustrated by Figure 3.1 have merely been made a separate process. Essentially what has

---

looping on the lock will increase the performance of a three processor system. This change may be incorporated into the system.

happened is that a page fault has been transformed from a call to the page fault procedure to an interprocess message to the page control process.

There are disadvantages to this design, mainly in terms of efficiency. The time required to resolve a page fault is increased by the length of time required to send the message to the page control process and to schedule the page control process.

What do we gain? First, the page control process has its own address space and execution point. A separate address space enables removal of all the paging algorithms and data bases from the user's address space. The execution point, as we shall see, allows parallel execution of the page control process.

A second benefit is guaranteed service. Since the messages to the page control process (i.e. the page faults) can be ordered, we can serve the page faults in the order they occur. There is nothing in Multics currently to prevent an unlucky process from always being locked out of the page fault handling code by competing processes who always manage to lock the global page table lock first. (That this actually ever happens is a very remote possibility, but important if guaranteed service is a system goal.)

A third benefit is the elimination of the global page table lock. Since only a single process, the page control process, may access the paging system data bases, data base contention is impossible. This benefit seems illusory because the single process has replaced the global lock, and the overall effect is the same -- only one page fault may be processed at a time; in fact only one page control function may be performed at a time, since there is only one process (and hence one

execution point) to perform them. However, replacing a lock with a single process is not only conceptually cleaner but also easier to understand and show correct.

The important thing here is the fact that the process has an independent execution point as well as a separate address space. Once we realize this fact, the question arises as to why not change the algorithm of Figure 3.1 to take advantage of this execution point? Why continue to deallocate page frames only when resolving a page fault? Since the page control process knows page frames will be needed, why not have him execute the page replacement algorithm between page faults, when he would otherwise be idle?

This concept of allowing independent parallel processing by a system process performing page control functions, leads us directly to the multi-process combination paging systems discussed in the next sections.

## 3.3 Multi-process Combination Paging Systems

Expanding on the possibilities suggested by the single system process design presented in the preceeding section, three multi-process combination paging systems are examined here. In each of these, the necessary allocation of main memory page frames to pages is performed by the faulting process. Deallocation, however, is done by the special system processes. Thus these paging systems classify as combination paging systems as defined in section 3.1. Additionally, each design uses multiple processes to implement the system performed paging functions,

hence the term multi-process combination paging systems. The number and organization of the system paging processes are what distinguish the three designs. The first is due to the author and has been implemented on Multics (see Chapter 4); the other two designs have appeared in the literature.

### 3.3.1 A Two Process Paging System

In Chapter 2 it was noted that the work of the paging system can be described largely as allocating and deallocating page frames to and from pages. Allocating a page frame to a page is a relatively simple task that a process can do for itself, since there is no need for parallelism — the process cannot continue until the page fault is resolved. In demand paged systems, allocation is performed only upon actual reference to a page, because it is impossible in general to predict which pages in its address space a process may reference.

Deallocating page frames (and thereby creating free page frames) is a more complex task involving decision making, namely choosing the page that is to be replaced. Deallocation, unlike allocation, may be done at any time.

In particular, page frames may be freed in advance, maintaining a pool of free page frames from which page frames are selected as needed. Replenishing the supply of free page frames may be done whenever convenient. The job of deallocating page frames may be assigned to a system process, distinct from user processes. Note this allows us to take

advantage of the parallelism offered by a process. This completely removes the page replacement function from the user process. There are several immediately obvious advantages to such a strategy: 1. Page faults may be resolved faster, since deallocation is no longer done at page fault time. 2. The page fault algorithm is simpler. 3. The procedures and data involved in doing the deallocation may be removed from the address space of the user process. These and other benefits of such a decision will be discussed fully later.

Since the memory model assumed here (Figure 2.1) incorporates two levels of memory managed by the paging system, two system processes will be used in the multi-process page control suggested here. One will be assigned the task of deallocating page frames for each level in the memory. The three parts of the resulting design (handling page faults in the faulting process being the third) are discussed in turn.

The Core Manager Process

The special system process assigned the task of deallocating main memory page frames will be called the core manager process. The algorithm followed by the core manager is depicted in Figure 3.2. As long as the number of free page frames in the pool available for allocation is less than some system determined value, the core manager keeps deallocating page frames. First, the page replacement algorithm is invoked to decide which page frame is to be deallocated. Note this is strictly a policy decision. Once a page frame has been selected, it can be freed by writing the page out of main memory and changing the page table word for the page appropriately. When the write operation is completed, the newly freed

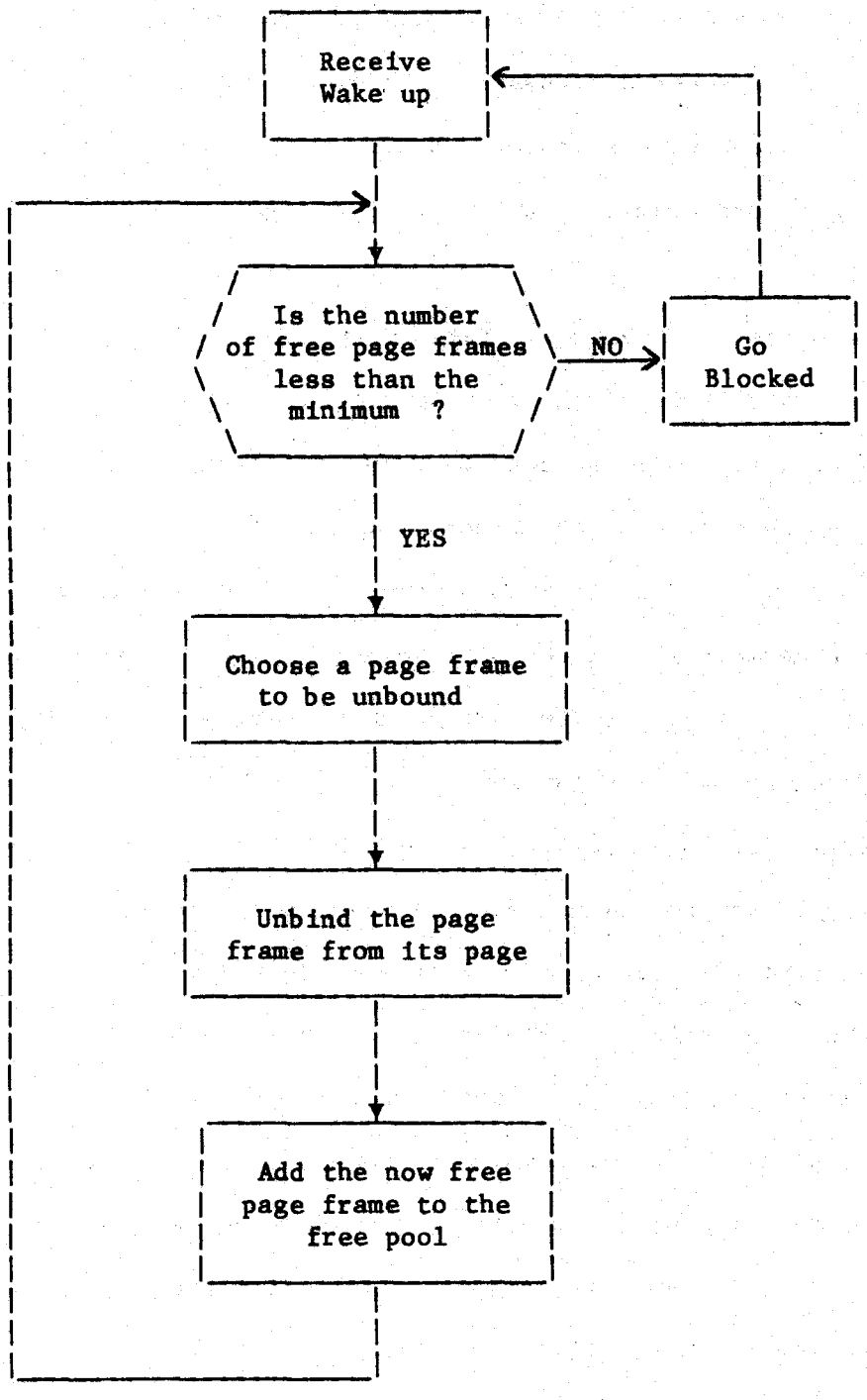



Figure 3.2

Algorithm of the core manager process

page frame may be made available for allocation to some process requesting a page frame. This sequence of steps may be repeated until the supply of free page frames reaches some system determined value, at which time the core manager process blocks itself. Notice that processes may be requesting free page frames from the free pool even while the core manager is executing.

There must be some means of starting up the core manager process. One way to do this is to simply wake up the core manager periodically. An alternative strategy which adjusts to varying demands for free page frames is to wake up the core manager process whenever the pool of free page frames becomes low. This requires interprocess communication, for the process which notices the number of free page frames is down must wake up the core manager. That is, the routine which allocates free page frames must follow the algorithm shown in Figure 3.3. If there is at least one free page frame, it is immediately allocated to the caller. If the remaining supply of free page frames is under a system defined minimum, a wakeup is sent to the core manager process. However, if there are no page frames in the free pool, the allocation code must do one of two things: 1. Report failure to its caller, who must try again later, or 2. Block the calling process until the core manager process signals that the supply has been increased. Of course, in either case the core manager must be awakened to start replenishing the free pool. The latter approach is chosen here because it results in an allocation strategy which always succeeds in the eyes of the caller, i.e. always returns a free page frame. This simplifies the code in the calling procedure. Indeed the caller will never know what happened, except perhaps that it took longer for the

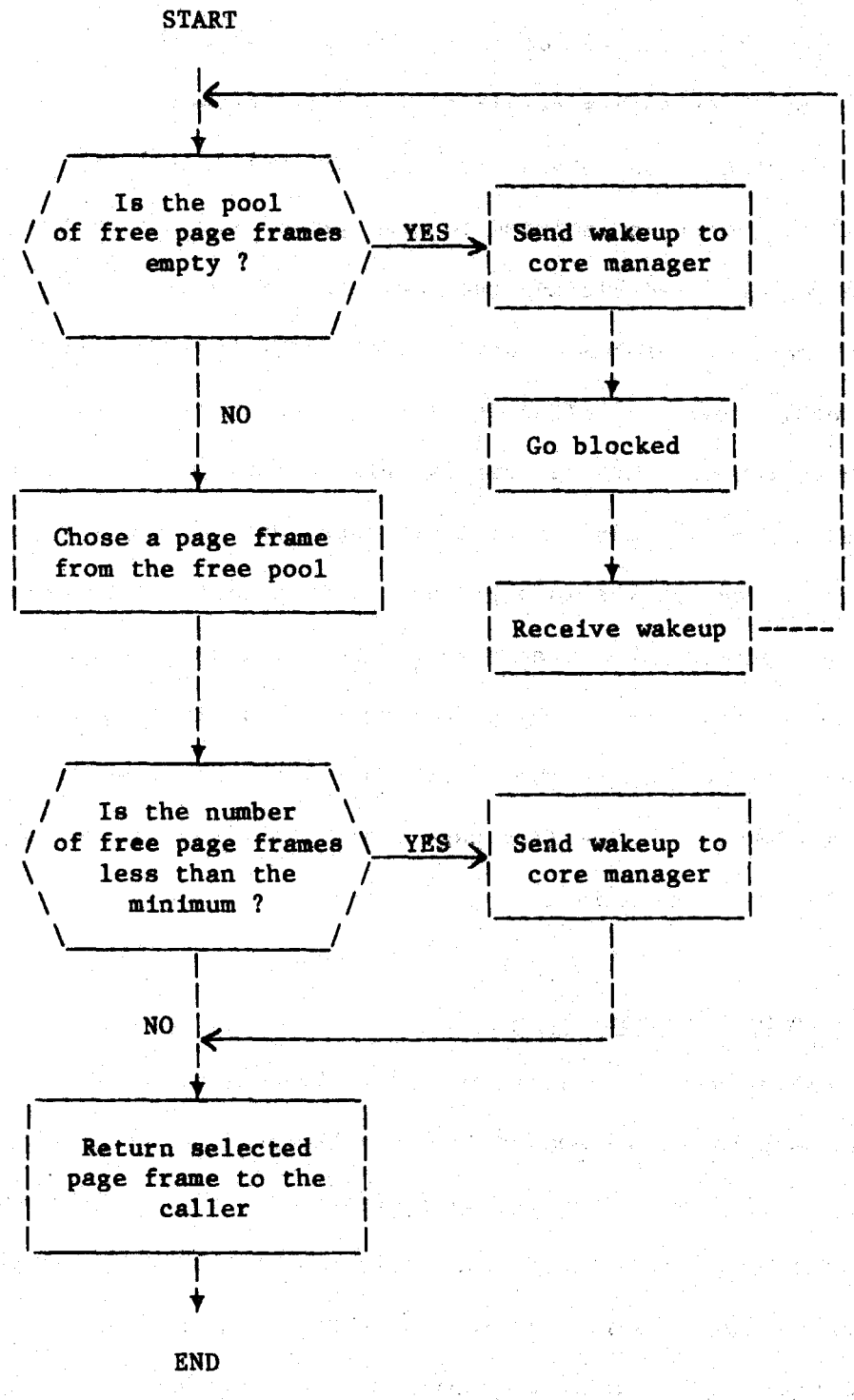



Figure 3.3

Algorithm of the page frame allocator procedure

allocating procedure to return the requested page frame. (A complication may arise here with the use of locks; see section 5.1)

Two additional points remain to be made. First, adopting the just described strategy means the algorithm of Figure 3.2 is incomplete. An additional step must be included to send wakeup signals to any processes that have gone blocked because the page frame pool was empty. Second, since any number of processes may be requesting free page frames simultaneously, some technique is necessary to insure a page frame is not allocated to two requestors. For example, a lock on the free pool is sufficient. The fact that several processes may be competing for any page frames in the free pool also explains the loop in the algorithm of Figure 3.3. When a process is awakened by the core manager, there is no guarantee that there are still page frames in the free pool, since other processes may have grabbed them all. Therefore, after going blocked to await replenishment of the free page frame pool, the algorithm must be repeated from the beginning.

The Paging Device Manager Process

The paging device manager process is the second of the two system processes used to manage memory in our multi-process design. Chapter 2 noted the similarity of the paging device memory to the main memory, and that allocating and deallocating page frames must be done for each level in the multi-level memory hierarchy assumed in our model. In fact, the allocation and deallocation of paging device page frames is so similar to the allocation and deallocation of main memory page frames that the algorithms to be used by the paging device manager process and the core

manager process are almost identical. Figure 3.2 describes the paging device manager's algorithm as well as the core manager's algorithm. The details need not be the same, e.g. no doubt a different policy may be in force for deciding which paging device page frames are to be freed, but the general form and structure are the same.

In a like manner, Figure 3.3 also describes the algorithm used by the paging device page frame allocating procedure, except of course the wakeup signals would be directed to the paging device manager process rather than the core manager process. The parameter used to trigger the signal to the paging device manager, the number of free paging device page frames, may also be different.

## Handling Page Faults

Now that we have added two system processes to do the deallocating of page frames at each level of the multi-level memory system, we turn to the allocation operation. Figure 3.4 shows the steps necessary to resolve a page fault, i.e. allocate a main memory page frame to a page in a system using two system processes to perform deallocation. The first box invokes the page frame allocation procedure, previously presented in Figure 3.3. This may result in the faulting process blocking itself if no free page frames are available. In the usual case however, a free page frame will be available and will be returned. The page may then be bound to the allocated page frame, and the necessary read operation begun to read the contents of the page into the memory locations of the page frame.

The remaining procedure needed to fill in the picture completely is the procedure which performs the allocating of pages to paging device page

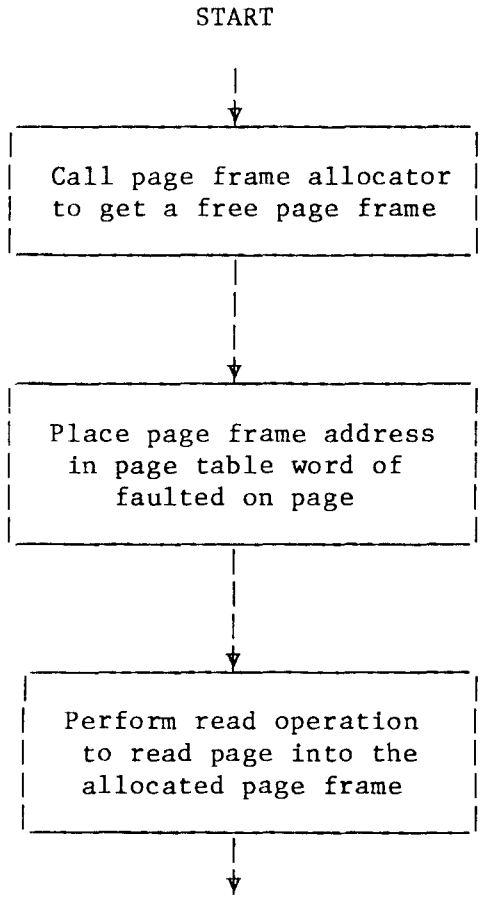



Figure 3.4

Binding a page to a page frame

frames. This occurs during the freeing of main memory page frames. One of the steps in the algorithm of Figure 3.2 is to free the main memory page frame from its page. This deallocation results in the contents of the page being copied to some other page frame in the memory hierarchy. Thus the replacement really expands into the three steps already depicted in Figure 3.4 for allocating a page to a page frame. That is, a paging device page frame is allocated and the page is written to the memory locations of the paging device page frame.

The interrelationship of the core manager process, the paging device manager process, and a process trying to resolve a page fault is illustrated by Figure 3.5. The boxes represent program modules which perform the function indicated by their label. The solid arrows depict calls made by one module to another, and the broken arrows represent interprocess signals. For example, the main memory allocation procedure will send a wakeup signal to the core manager process when the number of core page frames becomes too low, as indicated by the broken arrow from the box labeled "allocate core" to the box titled "core manager". Similarly, if in removing pages from main memory the core manager discovers there is an insufficient supply of free paging device page frames, a wakeup signal is sent to the paging device manager process, represented by the arrow from "allocate pd record" to the "paging device manager".

This design, as implemented on the Multics system as described in Chapter 4, incorporates the same features as the Multics page control for preventing data base contention and page fault contention. That is, a global page table lock prevents the core and paging device managers from

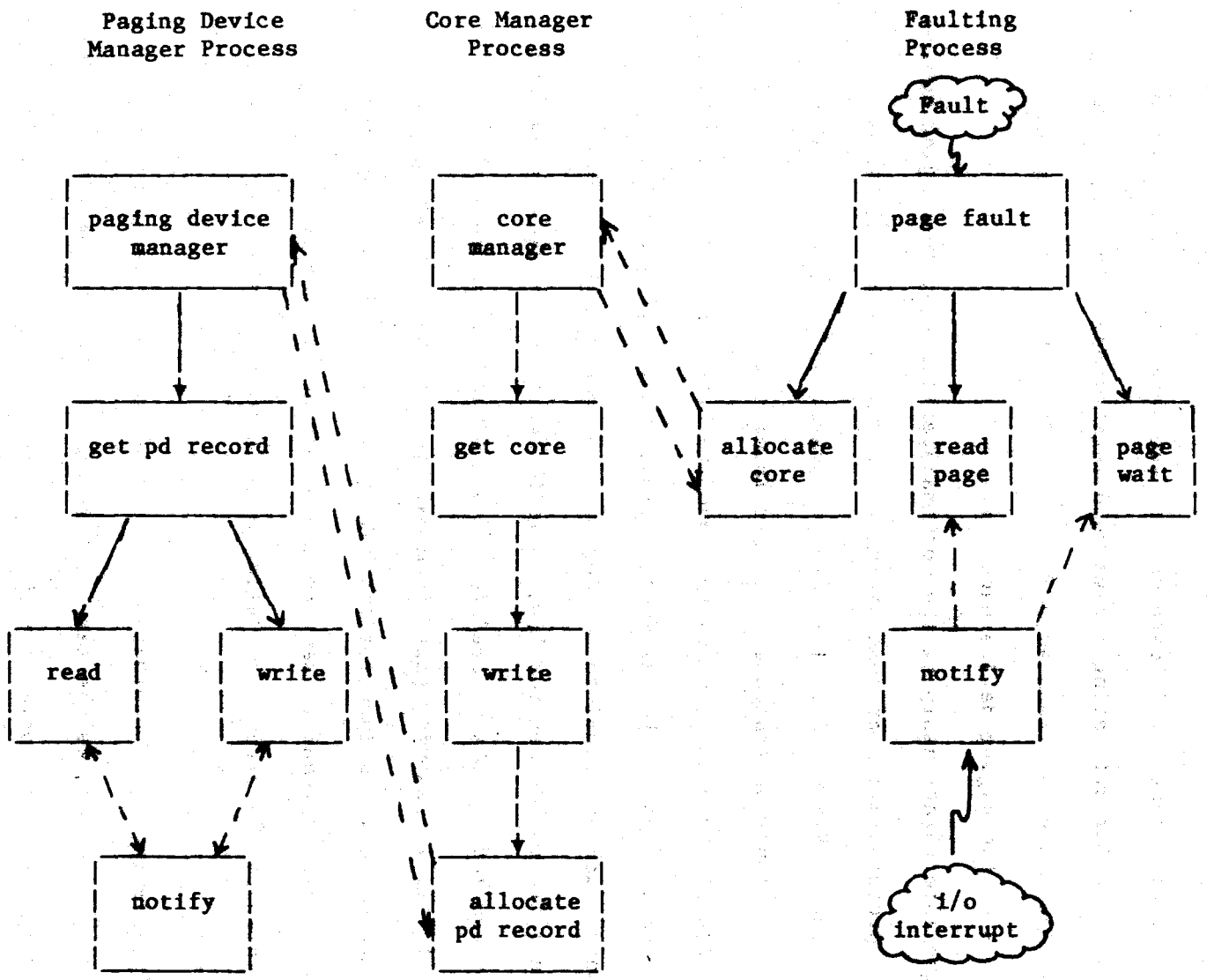



Figure 3.5 -- Multi-process page control

executing simultaneously, or one of the system processes from running while a user process was resolving a page fault. Per page locks are also used to solve the page fault contention problem. However, one of the benefits seen from this design, as discussed in section 3.4.4, is the potential for splitting the global page table lock. This question will be considered fully in Chapter 5.

### 3.3.2 Hoare's Structured Paging System

Hoare has proposed [Ho73] a multi-process paging system intended for a general computer system. The model Hoare uses for a general computer system is similar to the model assumed here; the major difference in the models used is due to the one level memory incorporated into Hoare's model. That is, Hoare assumes a memory system consisting of a main memory and a drum as a backing store, but does not include a second level of memory such as the disks assumed here.

Hoare uses monitors [Ho74] to describe his system. Monitors are procedures with built-in synchronization primitives. A monitor defines a group of procedures only one of which may be in execution at any time, thus ensuring mutual exclusion among processes executing the procedures comprising the monitor. Hence monitors are a high level locking device. In Hoare's system a monitor is assigned to each page; this monitor includes procedures to access the page, bring it into main memory on demand, etc. Thus a process faulting on a page invokes a procedure in the monitor for that page to bring the page into core. The built-in

synchronization ability of the monitor ensures that another process does not simultaneously attempt to bring the same page into core.

Memory deallocation is done by system processes in Hoare's design. Rather than using a single process for each level of memory, Hoare assigns the page replacement task to a separate process for each page. When a page is brought into main memory in Hoare's system, a process is created and started up which periodically tries to throw the page out of main memory if it has not been referenced recently.

Hoare's monitors permit a high level solution to both the page fault contention problem and the data base contention problem. The monitors assigned to each page are essentially per page locks, solving the page fault contention problem. Similarly, putting the other paging system functions inside a monitor also guarantees exclusive access to paging system data bases.

While Hoare's monitors allow him to describe his system in a rather elegant fashion, the system suffer two serious drawbacks in practice. The first is actually implementing the synchronization implicit in the use of monitors. There are serious efficiency issues unanswered here because a combination of hardware, or "busy" waiting, and software waiting is required.

The second, perhaps more serious deficiency in Hoare's proposal is the number of processes involved, one for every page in main memory. There is always overhead involved in implementing processes, both in keeping track of the state of the process, and scheduling the process at the appropriate time. Most systems are not capable of supporting the large number of processes required, and most schedulers are not designed

to give the fast response that would be necessary to make Hoare's scheme efficient enough for practical purposes. For these same reasons Hoare's system would expand poorly to a system with more levels of memory. Adopting the same strategy of one removal process per page would worsen the problems of implementing and scheduling the necessary number of processes.

There is an orthogonal viewpoint of paging systems from that taken in this thesis, a view which Hoare's description adopts in part. We have pictured pages as objects manipulated by system and user processes. Instead, each virtual page may be thought of as a process, a process that performs all desired actions on the page, moving it in and out of memory, wiring it, etc. (Not just removing it from memory as do Hoare's processes.)

This concept of a page as a process has also been used to explain Multics page control. (1) As already pointed out above, it is prohibitively expensive to actually implement a process for each page, however pages can be thought of as being implemented as very simple processes with page control acting as an interpreter for these processes. The per page information (e.g. flags, locks) define the current state of each page process; the various actions of the page processes (such as wiring themselves, bringing themselves into memory) are done interpretively by the page control code.

A more formal characterization of this view is to define each virtual page to be a finite state machine. The state of each such finite state

(1) This description of Multics page control is originally due to Bernard Greenberg of Honeywell Information Systems.

machine (page) is defined by the values of all the per page information contained in and associated with the page's page table word -- the used and modified bits, the wired flag, etc. Each transition of the finite state machine corresponds to an action performed on the page, and is implemented as some page control procedure.

For example, two states of a page are the "in core" state (i.e. allocated a core page frame as indicated by the page frame address in the page table word) and the "out of core" state (not allocated a core page frame as indicated by the fault tag in the page table word). The transition from the "in core" state to the "out of core" state is implemented by the code of the page replacement algorithm. Conversely the transition from "out of core" to "in core" is performed by the allocation code. The inputs which cause the various state transitions are requests from processes, e.g. a user process wishing to reference a particular page may cause that page to move from the "out of core" state to the "in core" state (and as a side effect cause some other page to make the transition from in core to out of core).

Page control, then, emulates these finite state machines by driving the pages through the various states in response to the demands of user processes. Hoare's monitors, which perform all the allowable actions (transitions) on pages, make explicit the concept of a finite state machine. The procedures of the monitor directly implement the state changes of the page.

### 3.3.3 Saxena and Bredt's Hierarchical Paging System

As part of a structured design of an operating system Saxena and Bredt [Sa75] include a description of a paging system. Their hierarchical operating system consists of four levels, numbered one to four, each level built on top of the lower numbered levels (level 0 is the hardware). The four levels are: 1. A simple scheduler for running and synchronizing a fixed number of system processes. 2. A simple memory manager which implements a virtual memory for these system processes. 3. A scheduler for implementing and synchronizing a large number of concurrent processes using virtual memory. 4. A memory manager for implementation of the virtual memory. Essentially the simple scheduler and simple memory manager implement system processes which provide complete process multiplexing and virtual memory to a large number of user processes. Monitors are also used to describe this system, and to solve the data base and page fault contention problems.

The chief distinction of this system from the one presented in section 3.1.1 is in the extra scheduler and memory manager. Like Hoare's system, only a single level memory is considered. However, unlike Hoare's system, only a small fixed number of processes is necessary to implement the paging system, because a process is not assigned to each page. Saxena and Bredt specify a page replacement process which, like the core manager process of Figure 3.2, can operate on any page, rather than creating a separate replacement process for each page as Hoare does. And instead of assigning a separate monitor to each page, a single monitor performs the memory allocation function for all pages. Thus, only one page fault may

be handled at a time.

Though much closer to the design presented here, there is a fundamental difference, due to the extra scheduler and memory manager. The high level scheduler (which is itself a process at the simple process scheduler level) and the high level memory manager processes are both allowed to take page faults. These "system" page faults are handled by the simple memory manager. This means there are two different kinds of page faults: "system" page faults and "user" page faults, and it must be possible to differentiate between them. This is an added complexity, and one which may require hardware assistance which not all systems may be capable of providing.

What is gained by the extra levels of scheduler and memory manager? Primarily the ability of the high level scheduler and memory manager to use, in a limited fashion, the functions each implements. Thus, the high level memory manager may be implemented as processes which may take page faults; similarly with the high level scheduler. The extra levels also solve the problem of whether to implement the virtual memory below the scheduler or vice versa.

System designers are often presented with a dilemma because both the scheduler and the memory manager would like to use the function implemented by the other. If the operating system is designed hierarchically, whichever of these two modules is implemented beneath the other cannot use either the function it itself provides or the function provided by the higher module. The problem is generally solved by splitting the scheduler or the memory manager into two levels, one below and one above the other module. Having two levels of each as do Saxena

and Bredt removes the mutual dependency of the top two levels.

In practice, the advantages of allowing the memory manager and scheduler to take page faults may never be realized. Supposedly, paging the memory manager and scheduler will free physical memory for user pages. Yet the pages of these two modules are normally so heavily used that they will always be in main memory anyway. There is also an efficiency issue in allowing the scheduler and memory manager to take page faults, for overhead is increased and response time adversely affected. This is a major reason why many systems make these these two modules permanently resident.

Hence transparency of structure rather than efficiency is the real issue. Careful design may eliminate the need for two levels of both the memory manager and the scheduler. Such a design has been proposed for Multics using a two level scheduler and a single memory manager. A simple scheduler implemented below the virtual memory would allow use of processes by the virtual memory manager, while a more complex scheduler implemented above the virtual memory would implement user processes and be able to take page faults. By careful design, the low level scheduler does not need to use the virtual memory.

One of the key questions here is the larger issue of the proper structure for an operating system. We have concentrated on the design of just one part of an operating system, the paging system. The previous discussion points out the need for considering the design of the paging system in the context of the overall system structure. The general problem of structuring operating systems has been treated by many researchers [Li72] [Di68b] [Ha70], and is beyond the scope of this thesis.

### 3.3.4 System Versus Combination Paging Systems

Little has been said to this point about system-process paging systems, with the exception of the discussion in section 3.2.2 considering Multics as a system process paging system with a single page control process. To remedy this deficiency, we discuss in this section how the two process combination paging system presented in section 3.3.1 (and implemented on Multics as discussed in Chapter 4) could become a system process paging system using three system processes to implement the page control functions.

The combination paging system of section 3.3.1 can be made into a pure system process paging system by removing page fault handling (memory allocation) from the user processes. Instead, a third system process will be assigned the page fault handling job. Thus a user process taking a page fault sends an interprocess message to this fault handling process which performs the steps of Figure 3.4. When the faulted on page has been read into the allocated page frame, a message is sent back to the faulting process, starting it up again.

The essential difference between such a three process system page control and the two process combination page control is that memory allocation (page fault resolution) is occurring in a single system process instead of in many user processes. This has major implications in two areas: security and efficiency.

The system process design seemingly offers improved system security. The memory allocation code, and the data bases referenced by this code are removed from the address space of the user's process. This not only makes the user's address space smaller and more compact, but makes it impossible for the user to intentionally or inadvertently damage this code and data and thereby affect other users. This separation is important in systems with no protection mechanisms, but since most computer systems do offer some means of protection (e.g. supervisor mode, write protected memory, or rings as in Multics) there is likely to be little if any extra protection from the user afforded in practice by handling page faults in a separate process.

More significant is the effect of the page fault handling process on system efficiency. First, there is the extra overhead required by the interprocess messages needed to report the page fault to the system process, and to signal completion of the fault to the faulting process. Even if the message sending overhead can be minimized, there is the additional expense of scheduling, that is saving the state of the faulting process and starting the page fault process, and vice versa when the fault is completed.

There is yet another consideration with respect to efficiency, important in multi-processor configurations. Namely, only one page fault may be processed at any time, because there is a single page fault handling process to resolve page faults. While this could conceivably be remedied by explicitly adding a page fault handling process for each physical processor, note that the combination paging system does this implicitly by having the user process resolve the page fault. Since as

many user processes may be executing simultaneously as there are physical processors, the combination paging system automatically expands or contracts the number of processes handling page faults at any time.

Of course, the preceding argument is irrelevent if a global lock is employed to prevent data base contention, because then only a single process may be resolving a page fault in any case. However, Chapter 5 will describe how using system processes enables splitting the global lock into several locks. Hence the tangible differences between the two designs are likely to be slight, and the decision as to which is best for a given system will depend heavily on such factors as the locking strategy and how efficient the implementation of processes is.

## 3.4 Advantages of Multi-Process Paging Systems

Having examined numerous multi-process paging systems, the question arises as to the superiority of such designs over a conventional design such as the Multics page control described in section 3.2.1. There are four areas where the multi-process designs offer decided advantages: simplicity, modularity, security, and expandability.

While these advantages accrue to all multi-process designs appearing in section 3.3, the following discussion pertains directly to the two process design presented in section 3.3.1 whose implementation is discussed in the next chapter.

### 3.4.1 Simplicity

The multi-process design is clearer and easier to understand due to the separation of the allocation and deallocation tasks into separate processes. Both the core manager process and the paging device manager are simple, sequential algorithms which can be understood without reference to the other parts of the paging system. In contrast, the corresponding algorithms in Multics are intertwined in a complex manner. This complexity is largely due to the fact that the three tasks split into separate processes by the multi-process design are lumped into a single process, that which takes the page fault. This process becomes something of a three ring circus, trying to do everything at once -- free space on the paging device, free space in main memory, resolve the page fault. In order to do so, an ordering must be imposed on these tasks, since a single process must do things sequentially. The fundamental problem here is caused by trying to place a sequential order on inherently parallel tasks. There is no satisfactory way to avoid these difficulties except to realize the parallel nature of these tasks and allow them to be done in parallel.

Separate processes also greatly simplify the treatment of i/o interrupts. The chief source of difficulty with input and output operations is the relatively long time they require relative to instruction execution times. We have already seen that in the Multics page control the process which starts a read write sequence does not wait for the disk write to complete, since to do so would delay page fault resolution. Therefore the completion of the read write sequence must be noticed by whatever process is around at the time. This of course

complicates things, as all processes must be ready to pick up where someone else left off.

On the other hand, the paging device manager process can wait for a read write sequence to complete, since his job is mostly nothing but performing read write sequences. Similarly, the core manager process, once a write has been started to copy a page to the paging device or to disk, can simply wait until the write is finished.

Essentially we are arguing in favor of a separate process for performing i/o (e.g. the paging device manager process doing the i/o for read write sequences) as opposed to a traditional interrupt handler, which spreads the i/o among whatever processes are executing. There are two chief advantages of the process approach over the interrupt handler.

The first of these is the clarity of structure of the process approach. The sequential nature of a read write sequence is obvious from the paging device manager's algorithm: start a read, wait for the read to complete, start the write, wait for the write to complete. In contrast, the same algorithm implemented in an interrupt handler obscures the fact that a disk write always follows a bulk store read in performing read write sequences. Some process starts the read; when the read completes the interrupt handler receives control. Interrupt handlers are invariably written as dispatchers -- the source of the interrupt is determined and appropriate routines performed to do whatever is necessary. Thus, after determining the read portion of a read write sequence has completed, the interrupt handler starts the write. The interrupt handler regains control later, on completion of the write, and finishes up.

In other words, the process which starts the i/o is best equipped to

know what actions should be taken when the i/o completes. Having a process perform i/o allows us to take advantage of this fact, while using an interrupt handler places all knowledge of what action to take in the interrupt handler code, forcing the interrupt handler to sort out all the various possibilities.

The second major advantage of the process approach is that it permits formalized interprocess communication mechanisms to be used in implementing the i/o. Block and notify primitives may be used by the i/o process, which blocks after starting i/o. The process receiving the interrupt merely turns it into an interprocess notification (the "notify" of Figure 3.5). The awakened i/o process then continues with whatever steps are appropriate upon completion of the i/o. In addition, the i/o process can, if necessary, wait on a lock, where an interrupt handler cannot (since the interrupt handler may have interrupted the process that locked the lock).

The end result is a simplification of the treatment of interrupts; only the lowest level of the system, directly above the hardware, need be aware of and deal with interrupts. All the processes performing i/o implement the i/o in terms of waiting on events using the standard interprocess communication tools.

The philosophy of using separate processes for i/o in place of interrupt handlers is given in more detail by Clark [C174].

Dedicating a process to manage the paging device allows another simplification in performing read write sequences. A read write sequence requires a main memory page frame. If any process may start a read write sequence if may be difficult to obtain the necessary page frame without

adding complex module interconnections. Since the paging device manager repeatedly performs read write sequences, a main memory page frame may be assigned to the paging device manager permanently for use as a buffer, avoiding the problem of dynamic allocation. This solution is possible in the Multics page control, but much more difficult for two reasons: 1) Since any process may start a read write sequence, any page frame used as a buffer must be protected against multiple simultaneous use. (Note in the multi process scheme the paging device manager process acts as a lock on the frame used as a buffer.) 2) A single process may start several read write sequences at the same time. (This is how the Multics page control achieves parallelism.) This would require several page frames be available as main memory buffers.

The factors just discussed result in a simpler, easier to understand paging system. This has important ramifications in many areas. Since the code is simpler and more understandable, it is easier to modify and maintain. This is valuable not only in testing and debugging the code, but in being able to change the algorithms at a later date with confidence that the system will continue to work, and to be able to predict any changes in system performance. For the same reason, the code would be easier to certify, or to use in proving a given property about the system.

### 3.4.2 Modularity

The separation of the main memory page replacement function and the paging device page replacement function into separate processes makes

possible a much cleaner modularization of page control. This is apparent by comparing Figures 3.1 and 3.5. For example, it is clear from Figure 3.5 that the main memory replacement algorithm (represented by the box labeled "get core") is part of the core manager process, and is invoked only by the core manager. This is not the case with the Multics design of Figure 3.1, where when performing paging device page replacement we can suddenly find ourselves executing the main memory page replacement algorithm.

Improved modularity reduces the possible paths through the code, i.e. lessens the interconnections between modules, and simplifies the interfaces between the resulting program modules. Many of the benefits of better modularity match those discussed in conjunction with simplification. However, though improved modularity and greater simplicity complement each other they are not the same thing. Modularity can be bought at the expense of complicating the individual modules; conversely a system often can be made to seem simpler by increasing the number of modules.

The most important advantage of the modularity of the multi-process design is when considering modifications of the design to other systems. For example, consider a computer system with paging but without the multi-level memory assumed in Figure 2.1, i.e. consisting only of main memory and disks, without a paging device. To use the two process design presented in section 3.3.1 would require elimination of the paging device manager and a slight change to the core manager so that pages evicted from main memory were always written to disk. Similarly, if another level of memory were added, another module analogous to the paging device manager

could be added in a relatively straightforward manner to manage the additional memory. That is, the design expands and contracts easily and modularly to fit any multi-level memory system. Either of these two modifications would necessitate extensive, major alterations to the page control of Figure 3.1, due largely to its lack of functional modularity.

### 3.4.3 Security

The multi-process design presented here offers significant security advantages over a traditional scheme. By security we mean the prevention of unauthorized release or modification of information (either procedures or data). Dividing page control into separate processes increases security between parts of the system, and allows separation of policy from mechanism within page control.

Protection of the user from the system, or the system from the user, is not directly enhanced where mechanisms such as supervisor mode, rings etc. already exist. However, the advantages of simplicity and modularity previously discussed would make any attempts at certification of the multi-process page control much easier. For example, the places that read and write arbitrary pages are localized and easily identifiable, and few in number.

Security between parts of the system is affected by the separate address space afforded each page control process. For instance, only the paging device manager process need be permitted to execute the paging device page replacement algorithm. Since the paging device used list is

used primarily for this task, we can also restrict access to the paging device used list to the paging device manager process. No other processes need access to this list.

Separation of policy from mechanism is possible if the system offers rings as does Multics (or some other form of protection domains) [Sc75]. The address space of each page control process can further be divided by use of these protection rings. The programs implementing the mechanics of paging, e.g. reading or writing a page from or to disk, adding or removing a page frame from a list, gathering usage statistics, etc. can be placed in the most privileged ring. The policy algorithms, e.g. deciding what page to remove from primary memory, execute in a less privileged ring, and must call the inner ring procedures to get the information needed and to actually implement the decisions made. Thus the failure of the policy algorithms could never cause unauthorized use or modification of the information in the pages. The system could be certified without having to certify these policy components. (Failure of the policy algorithms could still result in denial of service.)

To summarize, the separation of the parts of page control permitted by the multi-process design effectively allows extra "fire-walls" between pieces of the system and and between procedures implementing mechanisms from procedures deciding policy.

### 3.4.4 Expandability

Expandability encompasses two ideas. One has been mentioned in the

discussion of modularity and might better be termed adaptability, namely the ability to add another manager process to the paging system to manipulate another level of memory. The second aspect of expandability is the ability to increase the number of processes executing as core or paging device managers as the size of the computer system grows.

In a generalized computer utility with multiple processing units and large amounts of memory, a point will eventually be reached where a single core manager process will be unable to supply free main memory page frames fast enough, even if the core manager is always executing, since with several processors there will be multiple user processes executing simultaneously, each taking page faults and demanding page frames. In such a situation, the solution is to create additional core manager processes (or paging device manager processes) as needed to supply free page frames at a sufficient rate. All of the core manager processes would be identical, and follow the algorithm of Figure 3.2.

This design would be rather inefficient if the global locking strategy used by Multics is employed. The multi-process design, however, enables elimination of this lock by structuring the paging system's data bases into distinct parts, each of which needs to be accessed only by a single process (or type of process, e.g. if there are multiple core manager processes). This would significantly decrease the interference among processes, producing a corresponding increase in system efficiency. This issue is considered in more detail in Chapter 5.

To conclude, the multi-process design offers advantages in simplicity, ease of understanding, increased functional modularity,

enhanced user and system security, adaptability and expandability. The implementation described in the next chapter demonstrated that these are not just theoretical benefits but offer practical advantages as well.

# CHAPTER 4

## A Multics Implementation of Multi-process Page Control

### 4.1 The Multics Implementation

Many readers will doubtlessly be strongly tempted to skip this chapter; we urge this temptation be resisted. Although the topic of this chapter is an actual implementation on the Multics system of the multi-process paging system presented in section 3.1.1, the emphasis is not on the details of Multics or the particular implementation of a paging system. Rather, the emphasis is on the insights gained into the design by its implementation. There are always problems arising in implementing a system that are not apparent from the design of the system. The purposes of implementing a real multi-process paging system were to demonstrate the validity of the design, determine if the system's theoretical benefits were manifested in practice, and to measure the performance of such a system.

### 4.1.1 Size and Scope of the Implementation

To give some idea of the size of the system implemented, the standard Multics page control consists of 28 modules written in assembly language and PL/1. These total approximately 4700 source statements, 3600 in assembly language and 1100 in PL/1, which compile into almost 11,000 lines (words) of object code. To implement the multi-process design, extensive changes were necessary. These changes are summarized in Appendix A, which lists the modules in the Multics page control that were changed or deleted, and the modules that were added. Appendix B lists the program modules required for the multi-process page control. For ease of implementation, the entire multi-process page control was written in PL/1 except where already existing components written in assembly language were used with little or no alterations. The size of each of the modules in source statements is also listed in Appendix B, and the size of the object code for each program. Excluding minor changes in existing modules and some changes to the scheduler needed to enable implementing page control as system processes, approximately 1700 PL/1 statements were written. The total size of the 32 modules comprising multi-process page control was roughly 3700 source statements, 1500 in assembly language and 2200 in PL/1. Note the number of PL/1 source statments doubled while the number of assembly language source lines was reduced by more than half. Because of the large increase in PL/1 source lines, the resulting modules compiled into slightly more than 13,000 lines (words) object code. This increase in size was due to the effect of writing the programs in a higher level language.

The structure of the implemented system was identical to that illustrated in Figure 3.5. Both system processes, the core manager process and the paging device manager process, were driven by control procedures named "core manager" and "pd manager" respectively. These programs received wakeup signals from other processes, determined what action to take as a result of those signals, called the necessary routines to accomplish that action and then signalled the completion of that action to any waiting process before blocking the system process. A more specific idea of how these processes work may be gotten from Appendix C, which contains some of the actual PL/I source programs for the core_manager and pd_manager modules. For completeness, comparable code from the third part of the system, the page fault path, is also included. This is the code that runs as part of the user process and is responsible for resolving page faults.

### 4.1.2 Differences of the Implementation from the Model

There were several points in the actual implementation where it was found necessary to deviate from what the model implies. One of the most significant of these was in the mechanism used to implement the core and paging device manager processes. The model does not differentiate between the system processes used to implement the core manager and paging device manager and the typical user process except in the functions they perform. In practice however, they may need to be implemented differently in order to obtain the efficiency and responsiveness required for system functions.

Additionally, the system processes must be able to operate without taking page faults, since they are used to implement page faults.

Hence a special type of process was used to implement the core manager and paging device manager processes that were simpler and involved less overhead than a full Multics process. All procedures, tables, and temporary variables used by the core and paging device manager processes were fixed permanently in main memory. The processes also lacked the ability to add new segments to their address space, but this is not an ability needed by the page control processes anyway.

The manager processes were also restricted from using the full interprocess communication mechanism of Multics, because to permit them to use this facility would have required much more code and data be kept in main memory permanently. Instead, less powerful primitives were used which allowed processes to wait on events and signal the occurrence of events but did not allow interprocess message sending. The use of these primitives, which were already part of the standard Multics system, had some performance implications because of their interaction with the Multics scheduler. Therefore, a special set of primitives was implemented and used only for waking up the memory manager processes. These primitives insured that once either of the system processes was ready to run, it was started as soon as possible.

Another difficulty involving the wait primitive arose from the restricted environment a process operates in after a page fault. At this time, the faulting process cannot take another page fault, thus it must run on a wired stack. Multics does not provide a wired stack on a per process basis, but rather on a per physical processor basis. In a

situation where a process needs a wired stack, it uses the wired stack (the "prds", or processor data segment, in Multics terminology) associated with the physical processor currently executing the process. This has severe consequences for the waiting operation. If a process surrenders the processor while using the prds as a stack, its stack history is lost. The next process to run may overwrite the prds stack, and even if this could be prevented the process may run on a different physical processor (with a different prds) when restarted.

The result of this restriction is that if a process resolving a page fault must wait in a manner which requires abandoning the processor, it must do so at a point where it has no stack history on the prds. This situation arises in the implemented multi-process page control when a faulting process calls the main memory page frame allocator, who discovers there are currently no free page frames. At this point the core manager is signalled to free more page frames, but the faulting process must wait, blocking itself and surrendering the processor. If the faulting process did not give up the processor, the core manager process might never be able to run (e.g. in a single processor system). Thus the stack history at this point must be lost. This is not too severe, since nothing has really been done up to this point other than determining what page caused the fault. The mechanism used to solve this problem is to have the wait primitive note the process is running on the prds, and restart the process by repeating the instruction that caused the page fault when the process is unblocked. This same action, repeating the faulting instruction, is also used to restart a process waiting for the completion of a read operation to bring a faulted on page into core. In the first case, since

the fault has not been resolved, the page fault code is invoked again, but this time there should be a page frame available. In the latter case, the fault has been successfully resolved, and the process continues merrily on its way.

To summarize, the implementation differences were due primarily to the simpler type process used to implement the core and paging device manager processes, which imposed some restrictions on the functions these processes could perform, and to the strategy used on Multics for implementing a wired stack. The other differences from the model due to segmentation are presented in section 4.2, and result in adding extra functions required to deal with segmentation to the job of the system processes.

### 4.1.3 Performance

To compare the performance of the multi-process paging system with the standard Multics paging system, a system benchmark was run using both systems. A slight change was made to the standard system in order to obtain more meaningful results for comparison. The reason for this change was the larger size of the multi-process page control system. Nine additional pages of memory were devoted to permanently wired system programs and data in the multi-process page control. This meant that the primary memory available for holding user pages was reduced a corresponding amount. So that the size of main memory usable for paging by user processes would be comparable in both cases, nine additional pages

were wired in the standard system and left empty.

This modification did not make the size of the pageable memory exactly equal on both systems. The multi-process page control keeps a free list, and the number of frames on this list varies constantly as the core manager adds page frames and faulting processes request them. Each page frame on the free list reduces the amount of available memory available for paging; if, on the average, two page frames are on the free list, the effective pagable memory has been reduced by two pages. When the benchmark was run, the core manager was set to keep between four and eight page frames free. (That is, when awakened, the core manager would keep freeing page frames until there were eight; the allocating procedure would wake up the core core manager when the number fell below four.) A very conservative estimate is that on the average three pages were on the free list. To compensate for this effect, another three pages were left empty and wired when running the benchmark on the standard system, for a total of twelve (the previous nine pages due to the increased wired code and data plus three to compensate for the pages on the free list).

The results of running both systems are summarized in Figure 4.1. (1) The multi-process page control system took 8.7% more page faults. The increase in page faults is accounted for by three effects. The first of these is the inability of the adjustment described in the preceding paragraph to make the effective pageable memory exactly equal for both systems. The second effect is due to differences in the algorithms used

(1) While useful for comparison, these numbers were obtained in a special test environment and do not reflect the normal operating performance of Multics.

| | Standard MIT System | Multi-process page control | |
|---|---|---|---|
| | | Actual | Estimated |
| Number of page faults | 60,261 | 65,504 | 65,504 |
| Average time to process a page fault (microsec.) | 1973 | 2043 | 1226 |
| Total CPU time attributable to paging (sec.) | 119 | 307 | 184 |
| CPU time spent (sec.): | | | |
| processing page faults | | 134 | 80 |
| in core manager process | | 141 | 85 |
| in paging device manager process | | 32 | 19 |

Figure 4.1

Performance of multi-process page control

for page replacement, specifically in when pages are replaced. Since the multi-process page control evicts pages before the system runs out of free page frames while the standard system only replaces pages when no free page frames are left, the pages held in memory at any given time may differ. Given the same execution sequence, changing the pages in memory will cause a different fault pattern and fault rate. Third, the average time to resolve a fault changed, as Figure 4.1 shows. Any difference in the time required for any event in a multiprocessing environment can alter the pattern of page faults by changing the contents of the memory.

Although the average time spent processing a page fault remained relatively constant, these times are measured differently and are not directly comparable. Since page replacement in both main memory and the paging device is done at page fault time in the standard page control, that time is included in the time to process a page fault, while this time is attributed to the core manager or paging device manager in the multi-process scheme. Thus one would expect the time spent processing a page fault to be much less for the multi-process implementation.

The fact that the time is not smaller is due to the overhead of PL/1. In the standard system, all but a small fraction of the code that runs at page fault time is written in assembly language. In the multi-process system the situation is reversed, with the large majority of the programs written in PL/1. There are two sources of overhead attributable to PL/1 at work here. One is the fact that in general, algorithms written in assembly language are shorter and execute faster than the same algorithm written in PL/1. (In cases, the object code generated by PL/1 may be a factor of two or three larger.) Second, and more important, is the

overhead involved in making a PL/1 external procedure call. In the assembly language version, subroutine calls and returns are made via a single transfer instruction. A more complex sequence is required in PL/1 so that the stack and the PL/1 environment are managed properly.

In Multics measurements have shown that a PL/1 external procedure call requires on the average 67 microseconds. This figure is for a call with no arguments; each argument passed adds approximately two microseconds. The path followed after a page fault occurs in the multi-process page control involves twelve external calls. Using 70 microseconds as an average time for one external call (i.e. assuming one and a half arguments per call), this means that a total of 840 microseconds of the average 2043 microseconds required to resolve a page fault, or about 41% of the total, is due to the procedure call overhead alone.

A similar calculation shows that twelve PL/1 calls are also executed in the course of freeing one main memory page frame. Measurements from the benchmark show, assuming that all of the time spent by the core manager was spent freeing page frames (not strictly true, see sections 4.2 and 4.3), that an average of roughly 2100 microseconds was required to free one page frame. Again, the PL/1 call overhead was 840 microseconds, or about 40%.

Using this figure of 40%, and reducing the amount of time spent by each component of page control in the actual benchmark by 40%, gives the results shown in Figure 4.1 as the estimated performance of multi-process page control. This shows the estimated performance improvement if all the external PL/1 calls were changed to internal procedure calls.

There is a smaller effect due to the repetition of certain steps in each PL/1 program. For example, pointers to data bases may have to be initialized in several procedures, instead of just once as in the assembly language version of page control. Another factor in the increased percentage of processor time attributable to the paging system is the fact that some operations included in the total time charged to the multi-process page control are not counted towards the overhead of the standard Multics paging system (see sections 4.2 and 4.3). While it is extremely difficult to estimate the effect of these two factors on performance, their elimination might result in a further improvement of 5-10% over the estimates in Figure 4.1.

Achieving a performance level equalling or improving upon the current Multics page control was not a goal of the test implementation. However it is the author's belief that the multi-process implementation is not inherently less efficient; it could be made much more comparable if appropriate programming style was used, such as only using internal procedure calls, which Multics PL/1 implements very efficiently, and using global variables.

## 4.2 The Interface with Segment Control

Multics is a segmented system and has the concept of "active" and "inactive" segments as discussed in section 2.1.4. This necessitates some extra function in page control, which leads to a more complex core manager and paging device manager than would otherwise be the case. The

extra functions that must be added to page control, and the complications these extra functions introduce, are examined in the next two sections.

### 4.2.1 Necessary Segment Control Functions

The chief area of contention between segment control and page control is the page table. Page tables are allocated by segment control, but must be maintained by page control. When segment control wants to perform an action which may affect the page table words, it must call upon page control. In the case of the multi-process design of this thesis, that means the core manager and paging device manager processes.

There are four segment control functions which affect page table words. These are: 1. Activating a segment, which requires the file map containing the permanent disk addresses of the segment's pages be copied into the just allocated page table. 2. Changing the size of the the page table (a "boundsfault" in Multics), which requires the contents of a page table be copied into a new, larger page table when a segment grows. 3. Deactivation, which flushes the segment's pages back to disk. 4. Truncation, which deletes some or all of the pages of a segment, requiring the deletion of all copies of those pages in all levels of the memory system.

Of these four, only two require intervention by the core manager and paging device manager processes. Activation does not, because a process cannot take a page fault on a segment until the segment has been assigned a page table; thus segment control can be responsible for initializing the

page table. Similarly, when a process extends the size of a segment causing a larger page table to be allocated for it, the process can copy the page table itself, since no memory is allocated or deallocated the core and paging device processes need not be involved. On the other hand, both deactivation and truncation explicitly require memory deallocation, and thus the assistance of the memory manager processes, whose job it is to do memory deallocation.

Deactivation requires the "cleaning up" of any pages of the segment remaining in memory. Pages of inactive segments cannot stay in main memory or on the paging device because there will no longer be page table words for these pages. Thus the paging device manager must perform read write sequences on any pages of the segment being deactivated that reside on the paging device. Any pages in main memory must also be evicted, and the core manager must insure that the evicted pages are not put back on the paging device.

Truncation is somewhat easier in one respect, for no i/o need be done. Since the pages are being deleted, copies residing on the paging device or in main memory may simply be discarded, and their page frames claimed and added to the appropriate free list. Any disk copies of the deleted pages must also be thrown away, and the disk records they occupied returned to the file system for future reuse.

### 4.2.2 Complications Introduced

Since truncation and deactivation of segments both potentially

involve main memory and paging device memory deallocation; these operations are logical candidates for implementation in the core and paging device manager processes. Doing so necessarily complicates these processes as they no longer perform a single task. They must now be awakened when a segment is to be truncated or deactivated to perform the necessary steps. This means when the core or paging device manager is started up, they must determine why they were awakened, and perform the correct function. Note also that just sending a wakeup signal is insufficient; more information is required in the case of a truncation or deactivation. In both instances the segment on which the operation is to be performed must be specified; additionally for a truncation which pages are to be deleted must also be indicated.

Thus the core manager and paging manager become message receivers, responding to interprocess messages from other processes to free page frames, truncate specified pages or clean up designated segments. When a process wishes to truncate a segment, a message is first sent to the paging device manager process, which deletes any copies of pages of the segment on the paging device, returning the page frames bound to those pages to the free pool. Upon receiving notification of the completion of this part of the task from the paging device manager, a message is sent to the core manager process asking him to finish the job. The core manager deletes any copies of the segment's pages in core, adding their page frames to the pool of free main memory page frames, and signals that the truncation is complete. Deactivations are handled analogously, with pages being returned to disk rather than deleted.

An alternate strategy is possible and was contemplated for some time.

The truncation and deactivation functions could be performed by the user process, rather than asking the system processes to perform these tasks. This has the advantage of keeping the core and paging device manager processes simple, but distributes part of the function of page control back to the user. This implies deallocation of memory may be going on in more than one place at a time. There is clearly a trade-off here between making the system process more complex and shoving system functions back into the user processes. In the final analysis it was felt the prime consideration was to collect all the page control operations into a single process.

## 4.3 Other Page Control Functions

In section 2.1 two other page control functions were discussed: memory reconfiguration and memory wiring. In the context of the system processes, memory reconfiguration amounts to adding or deleting page frames from the supply that may be allocated; memory wiring means guaranteeing certain pages will not be removed from main memory. These tasks, though of secondary importance, are also within the province of the core and paging device manager processes.

The steps involved in adding or removing memory have already been described in discussing memory reconfiguration (see section 2.2.3). These steps are carried out by the appropriate memory manager process in response to a request from the process performing the reconfiguration. On completion, the reconfiguration process is notified. Hence reconfiguring

page frames presents no additional complications, merely increasing the number of functions the paging device manager and core manager processes must perform.

Wiring pages (section 2.2.4) was implemented as a system procedure called by user processes. The only effect upon the core manager was to include a check for wired pages when choosing pages for removal. Implementing wiring in this fashion requires no action by the core manager process and was done largely for convenience, as the currently used wiring procedure could be used unchanged. Wiring could be done by the core manger process just as easily; becoming an interprocess call instead of a simple procedure call. Absolute wiring, however, must be implemented by the core manager process since deallocation of some pages may occur and special allocation techniques may be necessary. This adds an extra function to the core manager process.

Unwiring pages can be implemented in the core manager process or simply by procedure calls. The choice is largely one of convenience. To reduce the amount of code rewritten for the test implementation, unwiring was implemented without the intervention of the core manager process.

# CHAPTER 5

## Eliminating the Global Page Table Lock

One of the major benefits of having multiple processes implement the paging system is the ability to simultaneously execute two processes performing page control functions. This parallelism in the performance of page control functions is lost, however, if a global lock such as used in Multics (section 3.2.1) is used to prevent data base contention. Since only a single process may have control of the lock, only one page control function may be executing at any moment. This of course prohibits handling several page faults in parallel.

In this chapter a strategy for splitting the global page table lock will be developed. By identifying the processes using each page control data base and which data bases each process may reference simultaneously a strategy using individual data base locks can be implemented. Such a scheme allows full advantage to be made of the multi-processing capability of the combination page control presented in section 3.3.1, including simultaneous handling of page faults.

## 5.1 The Strategy

One reason the global lock is used in Multics is that all page control functions are performed at page fault time. Thus a process handling a page fault will first access the paging device used and free lists, then the core used and free lists, etc. Since every user process taking a page fault must access all the page control data bases, all the data bases are subject to data base contention. The global lock protects everything, even though some data will no longer be referenced or are not yet needed.

Hence a first step in dividing the global lock is determining which data bases are subject to contention. Clearly if a data base is accessed by only a single process that data base need not be protected. Figure 5.1 presents this information for the page control data bases. For example, a user process handling a page fault would have to access the core free list to obtain a free page frame to allocate to the faulted-on page. Clearly the core free list must also be referenced by the core manager process since the core manager is the process responsible for adding page frames to the free list.

Not surprisingly, all the data bases are used by more than one process. Pages are referenced not only by both of the system processes but also by user processes faulting on the pages. The other four data bases are each accessed by two or more processes. To allow parallel execution while preventing contention, access to these data bases must be arbitrated in some fashion. A lock on each list is the obvious solution. Thus we assume a lock is associated with each of the four lists; the lock

| Data Base | Referencing Process | Reason for access |
|---|---|---|
| core free list | core manager | add a free page frame |
| | user process | obtain free page frame to resolve page fault |
| core used list | core manager | chose page frame for deallocation |
| | user process | add newly allocated page frame |
| paging device free list | core manager | obtain free page frame for allocation to page removed from core |
| | paging device manager | add a free page frame |
| paging device used list | core manager | add newly allocated page frame |
| | paging device manager | chose page frame for deallocation |
| pages (page table words) | core manager | when doing main memory page replacement |
| | paging device manager | when doing paging device page replacement |
| | user process | when resolving page faults |

Figure 5.1

Processes accessing page control data bases

must be set before access to the corresponding list is allowed.

Similarly a lock will be associated with each page, and the lock must be locked before operations may be carried out on the page (e.g. resolving a page fault). This of course is not new; Multics already has such a per page lock.

With multiple locks, precautions are necessary to preclude system deadlocks. Thus a second important step in eliminating the global lock and replacing it with distributed locks is determining under what conditions a process needs to lock more than one data base. If such conditions never occur, a system deadlock cannot occur due to two processes waiting for locks held by one another.

Situations where a process needs access simultaneously to two objects protected by locks occur frequently, as shown in Figure 5.2. For instance, any user process taking a page fault must lock the faulted on page while the page is read in, and while the page is locked the process must access the core used list to add the page to the used list.

At this point the next step is to develop a locking protocol defining allowable actions on the locks which guarantee system deadlocks cannot occur. We will use the standard Multics avoidance strategy which involves a "locking hierarchy" and "waiting rules". The locking hierarchy states the order in which locks are locked. This insures that if two processes both need locks A and B then both processes lock these locks in the same order, preventing one process from locking A and waiting for B while the other process locks B and waits for A. The waiting rules state when a process may wait for a lock without giving up the processor (i.e. when waiting may be "busy" waiting, done by repeating the attempt to set the

| Process | Data Bases To Be Locked | Situation Requiring Both Data Bases Be Locked |
|---|---|---|
| user process | page<br>core used list | Adding a page just allocated a page frame to the used list |
| | page<br>core free list | Asking for a page frame to allocate to a page that has been faulted on |
| core manager process | page<br>core used list | Removing from the used list a page that is to be removed from main memory |
| | page<br>paging device free list | Requesting a free paging device page frame to allocate to a page |
| paging device manager process | page<br>paging device used list | Deleting from the used list a page that is being removed from the paging device used list |

Figure 5.2

Processes locking multiple locks

lock until successful, as opposed to non-busy waiting, implemented in software and requiring the process to surrender the processor). Thus a process must not be allowed to surrender the processor (block itself) with a lock set if some other process might perform a busy wait on that lock.

It is not difficult to determine what the protocols must be. From Figure 5.2 it can be seen two levels of locks exist -- the locks on the four lists, and the page locks. A process needs to have only one of each locked at a time. Clearly, the protocol must require locking the page lock first. For example, after a page fault, the process taking the fault must lock the page before accessing the core free list to allocate a page frame. This is to insure another process has not already begun allocating a page frame to the page. Hence we have the following rule defining the locking hierarchy (order of locking):

A page must be locked before attempting any operation on the page, and before that page may be added or removed from the core used or free list, or the paging device used or free list.

The waiting strategy is largely determined by the relatively long i/o times. That is, pages must remain locked while read and writes from and to the paging device and disks are in progress. Hence pages will be locked for long times, making busy waiting on page locks hopelessly wasteful of processor time. (In addition, a process looping on a page lock could prevent the process that wished to unlock the page from ever executing and thereby freeing the lock.) Thus a process wishing to wait on a page's lock must block itself, giving up the processor. Note the hierarchy rule given above implies a process waiting on a page lock cannot possibly have one of the four used/free lists locked.

There is a further question of what to do if a process needs to lock several pages of the same segment simultaneously. Such a case may occur in performing such functions as deactivation or truncation (section 4.2.1) that operate on all pages within a segment. Usually such a problem may be solved by locking each page in turn, performing the necessary actions on the page, unlocking it and continuing with the next page, etc. In Multics this method is adequate, however if it is not sufficient, locking the pages in order by page number imposes the necessary lock ordering to prevent deadlocks.

For the locks on the used/free lists, busy waiting is not only possible but desirable. These lists need only be locked for several instructions, as long as required to add or delete an entry. Thus wait time should be minimal. Note assuming busy waiting here implies a process never gives up the processor with one of the four lists locked; that is, the add/remove operations must be non-interruptible.

To summarize, the rules for waiting on locks are:

1. A process must block itself while waiting on a page lock.

2. A process may block itself with a page lock locked.

3. A process may busy wait on the lock associated with any of the four lists of Figure 5.1.

4. A process may not block itself with one of the four lists of Figure 5.1 locked.

The last two rules are enforceable by requiring all additions and deletions to the lists be made using system functions. This has the added consequence that the callers need not even be aware of the existence of the locks or the rules. The primitives themselves are written to obey the

protocols. Indeed, if the used/free lists could be implemented without locks by carefully choosing their structure, the last two rules would be unnecessary. Thus the implementation would be transparent to the user of the primitives.

In other cases, following these rules may require knowledge of the implementation of certain system functions. In particular, section 3.3.1 discussed implementation of the routine that allocates free page frames. The approach chosen involves blocking the calling process if there are no free page frames. Processes using such an allocation routine must be aware that they may block themselves by calling the allocation routine, and ensure this would not violate the locking rules.

How do these rules manifest themselves in practice? Consider the core manager while attempting to free page frames. He attempts first to lock a page. If the core manager fails in this attempt to lock the page, he merely tries another page on the used list. (If he really must have this particular page, by the rules above he must go blocked.) However assuming the core manager succeeds in locking the page, he may then examine it to decide if it is a good candidate for removal. If the core manager decides the page should be replaced, he removes it from the used list (locking the core used list while doing so), gets a paging device page frame to write the page to if the page is not already on the paging device (locking the paging device free list momentarily) and starts writing the page. The core manager may then block himself until the write completes, at which time he adds the paging device page frame to the paging device used list, unlocks the page, and finally adds the now free core page frame to the core free list.

A process taking a page fault blocks himself if he cannot lock the faulted-on page. When the page is unlocked, the process will be awakened and can try again. When the process succeeds in locking the page, he can then determine if the fault still needs to be resolved.

Adopting the scheme outlined above will indeed permit not only simultaneous execution of both system paging processes (or multiple instances of system processes) but also parallel execution of user processes handling page faults. As long as user processes do not attempt to resolve faults on the same page they will not interfere with one another. Waiting for data bases is minimized because the data bases (lists) need remain locked only while items are added or deleted.

## 5.2 Locks on Segments

The locking strategy presented in the preceding section is insufficient in a segmented system such as Multics. This is because certain information about each segment is maintained by page control. For example, the number of pages of the segment that are currently in main memory is one such item of information. In a per page locking scheme, there is no way to protect such data without additional mechanisms. For example, a process faulting on a page will need to increase the count of the number of pages in core for the page's segment; if simultaneously the core manager process is evicting a page of that segment it must decrement the number of pages in main memory by one. A race condition may develop leaving the number in an inconsistent state.

Another example of per segment information which page control maintains in Multics is "quota". In Multics, quota is an upper limit on the number of pages the segments of a directory may contain. (Multics has a hierarchical file system where all segments are cataloged in special directory segments. A directory's quota restricts the amount of storage that may be consumed by segments within that directory.) Page control must keep track of the quota as well as the number of pages used by the segments in the directory. A full discussion of quota is postponed to the next section.

Thus in practice a segmented system would need to add another level of locks, namely per segment locks, to protect the information associated with each segment and manipulated by page control. It should be emphasized that although the term segment lock is used, these locks are used only by page control and not by segment control. Segment control may need to use some sort of lock for proper implementation of its functions; however, the segment locks discussed here are not intended for such use. The per segment locks discussed here are not locks on the segment, but on the page control information associated with each segment. Implemented beneath segment control, segment control should not be aware of their existence.

How should these per segment locks be incorporated? One solution would be to use the per segment locks in place of the per page locks. In this scheme, access to all of the pages comprising the segment as well as to the per segment information, would be controlled by the segment lock. Having a single lock control all the pages in a segment means that once a process has locked a segment while processing a page fault, no other

process could perform any action on that segment (e.g. fault on another page, remove a page of the segment from core) until the page fault had been completed. Note, though, this restriction would be advantageous under certain circumstances; i.e. when performing a segment operation such as truncation or deactivation which operates on all of the pages of the segment. In such cases locking the segment lock allows the entire operation to be performed, where in a per page locking scheme each page in the segment must be locked.

A better strategy is to implement the segment locks beneath the page locks, in the same manner as the locks on the used and free lists. The segment locks, like the locks on the lists, need only be locked for a few instructions while the per segment information is updated. The rules applying to the locks protecting the used and free lists must also be observed for the segment locks. That is, a process may lock a segment only after the page (if any) the process is operating on is already locked. Segment locks can be busy waited on, but a process must unlock any segments it has locked before abandoning the processor.

This strategy of implementing the segment locks does not conflict with the implementation of the locks on the free and used lists because a process never needs to have one of the lists and a segment locked simultaneously. (If such a situation did arise, appropriate ordering rules would prevent deadlocks.) Happily the addition of per segment locks does not place any restrictions on what page control functions may be executing in parallel. Several user page faults may still be resolved at once; if by chance page faults on two pages of the same segment are being handled, at worst one process will wait momentarily while the other has

the segment of interest locked.

## 5.3 Multics Complications

The per segment locking strategy just described for Multics has not been implemented. This section discusses two complications which prevented the segment locking scheme from being added to the multi-process implementation of page control on Multics in the time available.

The first problem is ensuring that the global page table lock is not being used in obscure ways by programs knowledgeable of its function to protect data against contention. In fact, one good argument for removing the global lock is to force such assumptions to be made explicitly. Knowing that a global locks protects many data bases makes it very tempting for a programmer to take advantage of the global lock by using a certain location in a data base as a temporary because he "knows" the global lock protects that location against any other use while he has the lock set.

As an example of a hidden use of the global page table lock, consider the following from Multics: Requests to the bulk store paging device for i/o are queued as they arrive for actual execution later. The queues kept are protected only by the global page table lock. That is, the code is not written to allow several processes to be accessing the queues simultaneously. Removal of the global lock could therefore result in errors in these queues unless a separate lock were added to protect the queues.

Unfortunately such assumptions are not usually documented. They are not discovered until such time as they result in a fatal system error of some type.

The second source of difficulty is the Multics implementation of quota. Actually, the problem is caused by the interaction of three features: quota, the hierarchical file structure, and dynamic segment growth. There are two numbers associated with each directory in Multics; the quota or maximum number of pages (disk records) the segments of the directory and inferior directories may occupy; and the records used, which is the actual current count of storage used. A directory may be specified as having no quota, in which case any quota placed on superior directories is the only constraint on the directory (e.g. if directory beta is immediately inferior to directory alpha and assuming alpha has a quota of 100 and beta has no quota, segments in beta can never occupy more than 100 pages).

The crucial factor is that Multics allows dynamic growth of segments. By merely referencing a non-existent page of a segment a process can create that page. Referencing the non-existent page causes a page fault, and page control creates a page of zeroes. At this point trouble arises, for this creation must be reflected in the records used count of the segment's parent directory. Thus, while the segment the page fault was on is locked, the segment's parent directory must also be locked to update the records used count. If the records used is less than the directory's quota, the creation is valid. However if the records used would now exceed the quota, the page may not be created and page control must notify the faulting process of an error. The situation is complicated if the

segment's parent directory does not have a quota limit, in which case the directory's parent must be checked, etc. until a superior directory is found that does have such a limit. At each step up the hierarchy, the directory (which is, of course, a segment) must be locked in order to increment its records used count. When a directory with a quota is found, the check can be made.

The difficulty arises in locking all the segments at the same time. They must be locked, because some very important per segment information is being changed. Since locking the directories is always from the bottom up (in terms of the hierarchy tree), there is no danger of a deadlock. But recall that the previously presented locking rules forbid a process from blocking itself with any segments locked. Hence if at any point, a process cannot lock a particular directory in its search for a directory with a quota limit, it must unlock all locked segments and block itself, starting over again when awakened.

Of course, when pages are deleted (e.g. by a truncate operation), the records used must also be updated in a like manner. Multics further complicates matters by always deleting pages of zeroes. That is, if a program or data segment has an entire page of zeroes anywhere, that page of zeroes is automatically deleted each time it is removed from main memory (and recreated upon next reference). This is done on the assumption creating the page is faster than reading and deleting is faster than writing, and that disk space will be saved. There is an impact of this decision on quota, in that such a page of zeroes is only charged against quota when actually in core.

The implementation of quota and the deletion of zero pages complicate

the page control algorithms, and especially the locking strategy, tremendously. Various simplifications are possible; for example: do not allow segments to grow dynamically, or allow them to grow dynamically but insist a maximum size be specified and always count that maximum size against the quota (thus no change is needed in records used when a page of zeroes is created). Explicit operations could be used to change the size of a segment instead of having page control do the work automatically. Unfortunately all such solutions have noticeable effects on the system, and would change its functionality. The issue of quota, its implementation and impact on the system, is quite complex and is still being studied.

# CHAPTER 6

## Conclusion

This thesis has presented a design for a system that implements a virtual memory using asynchronous, cooperating sequential processes. This design was demonstrated to offer significant potential advantages over other designs in terms of simplicity, modularity, system and user security, and degree of expandability.

The proposed system was built and tested on the Multics system. The implementation showed the feasibility of the design and the validity of the claimed advantages.

It is felt that the technique of exploiting parallelism in performing system tasks by implementing those tasks as several cooperating sequential processes is extremely important and powerful. That this method can be made to work in practice and lead to operating systems whose design is simpler and better structured is the most significant result of this thesis.

The Multics system offers several additional examples of places where a system process could be incorporated to perform tasks currently done by the user process. For example, section 2.1.4 mentioned that page tables are multiplexed among segments in the same fashion that page frames are

multiplexed between pages. Currently, when a segment is activated, if no page tables are available, the user process must execute a "replacement algorithm" which frees up a page table by deactivating some other segment. The similarity with page replacement is obvious, and a system process could be used to keep a free pool of page tables in the same fashion as the core manager does for page frames in the design presented here.

There is much that still can be done in this area. The test implementation could be greatly improved if the Multics scheduler were redesigned to truly implement system processes that could be scheduled without the considerable overhead of the current scheduler. The per-segment locking strategy proposed in section 4.4.2 would greatly improve the performance of multi-process page control in multiple processor systems.

Finally, it is hoped the success of the implementation reported here will encourage other such attempts, perhaps along the lines of Hoare's proposed system or Saxena and Bredt's system, to see if the difficulties concerning those systems mentioned in sections 3.3.2 and 3.3.3 can be overcome. It would be interesting to compare implementations of such systems, or newly proposed systems, with that given here.

Bibliography

[C174] Clark, David D., "An Input/Output Architecture for Virtual Memory Computer Systems", MIT Project MAC Technical Report TR-117, Cambridge, Mass., (January, 1974).

[Co69] Corbato, F. J., "A Paging Experiment with the Multics System", In Honor of P. M. Morse, M.I.T. Press, Cambridge, Mass., 1969, pp. 217-228.

[Da68] Daley, Robert C., and Dennis, Jack B., "Virtual Memory, Processes, and Sharing in MULTICS", Communications of the ACM, vol. 11, no. 5, (May, 1968), pp. 306-312.

[De66] Dennis, Jack B., and Van Horn, Earl C., "Programming Semantics for Multiprogrammed Computations", Communications of the ACM, vol. 9, no. 3, (March, 1966), pp. 143-155.

[Di68a] Dijkstra, Edsger W., "Co-operating Sequential Processes", Programming Languages, F. Genuys editor, Academic Press, New York, (1968), pp. 43-112.

[Di68b] Dijkstra, Edsger W., "The Structure of the 'THE' Multiprogramming System", Communications of the ACM, vol. 11, no. 5, (May, 1968), pp. 341-346.

[Gr75] Greenberg, Bernard S., and Webber, Steven H., "The Multics Multilevel Paging Hierarchy", paper presented at the IEEE INTERCON Conference, New York, New York, (April, 1975).

[Ha70] Hansen, Per Brinch, "The Nucleus of a Multiprogramming System", Communications of the ACM, vol. 13, no. 4, (April, 1970), pp. 238-241.

[Ho73] Hoare, C. A. R., "A Structured Paging System", The Computer Journal, vol. 16, no. 3, (August, 1973), pp. 209-215.

[Ho74] Hoare, C. A. R., "Monitors: An Operating System Structuring Concept", Communications of the ACM, vol. 17, no. 10, (October, 1974), pp. 549-557.

[Li72] Liskov, Barbara H., "The Design of the Venus Operating System", Communications of the ACM, vol. 15, no. 3, (March, 1972), pp. 144-149.

[Mu72] Murphy, D. L., "Storage Organization and Management in TENEX", AFIPS Conference Proceedings, 41, vol. 1, (Fall Joint Computer Conference, 1972), pp. 25-32.

[Sa75] Saxena, Ashok R., and Bredt, Thomas H., "A Structured Specification of a Hierarchical Operating System", Proceedings of 1975 Conference on Software Reliablity, (May, 1975), pp. 310-318.

[Sc72] Schell, Roger R., "Dynamic Reconfiguration in a Modular Computer System", MAC-TR-86, Project MAC, Cambridge, Mass., June, 1971.

[Sc73] Scherr, A. L., "Functional Structure of IBM Virtual Storage Operating Systems Part II: OS/VS2 Concepts and Philosophies", IBM Systems Journal, vol. 12, no. 4, (1973), pp. 382-400.

[Sc75] Schroeder, Michael D., "Engineering a Security Kernel for Multics", Operating Systems Review, vol. 9, no. 5, pp.25-32.

[Wh74] Wheeler, Jr., T. F., "OS/VS1 Concepts and Philosophies", IBM Systems Journal, vol. 13, no. 3, (1974), pp. 213-229.

## APPENDIX A

### Changes made to standard page control

| Changed Extensively | Modules Added |
|---|---|
| page_fault | page_fault_pll |
| post_purge | core_manager |
| pc | pd_manager |
| pc_abs | read |
| pc_contig | write |
| pc_wired | core_free_list |
| freecore | core_used_list |
| delete_pd_records | pd_free_list |
| wired_plm | pd_used_list |
| evict_page | utility |
| page_error | |
| initialize_dims | |
| init_sst | |
| pxss | |

| Changed Slightly | Modules Deleted |
|---|---|
| bulk_store_control | pd_util |
| disk_control | get_disk_meters |
| free_store | meter_disk |
| pc_trace | |
| wired_fim | |
| wired_shutdown | |

## APPENDIX B

### Components of Multi-process page control

| Name | Language | Source statements | Object length |
|---|---|---|---|
| page_fault | alm | 560 | 580 |
| page | alm | 28 | 116 |
| device_control | pl1 | 136 | 896 |
| bulk_store_control | alm | 369 | 386 |
| pc_trace | alm | 45 | 68 |
| free_store | alm | 133 | 138 |
| read | pl1 | 62 | 318 |
| write | pl1 | 192 | 956 |
| evict_page | pl1 | 39 | 142 |
| page_error | alm | 217 | 349 |
| post_purge | alm | 126 | 126 |
| get_disk_meters | pl1 | 12 | 22 |
| disk_control | pl1 | 247 | 1472 |
| pc_wired | pl1 | 70 | 312 |
| page_fault_pl1 | pl1 | 32 | 170 |
| pc | pl1 | 294 | 1740 |
| core_free_list | pl1 | 54 | 290 |
| core_used_list | pl1 | 49 | 232 |
| pd_free_list | pl1 | 53 | 230 |
| pd_used_list | pl1 | 40 | 180 |
| core_manager | pl1 | 282 | 1224 |
| pd_manager | pl1 | 179 | 724 |
| pc_contig | pl1 | 16 | 80 |
| utility | pl1 | 62 | 384 |
| quotaw | pl1 | 85 | 310 |
| thread | pl1 | 33 | 128 |
| get_ptrs_ | alm | 37 | 88 |
| pc_trace_pl1 | pl1 | 85 | 812 |
| pc_abs | pl1 | 17 | 160 |
| wired_plm | pl1 | 45 | 162 |
| delete_pd_records | pl1 | 75 | 416 |
| freecore | pl1 | 14 | 66 |
| | alm: | 1515 | |
| | pl1: | 2173 | |
| | | 3688 | 13,277 |

Components of standard page control

| Name | Language | Source statements | Object length |
|---|---|---|---|
| page_fault | alm | 1592 | 1616 |
| page | alm | 34 | 132 |
| device_control | pl1 | 118 | 134 |
| bulk_store_control | alm | 369 | 376 |
| pc_trace | alm | 45 | 252 |
| free_store | alm | 133 | 142 |
| evict_page | pl1 | 147 | 168 |
| page_error | alm | 376 | 614 |
| post_purge | alm | 145 | 146 |
| get_disk_meters | pl1 | 12 | 116 |
| disk_control | pl1 | 247 | 1478 |
| pc_wired | pl1 | 71 | 254 |
| pc | pl1 | 389 | 2144 |
| pc_contig | pl1 | 69 | 320 |
| quotaw | pl1 | 85 | 310 |
| thread | pl1 | 33 | 128 |
| get_ptrs_ | alm | 37 | 88 |
| pc_trace_pl1 | pl1 | 85 | 812 |
| pc_abs | pl1 | 69 | 328 |
| wired_plm | pl1 | 36 | 152 |
| delete_pd_records | pl1 | 111 | 546 |
| freecore | pl1 | 34 | 140 |
| meter_disk | pl1 | 72 | 68 |
| pd_util | alm | 394 | 402 |
| | alm: | 3423 | |
| | pl1: | 1280 | |
| | | 4703 | 10,866 |

# APPENDIX C

## Code from multi-process page control

The following code is taken directly from the implementation of the multi-process paging system implemented on Multics as described in Chapter 4. The procedure "page_fault_pll" is the code executed by the user process at page fault time; the procedures "core_manager" and "pd_manager" are the procedures executed by the core and pd manager processes respectively. While some code has been omitted (chiefly lower level subroutines and segment operations such as deactivation, trucation, wiring, etc.), no other changes have been made; all the programs listed were actually run on the Multics system.

The normal operation of the system is fairly straightforward for the most part and follows the ideas already presented. Page faults are the event which drive the entire system. On the occurrence of a page fault, the page fault code is invoked. After determining the page causing the fault, a call is made to allocate a free core page frame. The allocation procedure is ultimately responsible for driving the core manager process, for when the number of free page frames falls too low, a wakeup is sent to the core manager. On receiving this signal, the core manager selects an in-use page frame to be replaced and writes the page held in the page frame out of main memory. After waiting for the write operation to complete, the core manager adds the now free page frame to the free list.

The writing step may have several results, as an attempt will be made to write the page to the paging device. If a copy of the page is on the paging device and the page has not been modified, no write operation is necessary. But if the page is not yet on the paging device, or has been modified, a write must be performed. In the former case, a call must be made to allocate a paging device page frame, and this is the act which ultimately activates the paging device manager. When the allocation code notices too few paging device page frames are available, a wakeup signal is sent to the paging device manager. After receiving the wakeup the pd manager chooses a used paging device page frame to remove and performs a read write sequence if necessary (i.e. if the paging device copy has been modified with respect to the disk copy of the page, or if there is no disk copy). When the read write sequence is finished, the page frame is added to the free list.

Both the main memory replacement algorithm and the paging device replacement algorithm operate in a least recently used (LRU) fashion. The Multics hardware keeps modified and used bits in the page table word as mentioned in section 2.2.2. Each used list is implemented as a doubly linked circular list of entries, with a pointer to the least recently used item. This pointer identifies the first page frame examined when one is to be chosen for deallocation.

The main memory replacement algorithm examines the used list until a page whose used bit is off is found. Any page looked at during this search whose used bit is on has the bit turned off. Once such a page is found, it is a candidate for removal. (Certain other checks are made, for example to insure the page is not currently locked because it is

undergoing a read or write operation.) As pages are examined, the pointer to the least recently used item is advanced so that after the page to be removed is selected the page frame immediately following it in the list (i.e, the first page not looked at) becomes the least recently used page. When pages are faulted on and read in, they are placed immediately behind the page pointed to by the least recently used pointer; this makes them "most recently" used.

The paging device used list is managed in a similar way, however there are no used bits associated with paging device pages. Thus rather than searching for the first page on the paging device used list with a used bit off, the first page that is not currently also in main memory is selected for removal. The rationale for this decision is that the page is in use if in core, thus should not be removed from the paging device. Note since the page is in main memory, sooner or later it will be evicted from main memory, and the eviction will be made easier and faster if the page is already on the paging device. When a page is read from the paging device to satisfy a fault, that constitutes a use of the page, so it is moved to the most recently used position in the used list. Similarly, when pages are first written to the paging device, they are entered into the most recently used spot in the list.

The code that follows makes use of several data bases that are given rather cryptic names. The comments in the code often refer to these data bases. The list below explains the meaning of each abbreviation and the purpose of each data base. These data bases are defined by PL/1 structures. In the actual code, a statement of the form "%include sst;" causes the PL/1 structure declarations for the data base "sst" to be

included in the source file by the compiler at compilation time.

1. ast - active segment table

The active segment table contains one entry (an "aste", or "active segment table entry") for each active segment in the system. Each aste consists of all the page table words for pages of the segment plus the per segment information kept by page control such as segment length, quota, etc.

2. cmp - core map

Each page frame is described by a "core map entry" ("cme") in the core map. The cme contains the information associated with the page frame, e.g. a pointer to the page table word of the page allocated the page frame. The core used and free lists are merely linked lists of cme's.

3. pdmap - paging device map

Each paging device page frame is described by a "pdme", or "paging device map entry", in a manner analogous to the core map entries. Similarly, the pd used and free lists are linked lists of pdme's.

4. ptw - page table word

Each page of an active segment is described by a page table word which contains the current address of the copy of the page highest in the memory hierarchy; i.e. a core address if the page is in core, otherwise a paging device address or if the page is not on the paging device, a disk

address. Used and modified bits, a lock bit, and a fault tag are also kept in the page table word.

5. sst - system storage table

The sst is the primary page control data base. It contains not only the core map, the paging device map, and the active segment table, but also all other page control variables and constants such as pointers to the beginning of the various free and used lists, the global page table lock, etc. A large portion of the sst is also devoted to metering information (number of page faults, number of read write sequences performed, etc.).

```
page_fault_pli: procedure (rel_astep, rel_ptp) returns (bit (18) aligned);

      /* This routine acts as an interface between the ALM page_fault code and
         the pli read module.  It is called by the page_fault code when a read
         must be done on the page that "rel_ptp" points to ("rel_ptp" is offset
         of page table word in sst) of the segment whose ast entry is pointed
         to by "rel_astep".  This procedure allocates a block of core for the
         read and fills the information concerning the page required by
         the read module into the allocated cme.  When the read completes,
         the cme is put on the core used list.  This procedure also checks for
         the possibility of quota overflow, returning to the ALM page_fault
         code if a page is to be created and there is insufficient quota to
         do so.  The ALM page_fault code then signals the quota overflow.

         Written 8-1-75 by Andrew R. Huber for multi-process page control.     */

  declare rel_astep bit (18) aligned,                  /* Rel. ptr to aste of seg. of faulting page */
          rel_ptp bit (18) aligned,                    /* Rel. ptr to page table word of faulting page */
          pastep ptr,                                  /* Pointer to parent's aste */
          pds$quota_inhib fixed bin ext,               /* Non-zero means inhibit quota checking */
          pds$page_fault_data fixed bin ext;           /* Saved machine conditions */

  declare (addr, addrel, fixed, multiply, rel) builtin,
          core_free_list$allocate_cme entry returns (ptr),
          core_used_list$add_used_cme entry (ptr),
          read$page entry (ptr);

          %include sst;
          %include cmp;
          %include ptw;
          %include ast;
          %include mc;
```

```
sstp = addr (sst_seg$);                          /* Get pointer to sst */
astep = ptr (sstp, rel_astep);                   /* Get pointer to aste of faulting page */
ptp = ptr (sstp, rel_ptp);                       /* Get pointer to page's page table word */

if ptw.did = "0"b                                /* If faulted on a null page, */
     then if ^enough_quota ()                    /* see if have enough quota to create it */
          then return ("0"b);                    /* If not return and signal overflow */

cmep = core_free_list$allocate_cme ();           /* Allocate a block of core to the page */
cme.astep = rel_astep;                           /* Fill in info about page */
cme.ptwp = rel (ptp);

if ptw.did = "0"b                                /* If the page is null */
     then cme.diskadd = mptw.devadd;             /* use the null ptw address as the disk address */
     else if ptw.did = sst.pd_id                 /* Non-null, is it on the paging device? */
               then do;                          /* On paging device, so fill in cme.pdmep */
                    pdmep = addrel (sst.pdmap, multiply (fixed (ptw.add, 18), sst.pdsize, 18, 0));
                    cme.pdmep = rel (pdmep);     /* Compute pointer to pdme */
                    cme.diskadd = pdme.diskadd;  /* Fill in disk address from pdme */
                    sst.pd_page_faults = sst.pd_page_faults + 1;  /* Meter page faults from pd */
                    end;
               else cme.diskadd = mptw.devadd;   /* Not on pd, disk address in page table word */

call read$page (cmep);                           /* Waiting for the i/o to complete is done */
                                                 /* by code in page_fault since running on prds */
call core_used_list$add_used_cme (cmep);         /* Put cme on used list as most recently used */

return (rel (cmep));                             /* Return rel. ptr to cme assigned */
```

```
core_manager: procedure (unwanted_pointer);

      /*      This is the driving routine for the core manager process.  The
         core manager process is an H-proc (as implemented by Mabee; see RFC-66)
         created by initialize_dims early in initialization.  The argument is
         a pointer passed by create_supervisor_task when it starts the H-proc
         running by calling this routine; in this case it is a null pointer. The
         function of the core manager is to manage the core free and used lists,
         performing all duties involving operations on these lists.  The basic
         algorithm is to loop until a wakeup is received from some process
         requesting something be done.  The core manager discovers what the
         request was by comparing the values of the "_signals" variables with
         the values of the corresponding "_done" variable.  A requestor
         adds one to the "_signals" variable corresponding to the task he wishes
         performed; the core manager adds one to the corresponding "_done"
         variable when he has performed the task; thus when the two are equal there
         are no requests of that type outstanding.  Note once started, the core
         manager completes all outstanding requests of all types.  Only
         if there is no work to do does the core manager block until another
         wakeup is received.  The standard wait and notify primitives are used
         for inter-process communication; the core manager is signalled by
         executing "call pxss$notify ("cmev")".

         Written 8-6-75 by Andrew R. Huber for multi-process page control.          */

  declare (i,                                                  /* Loop index */
          j,                                                   /* Counter */
          try,                                                 /* Loop index, no. of times tried config sig. */
          base,                                                /* Index in core map of first cme in a port */
          size,                                                /* Number of cme's in current port */
          port,                                                /* Number of port currently being looked at */
          needed,                                              /* No. of contiguous abs.wirable pages needed */
          last_i,                                              /* Saved value of loop index */
          old_abs_wired,                                       /* Saved value of sst.abs_wired_count */
          records,                                             /* Number of records truncated */
          no_pages,                                            /* Number of pages to be operated on */
          first_page,                                          /* First page in seg. to be operated on */
          first_core,                                          /* Index of first cme to use for abs wiring */
          last_state,                                          /* Value of sst.core_manager_signals to wait on */
          last_page) fixed bin,                                /* Last page to be operated on */
          devadd bit (22),                                     /* Device address to be returned to free pool */
          device_did bit (4) defined (devadd) position (19),   /* Device id portion of device address */
          int_call bit (1) aligned,                            /* Flag allowing interrupts while abs wiring */
          old_gtpd bit (1) aligned,                            /* Saved value of aste.gtpd for cleanup entry */
          old_cmep ptr,                                        /* Pointer to an old cme */
          unwanted_pointer ptr,                                /* Arg ptr passed when H-proc started up */
          oldmask fixed bin (71),                              /* Old mask value needed by pmut routines */
          wind bit (18) aligned,                               /* Wait event returned by pageSevict */
          ind bit (18) aligned,                                /* Page event to wait on */
          scs$sys_level fixed bin (71) external,               /* Sys level mask */
          null_devadd_not_in_core bit (36) aligned init ("000000000000000000000000000000000001"b) int static;
```

```
declare (addr, addrel, bit, divide, fixed, min, multiply, ptr) builtin,
        core_free_list$add_free_cme entry (ptr),
        core_free_list$remove_free_cme entry (ptr),
        core_used_list$add_used_cme entry (ptr),
        core_used_list$remove_used_cme entry (ptr),
        core_used_list$select_core entry returns (ptr),
        iobm$force_timeouts entry,
        page$cam entry,
        page$copy entry (bit (18) aligned, bit (18) aligned),
        page$deposit entry (bit (22) aligned),
        page$evict entry (ptr, bit (18) aligned),
        page$pwait entry (bit (18) aligned),
        page$lock_ptl entry,
        page$unlock_ptl entry,
        pmut$set_mask entry (fixed bin (71), fixed bin (71)),
        pxss$block_on_event entry (fixed bin, fixed bin, fixed bin (71)),
        pxss$notify entry (char (4)),
        read$page entry (ptr),
        syserr entry options (variable),
        utility$move_page entry (ptr, ptr),
        write$page entry (ptr);

        %include sst;
        %include cmp;
        %include ast;
        %include ptw;
        %include null_addresses;
        %include controller_data;
```

```
sstp = addr (sst_seg$);

call pmut$set_mask (scs$sys_level, oldmask);      /* Make sure core_manager always runs masked */
last_state = sst.core_manager_signals;            /* Wait until someone signals core manager */
call pxss$block_on_event (sst.core_manager_signals, last_state, 3e6);

do while ("1"b);                                  /* Main loop; repeat forever. */

     last_state = sst.core_manager_signals;       /* Get counter value for waiting on later */

     do while (sst.free_cmes < sst.max_free_cmes);
          call get_core;                          /* Free core until maximum reached */
          end;

     do while (sst.cm_cleanups_done < sst.cm_cleanup_signals);        /* Do any cleanups */
          call cm_cleanup;
          sst.cm_cleanups_done = sst.cm_cleanups_done + 1;
          end;

     do while (sst.cm_truncates_done < sst.cm_truncate_signals);      /* Do any truncates */
          call cm_truncate;
          sst.cm_truncates_done = sst.cm_truncates_done + 1;
          end;

     do while (sst.add_cores_done < sst.add_core_signals);  /* Any core to add? */
          call add_core;
          sst.add_cores_done = sst.add_cores_done + 1;
          end;

     do while (sst.remove_cores_done < sst.remove_core_signals);      /* Any core to remove? */
          call remove_core;
          sst.remove_cores_done = sst.remove_cores_done + 1;
          end;

     do while (sst.cm_contigs_done < sst.cm_contig_signals);  /* Any contiguous page requests? */
          call get_contig_core;
          sst.cm_contigs_done = sst.cm_contigs_done + 1;
          end;

     call pxss$block_on_event (sst.core_manager_signals, last_state, 3e6);  /* Wait for more work */

     end;
```

```
get_core: procedure;                                    /* Internal procedure to free up one more */
                                                        /* block of core and add it to free list */
          call page$lock_pt();                          /* First lock the page table lock */

          cmep = core_used_list$select_core ();         /* Call replacemanet algorithm to find core */
                                                        /* block to be freed */
          call core_used_list$remove_used_cme (cmep);   /* Remove selected core block from used list */

          ptp = ptr (cmep, cme.ptwp);                   /* Get pointer to page table word */

          call write$page (cmep);                       /* Write out the contents of the core block */

          if ptw.os                                     /* If page out of service, i.e. i/o still */
               then call page$pwait (rel (ptp));        /* going on, then wait for it to finish */

          call core_free_list$add_free_cme (cmep);      /* Put the now free core block on free list */

          call page$unlock_pt();                        /* Unlock */

          call pxss$notify ("core");                    /* Notify anyone waiting for more core */

          return;

          end get_core;
```

```
core_used_list: procedure;

      /* This module contains all entries needed to maintain the list of cme's that are currently
         in use.  The cme's are organized into a circular, doubly-linked list in order of last
         use.  The pointer to the head of the list, sst.usedp, points to the least recently used
         entry.  Since the list is circular, the most recently used entry immediately preceeds the
         least recently used cme.  The number of entries in the list is maintained in the variable
         sst.used_cmes.                                                                          */

         /* Written 8-5-75 by Andrew R. Huber for multi-process page control. */

  declare i fixed bin,                                    /* Loop index */
          cme_ptr ptr,                                    /* Parameter, cme to be added or removed */
          up ptr,                                         /* Pointer to cme at head of used list */
          many fixed bin init (100000) int static;        /* Times to try replacement alg. */

  declare (null, rel, ptr) builtin,
          syserr entry options (variable);

          %include sst;
          %include cmp;
          %include ptw;

add_used_cme: entry (cme_ptr);                            /* This entry adds the cme pointed to by */
                                                          /* cmep to the list in the most recently */
                                                          /* used position */
          cmep = cme_ptr;
          sstp = addr (sst_seg$);

          if sst.used_cmes = 0                            /* If the used list is currently empty */
               then do;                                   /* make this the only entry in the list. */
                    sst.usedp = rel (cmep);               /* Set ptr to head of list to this entry */
                    cme.fp, cme.bp = rel (cmep);          /* Since this is only entry, set pointers */
                    end;                                  /* to next and last entry to itself */
               else do;                                   /* Other entries in the used list */
                    cme.fp = sst.usedp;                   /* Thread this entry in before first entry */
                    up = ptr (sstp, sst.usedp);           /* Get pointer to head of list */
                    cme.bp = up -> cme.bp;                /* Copy back pointer from cme at head of list */
                    ptr (sstp, cme.bp) -> cme.fp = rel (cmep);        /* Make last entry point to this cme */
                    up -> cme.bp = rel (cmep);            /* Make next entry point back to this cme */
                    end;

          sst.used_cmes = sst.used_cmes + 1;              /* Increment count of cme's in use */

          return;                                         /* End of add_used_cme entry */
```

```
remove_used_cme: entry (cme_ptr);                    /* This entry removes the cme pointed to by */
                                                     /* cmep from the used list */
          cmep = cme_ptr;
          sstp = addr (sst_seg$);


                                                     /* Now thread the cme out -- note these */
          ptr (sstp, cme.fp) -> cme.bp = cme.bp;     /* steps work even if there is only one */
          ptr (sstp, cme.bp) -> cme.fp = cme.fp;     /* entry  in the list, since in that case */
                                                     /* cme.fp and cme.bp point to themself */
          sst.used_cmes = sst.used_cmes - 1;         /* Decrement count of cme's on used list */

          if sst.used_cmes = 0                       /* If used list now empty, reset pointer to */
               then sst.usedp = "0"b;                /* the list to "0"b to indicate this */
               else if sst.usedp to = rel (cmep)     /* If still entries in list, set sst.usedp to */
                         then sst.usedp =  cme.fp;   /* next entry if entry at head of list removed */


          return;                                    /* End of remove_used_cme entry */
```

```
select_core: entry returns (ptr);                       /* This entry implements the page replacement */
                                                        /* algorithm, selecting from the core used list */
                                                        /* a cme whose page will be removed from core. */
                                                        /* The used list is scanned for an eligible */
                                                        /* page whose used bit is off. Eligible pages */
                                                        /* are those not wired, not undergoing i/o, and */
                                                        /* not being removed. */
     sstp = addr (sst_seg$);
     cmep = ptr (sstp, sst.usedp);                      /* Get pointer to head of used list */

     do i = 1 to many;                                  /* Look at many cme's before failing */

          sst.steps = sst.steps + 1;                    /* Add one to total step count */

          ptp = ptr (sstp, cme.ptwp);                   /* Get pointer to page table word for this cme */

          if cme.rws                                    /* Is page undergoing rws ? */
               then sst.skipspd = sst.skipspd + 1;      /* Yes, skip it, keeping count of why skipped */
               else if ptw.wired                        /* Not undergoing rws; is it wired ? */
                         then sst.skipw = sst.skipw + 1; /* Wired, so skip, counting as wired skip */
                         else if cme.removing           /* Not wired; is it being deconfigured ? */
                                   then sst.skipr = sst.skipr + 1;        /* Yes, count removing skip */
                                   else if ptw.os       /* Otherwise out of service for i/o ? */
                                             then sst.skipos = sst.skipos + 1; /* Count os skip */
                                             else if ptw.phu     /* Almost -- has page been used ? */
                                                       then do;  /* Yes, so skip it turning off */
                                                            ptw.phu = "0"b; /* the used bit */
                                                            sst.skipu = sst.skipu + 1;
                                                            end;
                                                       else do;            /* WIN! Claim this cme */
                                                            sst.usedp = cme.fp;
                                                            return (cmep);
                                                            end;

                                                       /* If reach this point, the cme was skipped */
          cmep = ptr (sstp, cme.fp);                    /* Advance pointer to next entry */

          end;

     call syserr (1, "cm: impossible to free core.");   /* Crash if can't free a cme after many tries */

     return (null);                                     /* End of select_core entry */


     /* This ends the core_used_list module */

     end core_used_list;
```

```
core_free_list: procedure;

      /* This module contains all entries needed to maintain the list of cme's that are currently
         free. The cme's are organized into a circular, doubly-linked list, whose head is pointed to
         by sst.freep. Normally entries are added to and allocated from the head of the list,
         since all free cme's are equal. A cme being removed from service, i.e. deconfigured,
         may be removed from anywhere in the list however. The number of entries on the list
         is kept in the variable sst.free_cmes. The core manager tries to keep this number >=
         sst.min_free_cmes at all times, thus when the count drops below this a signal is sent
         to the core manager to free some more.                                              */

         /* Written 8-5-75 by Andrew R. Huber for multi-process page control. */

  declare fptr ptr;                                    /* Pointer to head of free list */

  declare (addr, null, rel, ptr) builtin,
          page$lock_pti entry,
          page$unlock_pti entry,
          pxss$addevent entry (char (4)),
          pxss$delevent entry (char (4)),
          pxss$event entry (fixed bin),
          pxss$wait entry;

          %include sst;
          %include cmp;
```

```
add_free_cme: entry (cmep);                              /* This entry adds the cme pointed to by */
                                                         /* cmep to the head of the free list */
          sstp = addr (sst_seg$);

          cme.ptwp, cme.astep, cme.pdmep = (18)"0"b;     /* Blank out pointers in the cme */
          cme.diskadd = (22)"0"b;                        /* and the disk address */

          if sst.free_cmes = 0                           /* If the free list is currently empty */
               then do;                                  /* make this the only entry in the list. */
                    sst.freep = rel (cmep);              /* Set ptr to head of list to this entry */
                    cme.fp, cme.bp = rel (cmep);         /* Since this is only entry, set pointers */
                    end;                                 /* to next and last entry to itself */
               else do;                                  /* Other entries in the free list, so */
                    cme.fp = sst.freep;                  /* make this entry the new head of the list */
                    fptr = ptr (sstp, sst.freep);        /* Get pointer to cme at head of list */
                    cme.bp = fptr -> cme.bp;             /* Note this works even if only one entry */
                    fptr -> cme.bp = rel (cmep);         /* since in that case cme.fp and cme.bp */
                    ptr (sstp, cme.bp) -> cme.fp = rel (cmep);       /* point to themself */
                    sst.freep = rel (cmep);              /* Finally, make this entry new head of list */
                    end;

          sst.free_cmes = sst.free_cmes + 1;             /* Increment count of cme's in use */
          if sst.free_cmes = sst.max_free_cmes           /* Meter times ceiling hit */
               then sst.times_max_free_cmes = sst.times_max_free_cmes + 1;

          return;                                        /* End of add_free_cme entry */


remove_free_cme: entry (cmep);                           /* This entry removes the cme pointed to by */
                                                         /* cmep from the free list. This is a special */
                                                         /* entry for use when removing a specific cme */
                                                         /* from the free list, e.g. deconfiguring it */
          sstp = addr (sst_seg$);

                                                         /* Now thread the cme out -- note these */
          ptr (sstp, cme.fp) -> cme.bp = cme.bp;         /* steps work even if there is only one */
          ptr (sstp, cme.bp) -> cme.fp = cme.fp;         /* entry in the list, since in that case */
                                                         /* cme.fp and cme.bp point to themself */
          sst.free_cmes = sst.free_cmes - 1;             /* Decrement count of cme's on free list */

          if sst.free_cmes = 0                           /* If free list now empty, reset pointer to */
               then sst.freep = "0"b;                    /* the list to "0"b to indicate this */
               else if sst.freep to = rel (cmep)         /* If still entries in list, set sst.freep to */
                         then sst.freep = cme.fp;        /* next entry if entry at head of list removed */


          return;                                        /* End of remove_free_cme entry */
```

```
allocate_cme: entry () returns (ptr);                    /* This entry returns a pointer to a free */
                                                         /* cme. If there are none, it signals the */
                                                         /* core manager to free some and waits until */
                                                         /* he does so. If the number of free cme's  */
                                                         /* drops below the allowed number, the core */
                                                         /* manager is also signalled */
     sstp = addr (sst_seg$);

     do while (sst.free_cmes = 0);                       /* If there are no cme's on free list,  */
                                                         /* tell core manager to free some */
          sst.times_out_of_cmes = sst.times_out_of_cmes + 1;     /* Meter times out of cmes */
          call pxss$addevent ("core");                   /* Set up wait event */
          call pxss$event (sst.core_manager_signals);    /* Tell core manager to get busy */
          if sst.free_cmes = 0                           /* If still no free core */
               then do;                                  /* wait until there is some */
                    call page$unlock_pti;                /* Must unlock pti before waiting */
                    call pxss$wait;                      /* Wait for core manager to free some core */
                    call page$lock_pti;                  /* Re-lock pti before continuing */
                    end;
               else call pxss$delevent ("core");         /* If core now though, just continue */
          end;

     fptr = ptr (sstp, sst.freep);                       /* Get pointer to cme at head of free list */
     ptr (sstp, fptr -> cme.fp) -> cme.bp = fptr -> cme.bp;  /* Remove the entry at head of the list */
     ptr (sstp, fptr -> cme.bp) -> cme.fp = fptr -> cme.fp;

     sst.free_cmes = sst.free_cmes - 1;                  /* Decrement count of entries on free list */
     sst.needc = sst.needc + 1;                          /* Increment count of core blocks needed */

     if sst.free_cmes = 0                                /* If we removed the last entry from free list */
          then sst.freep = "0"b;                         /* set pointer to head of list to "0"b */
          else sst.freep = fptr -> cme.fp;               /* If not, set to entry following that removed */

     if sst.free_cmes < sst.min_free_cmes                /* If we've fallen below the desired minimum */
          then do;                                       /* free some more */
               sst.cm_need_core_signals = sst.cm_need_core_signals + 1;    /* Signal too few free cmes */
               call pxss$event (sst.core_manager_signals);       /* Wake up core manager */
               end;

     return (fptr);                                      /* Return the pointer to the free cme */
                                                         /* End of allocate_cme entry */

     /* This ends the core_free_list module */


     end core_free_list;
```

```
pd_manager: procedure (unwanted_pointer);

      /*     This is the driving routine for the pd manager process.  The
         pd manager process is an H-proc (as implemented by Habee; see RFC-66)
         created by initialize_dims early in initialization.  The argument is
         a pointer passed by create_supervisor_task when it starts the H-proc
         running by calling this routine; in this case it is a null pointer.  The
         function of the pd manager is to manage the pd free and used lists,
         performing all duties involving operations on these lists.  The basic
         algorithm is to loop until a wakeup is received from some process
         requesting something be done.  The pd manager discovers what the
         request was by comparing the values of the "_signals" variables with
         the values of the corresponding "_done" variable.  A requester
         adds one to the "_signals" variable corresponding to the task he wishes
         performed; the pd manager adds one to the corresponding "_done"
         variable when he has performed the task; thus when the two are equal there
         are no requests of that type outstanding.  Note once started, the pd
         manager completes all outstanding requests of all types.  Only
         if there is no work to do does the pd manager block until another
         wakeup is received.  The standard wait and notify primitives are used
         by processes waiting on a pd manager event, however the pd manager
         itself uses Habee's block_on_event and event primitives, therefore
         to wake up the pd manager a "call pxss$event (sst.pd_manager_signals)"
         is executed.

         Written 8-6-75 by Andrew R. Huber for multi-process page control.      */

  declare (i,                                          /* Loop index */
          j,                                           /* Counter */
          records,                                     /* Number of records truncated */
          last_state,                                  /* Value of sst.pd_manager_signals to wait on */
          first,                                       /* First page in seg. to be operated on */
          last) fixed bin,                             /* Last page to be operated on */
          time fixed bin (24),                         /* Current time as returned by clock_ */
          oldmask fixed bin (71),                      /* Saved mask used by pout routines */
          unwanted_pointer ptr,                        /* Extra arg passed by create_supervisor_task */
          ind bit (18) aligned,                        /* Page event to wait on */
          based_pdme (4*3) fixed bin based (pdmep),    /* A pdme entry as four fixed bin words */
          scs$sys_level fixed bin (71) external,       /* Sys level mask */
          update_interval fixed bin (71) int static init (1000000);   /* Length of time between pd map */
                                                       /* updates, in milliseconds */

  declare (addr, addrel, bit, divide, fixed, null, ptr, substr) builtin,
          clock_ entry returns (fixed bin (71)),
          core_free_list$allocate_cme entry returns (ptr),
          pd_free_list$add_free_pdme entry (ptr),
          pd_free_list$remove_free_pdme entry (ptr),
          pd_used_list$remove_used_pdme entry (ptr),
          pd_used_list$select_pd_record entry returns (ptr),
          page$cam entry,
          page$pwait entry (bit (18) aligned),
```

```
page$lock_ptl entry,
page$unlock_ptl entry,
pmut$set_mask entry (fixed bin (71), fixed bin (71)),
pxss$block_on_event entry (fixed bin, fixed bin, fixed bin (71)),
pxss$notify entry (char (4)),
write$pdmap entry,
write$rws entry (ptr, ptr);

%include sst;
%include cmp;
%include ptw;
%include ast;
```

```
sstp = addr (sst_seg$);
pdmap = sst.pdmap;                                  /* Get useful pointers */

call pmut$set_mask (scs$sys_level, oldmask);        /* Make sure pd_manager always runs masked */
last_state = sst.pd_manager_signals;                /* Wait until first pd manager event */
call pxss$block_on_event (sst.pd_manager_signals, last_state, 3e6);

do while ("1"b);                                    /* Main loop; repeat forever. */

     last_state = sst.pd_manager_signals;           /* Save counter value for waiting or later */

     do while (sst.pd_free < sst.max_free_pdmes);
          call get_pd_record;
          end;

     do while (sst.pd_cleanups_done < sst.pd_cleanup_signals);        /* Do any cleanups */
          call pd_cleanup;
          sst.pd_cleanups_done = sst.pd_cleanups_done + 1;
          end;

     do while (sst.pd_truncates_done < sst.pd_truncate_signals);
          call pd_truncate;
          sst.pd_truncates_done = sst.pd_truncates_done + 1;
          end;

     do while (sst.add_pd_records_done < sst.add_pd_records_signals); /* Any pdmes to add? */
          call add_pd_records;
          sst.add_pd_records_done = sst.add_pd_records_done + 1;
          end;

     do while (sst.remove_pd_records_done < sst.remove_pd_records_signals);  /* Remove pdmes? */
          call remove_pd_records;
          sst.remove_pd_records_done = sst.remove_pd_records_done + 1;
          end;

     time = clock_ ();                              /* See if time to update pd map again */
     if time - sst.last_update > update_interval    /* If longer than update interval */
          then do;                                  /* then write out pd map again */
               sst.last_update = time;              /* Save time written out */
               call write$pdmap;
               sst.rws_time_start = sst.rws_time_start + clock_() - time;       /* Count time as */
               end;                                 /* rws start time for comparison with cur. sys */

     call pxss$block_on_event (sst.pd_manager_signals, last_state, 3e6);

     end;
```

```
get_pd_record: procedure;                                   /* Internal procedure to free up one more */
                                                            /* pd record and add it to free list */
        call page$lock_pt();                                /* First lock the page table lock */

        sst.rws_time_temp = clock_();

        pdmep = pd_used_list$select_pd_record ();           /* Call replacemanet algorithm to find pd */
                                                            /* block to be freed */
        call pd_used_list$remove_used_pdme (pdmep);         /* Remove selected pd record from free list */

        call write$rws (sst.rws_cmep, pdmep);               /* Write out the contents of the pd record */

        time = clock_();

        call pd_free_list$add_free_pdme (pdmep);            /* Place the now free pd block on the free list */

        call page$unlock_pt();                              /* Unlock */

        call pxss$notify ("prec");                          /* Notify anyone waiting for a pd record */

        sst.rws_time_done = sst.rws_time_done + clock_() - time;

        return;

        end get_pd_record;
```

```
pd_used_list: procedure;

      /* This module contains all entries needed to maintain the list of pdme's that are currently
         in use.  The pdme's are organized into a circular, doubly-linked list in order of last
         use.  The pointer to the head of the list, sst.pdusedp, points to the least recently used
         entry.  Since the list is circular, the most recently used entry immediately preceeds the
         least recently used pdme.  The number of entries in the list is maintained in the variable
         sst.pd_used.                                                                          */

     /* Written 8-5-75 by Andrew R. Huber for multi-process page control. */

  declare i fixed bin,                                  /* Loop index */
          pdmep_ ptr,                                   /* Pointer to pdme of interest */
          up ptr,                                       /* Pointer to pdme at head of used list */
          many fixed bin init (100000);                 /* Number of times to loop looking for free */
                                                        /* pdme before giving up */
  declare (addr, rel, ptr) builtin,
          syserr entry options (variable);


          %include sst;
          %include cmp;
```

```
add_used_pdme: entry (pdmep_);                          /* This entry adds the pdme pointed to by */
                                                        /* pdmep to the list in the most recently */
                                                        /* used position */
          sstp = addr (sst_seg$);
          pdmep = pdmep_;                               /* Copy argument */

          if sst.pd_used = 0                            /* If the used list is currently empty */
               then do;                                 /* make this the only entry in the list. */
                    sst.pdusedp = rel (pdmep);          /* Set ptr to head of list to this entry */
                    pdme.fp, pdme.bp = rel (pdmep);     /* Since this is only entry, set pointers */
                    end;                                /* to next and last entry to itself */
               else do;                                 /* Other entries in the used list */
                    up = ptr (sstp, sst.pdusedp);       /* Get pointer to head of list */
                    pdme.fp = sst.pdusedp;              /* Thread this entry in before entry */
                    pdme.bp = up -> pdme.bp;            /* entry pointed to by up, i.e. */
                    ptr (sstp, pdme.bp) -> pdme.fp = rel (pdmep);     /* in most recently used spot */
                    up -> pdme.bp = rel (pdmep);
                    end;

          sst.pd_used = sst.pd_used + 1;                /* Increment count of pdme's in use */

          return;                                       /* End of add_used_pdme entry */

remove_used_pdme: entry (pdmep_);                       /* This entry removes the pdme pointed to by */
                                                        /* pdmep from the used list */
          sstp = addr (sst_seg$);
          pdmep = pdmep_;                               /* Copy argument */

                                                        /* Now thread the pdme out -- note these */
          ptr (sstp, pdme.fp) -> pdme.bp = pdme.bp;     /* steps work even if there is only one */
          ptr (sstp, pdme.bp) -> pdme.fp = pdme.fp;     /* entry  in the list, since in that case */
                                                        /* pdme.fp and pdme.bp point to themself */
          sst.pd_used = sst.pd_used - 1;                /* Decrement count of pdme's on used list */

          if sst.pd_used = 0                            /* If used list now empty, reset pointer to */
               then sst.pdusedp = "0"b;                 /* the list to "0"b to indicate this */
               else if sst.pdusedp = rel (pdmep)        /* If still entries in list, set pdusedp to */
                         then sst.pdusedp = pdme.fp;    /* next entry if entry at head of list removed */

          return;                                       /* End of remove_used_pdme entry */
```

```
select_pd_record: entry returns (ptr);                  /* This entry implements the page replacement  */
                                                        /* algorithm for pages on the paging device. */
                                                        /* A pointer to the next pdme to be freed is */
                                                        /* returned.  The least recently used pdme */
                                                        /* whose page is not in core is selected. */
     sstp = addr (sst_segs);
     pdmep = ptr (sstp, sst.pdusedp);                   /* and to least recently used pdme */
     sst.pdusedp = pdme.fp;                             /* Reset head of used list to next pdme */

     do i = 1 to many;                                  /* Loop through many entries */

          sst.pd_steps = sst.pd_steps + 1;              /* Up total of steps around used list */

          if pdme.incore                                /* If this entry's page is incore then skip */
               then sst.pd_skips_incore = sst.pd_skips_incore + 1;  /* It but count as in core skip */
               else if pdme.rws                         /* Page not in core, is pdme undergoing rws? */
                         then sst.pd_skips_rws = sst.pd_skips_rws + 1;     /* Yes, count as rws skip */
                         else return (pdmep);           /* No, so claim this pdme */

          pdmep = ptr (sstp, pdme.fp);                  /* Advance pointer to look at next entry */
          sst.pdusedp = pdme.fp;                        /* and move ptr to least recently used pdme */

          end;

     call syserr (1, "pd: impossible to free pdme.");   /* Crash if can't find pdme after many tries */
                                                        /* End of the select_pd_record entry */


     /* This ends the pd_used_list module */


     end pd_used_list;
```

```
pd_free_list: procedure;

      /* This module contains all entries needed to maintain the list of pdme's that are currently
         free. The pdme's are organized into a circular, doubly-linked list, whose head is pointed to
         by sst.pdfreep.  Normally entries are added to and allocated from the head of the list,
         since all free pdme's are equal.  A pdme being removed from service, i.e. deconfigured,
         may be removed from anywhere in the list however.  The number of entries on the list
         is kept in the variable sst.pd_free.  The pd manager tries to keep this number >=
         sst.min_free_pdmes at all times, thus when the count drops below this a signal is sent
         to the pd manager to free some more.                                                  */

      /* Written 8-5-75 by Andrew R. Huber for multi_process page control. */

  declare fptr ptr,                                   /* Pointer to first pdme in free list */
          last_state fixed bin (35);                  /* Saved counter value for waiting on */

  declare (addr, rel, ptr) builtin,
          page$lock_ptl entry,
          page$unlock_ptl entry,
          pxss$addevent entry (char (4)),
          pxss$delevent entry (char (4)),
          pxss$event entry (fixed bin),
          pxss$wait entry;

          %include sst;
          %include cmp;
```

```
add_free_pdme: entry (pdmep);                          /* This entry adds the pdme pointed to by */
                                                       /* pdmep to the head of the free list */
          sstp = addr (sst_seg$);

          pdme_bits = (144)"0"b;                       /* Zero entry before adding to free list */

          if sst.pd_free = 0                           /* If the free list is currently empty */
               then do;                                /* make this the only entry in the list. */
                    sst.pdfreep = rel (pdmep);         /* Set ptr to head of list to this entry */
                    pdme.fp, pdme.bp = rel (pdmep);    /* Since this is only entry, set pointers */
                    end;                               /* to next and last entry to itself */
               else do;                                /* Other entries in the free list */
                    fptr = ptr (sstp, sst.pdfreep);    /* Get pointer to head of free list */
                    pdme.fp = sst.pdfreep;             /* Make this entry the new head of the list */
                    pdme.bp = fptr -> pdme.bp;         /* Note this works even if only one entry */
                    fptr -> pdme.bp = rel (pdmep);     /* since in that case pdme.fp and pdme.bp */
                    ptr (sstp, pdme.bp) -> pdme.fp = rel (pdmep);     /* point to themself */
                    sst.pdfreep = rel (pdmep);         /* Finally, make this entry new head of list */
                    end;

          sst.pd_free = sst.pd_free + 1;               /* Increment count of pdme's in use */
          if sst.pd_free = sst.max_free_pdmes          /* Meter times ceiling hit */
               then sst.times_max_free_pdmes = sst.times_max_free_pdmes + 1;

          return;                                      /* End of add_free_pdme entry */



remove_free_pdme: entry (pdmep);                       /* This entry removes the pdme pointed to by */
                                                       /* pdmep from the free list. This is a special */
                                                       /* entry for use when removing a specific pdme */
                                                       /* from the free list, e.g. deconfiguring it */
          sstp = addr (sst_seg$);

                                                       /* Now thread the pdme out -- note these */
          ptr (sstp, pdme.fp) -> pdme.bp = pdme.bp;    /* steps work even if there is only one */
          ptr (sstp, pdme.bp) -> pdme.fp = pdme.fp;    /* entry  in the list, since in that case */
                                                       /* pdme.fp and pdme.bp point to themself */
          sst.pd_free = sst.pd_free - 1;               /* Decrement count of pdme's on free list */

          if sst.pd_free = 0                           /* If free list now empty, reset pointer to */
               then sst.pdfreep = "0"b;                /* the list to "0"b to indicate this */
               else if sst.pdfreep = rel (pdmep)       /* If still entries on list, set sst.pdfreep to */
                         then sst.pdfreep = pdme.fp;   /* next entry if entry at head of list removed */


          return;                                      /* End of remove_free_pdme entry */
```

```
allocate_pdme: entry () returns (ptr);                    /* This entry returns a pointer to a */
                                                          /* free pdme. If there are none, it signals */
                                                          /* the pd manager to free some and waits until */
                                                          /* he does so. If the number of free pdme's  */
                                                          /* drops below the allowed number, the core */
                                                          /* manager is also signalled */

     sstp = addr (sst_seg$);

     do while (sst.pd_free = 0);                          /* If there are no pdme's on free list,  */
                                                          /* tell pd manager to free some */
          sst.times_out_of_pdmes = sst.times_out_of_pdmes + 1;   /* Meter times no free pdmes */
          call pxss$addevent ("prec");                    /* Get set to wait for more pd records */
          call pxss$event (sst.pd_manager_signals);       /* Wake up the pd manager to free some */
          if sst.pd_free = 0                              /* If still none free */
               then do;                                   /* block until some free */
                    call page$unlock_pti;                 /* Must unlock pti before waiting */
                    call pxss$wait;                       /* Wait for completion signal from pd manager */
                    call page$lock_pti;                   /* Re-lock pti before continuing */
                    end;
               else call pxss$delevent ("prec");          /* But if some now don't bother waiting */
          end;

     fptr = ptr (sstp, sst.pdfreep);                      /* Get pointer to pdme at head of list */
     ptr (sstp, fptr -> pdme.fp) -> pdme.bp = fptr -> pdme.bp;   /* Remove entry at head of the list */
     ptr (sstp, fptr -> pdme.bp) -> pdme.fp = fptr -> pdme.fp;

     sst.pd_free = sst.pd_free - 1;                       /* Decrement count of entries on free list */
     sst.pd_needed = sst.pd_needed + 1;                   /* Up cumulative total of pdmes allocated */

     if sst.pd_free =0                          /* If we removed the last entry from free list */
          then sst.pdfreep = "0"b;                        /* set pointer to head of list to "0"b */
          else sst.pdfreep = fptr -> pdme.fp;             /* If not, set to entry following that removed */

     if sst.pd_free < sst.min_free_pdmes                  /* If we've fallen below the desired minimum */
          then do;
               sst.pd_need_pdmes_signals = sst.pd_need_pdmes_signals + 1;
               call pxss$event (sst.pd_manager_signals);        /* Wake up the pd manager */
               end;                                       /* to free some more */

     return (fptr);                                       /* End of allocate_pdme entry */


     /* This ends the pd_free_list module */


     end pd_free_list;
```

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

**CS-TR Scanning Project**
**Document Control Form**

Date : 11/30/95

**Report #** LCS-TR-171

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: 146 (150-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

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

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

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

Check each if included with document:

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

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description : Page Number:
IMAGE MAP (1-146) UNTITLED TITLE PAGE, 2-145, UNTITLED BLANK
(147-150) SCANCONTROL, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 11/30/95 Date Scanned: 12/5/95 Date Returned: 12/7/95

Scanning Agent Signature: Michael W. Cook

# Scanning Agent Identification Target

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

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