Technical Report TR-144
NGR-21-002-206

January 1971

A Virtual Memory System Design

by

Oliver Ray Pardo

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER
COLLEGE PARK, MARYLAND

Technical Report TR-144 January 1971
NGR-21-002-206

A Virtual Memory System Design

by

Oliver Ray Pardo

This research was supported in part by Grant NGR 21-002-206 from the National Aeronautics and Space Administration to the Computer Science Center of the University of Maryland.

Abstract

This paper describes a virtual memory management system based upon the working set model that is suitable for microprogramming. As introduction, several current systems employing virtual memory are briefly described, and the working set strategy, the task scheduling/memory management algorithm, and the concept of a paging drum are introduced. The paper then presents, in detail, the hardware and software components of a virtual memory management system. The hardware components which are described in CDL include the main memory, the translation memory, the page table memory, and the paging drum channel. The software components are described in ALGOL and interfaced with the hardware by defining registers as global variables and referring to hardware sequences through procedure calls. The software description includes complete description of the addressing mechanism, memory management, and task scheduling. Memory management is enhanced by the use of a page-table memory that posts the current status of each physical page.

TABLE OF CONTENTS

1. INTRODUCTION

Storage allocation has been a major problem since the development of the first computers. Efficient use of processors requires fast access to the data being processed. Program logic for the more complex programs together with the program data require large memory storage. However, as memories get larger (and more expensive) they become slower. Early solutions to the problem were for the programmer to perform elaborate overlays of both program and data in order to fit into a relatively small, fast memory.

In the early 1960's, a more satisfactory solution to the problem of storage allocation was proposed. This solution, now known as virtual memory, gives the programmer the illusion of a very large memory although the computer may actually have a relatively small memory. This is achieved by drawing the distinction between the address (or name) of a quantity (e.g., register) and the actual location. The programmer programs as if his memory space (virtual name space) is extremely large. The computer system provides a mechanism for executing this program.

One mechanism, paging, is based upon the idea that although a program may be very large in total size, only a portion of it is being processed at any one time. Paging divides the program into equal sized blocks, called pages, and provides a translation mechanism for associating a virtual memory page with an equal sized block of memory (Figure 1-1). During instruction execution, each virtual address is translated to a physical address before the actual memory access is made. This translation consists of accessing a special table with the virtual page address and obtaining the actual page address in memory from the addressed table location.

It is the responsibility of the computer system to move copies of the

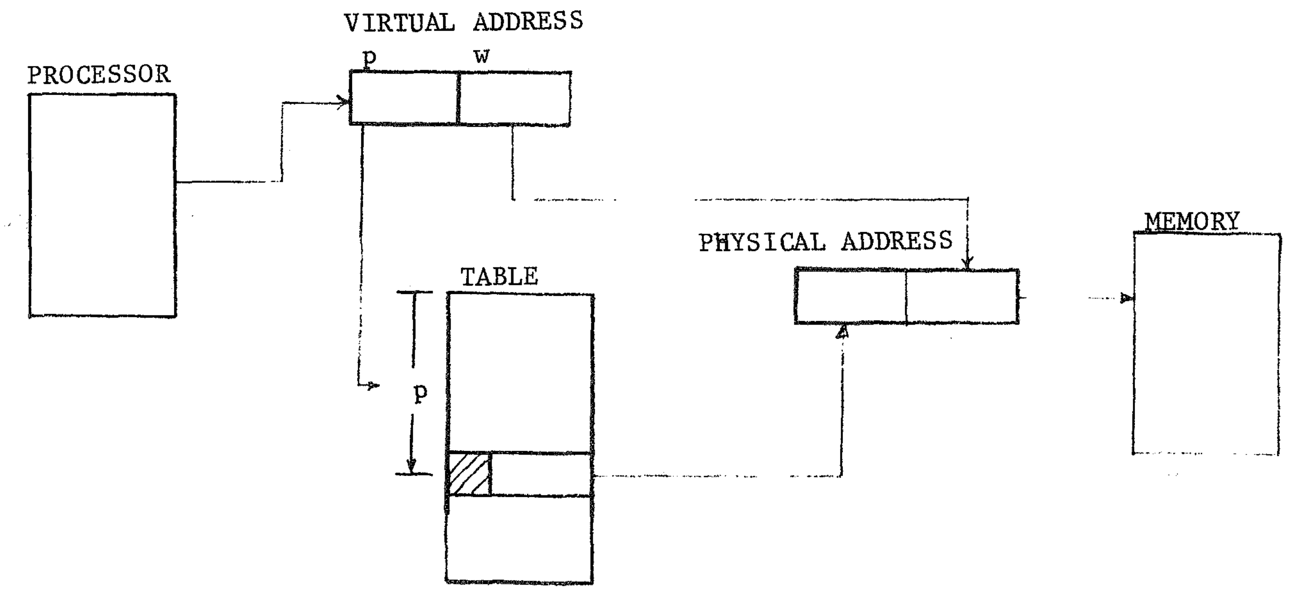


Figure 1-1 A paging scheme

virtual memory to the main memory as it is needed (placement), to remove those pages that are no longer needed out of memory (replacement), and to perform the address translation. This function of the computer is called memory management.

Besides the additional memory space afforded the programmer, this concept of a virtual memory is ideal for a multi-programming computer system, where several users (called tasks) share the systems facilities (e.g., processors, memory, mass storage, etc.) concurrently. A program that is paged must halt processing each time it references a virtual page that does not currently reside in main memory and wait for it to be entered. In a multi-programmed environment, another task can be given control of the processor in the interim.

This paper presents a virtual memory management system for microprogramming. The virtual memory system is based upon the working set model reported by Denning [12]. The system is defined to exist in a multi-programmed environment , the scheduling implementation of which is described in detail. Section 2 introduces the topic of dynamic storage allocation, the working set model, the memory management system, and the multi-programmed environment. Section 3 presents the hardware subsystems upon which the implementation is based. These are described in the Computer Design Language [5,6]. Section 4 presents the scheduling system, described in ALGOL.

This paper is regarded as the first step in a microprogrammed implementation of the virtual memory management system described herein (under the methodology described in [7]).

2. A VIRTUAL MEMORY PAGING MACHINE

The virtual memory paging machine presented here is primarily based upon a number of articles written by Peter J. Denning [12, 13, 14] in which he presents some detailed analysis into the problems of paging and segmenting systems and suggests a solution, the working set strategy. This section first introduces virtual memory systems in general and the working set model in particular, and then presents an implementation of the working set strategy.

2.1 Dynamic Storage Allocation Systems

The system described in this paper was developed after studying several of the existent paging and/or segmenting systems. A brief survey of a number of these systems appears in Randell and Kuehner's paper [27]. Several of the salient features are discussed below. The systems discussed are the MULTICS GE 645 System, the IBM 360/67 system, and the RCA Spectra 70 Systems (70/46 and 70/61).

The MULTICS System [11] employs a segmented addressing scheme with dynamic allocation implemented by paging. Addressing consists of the descriptor base register (DBR), the generalized address, and two types of segments in memory -- the descriptor segment and the information segment. The generalized address consists of a segment number/word number pair. As shown in Figure 2-1, the DBR points to the current descriptor segment, the segment number is used to access the address of the information segment, and the word number is used to locate the desired word. All segments are divided into pages and when any segment is addressed, the hardware translation to the actual page is performed transparent to the user. Paging is implemented by means of page tables in main memory which provide for trapping in case a page is not present in main

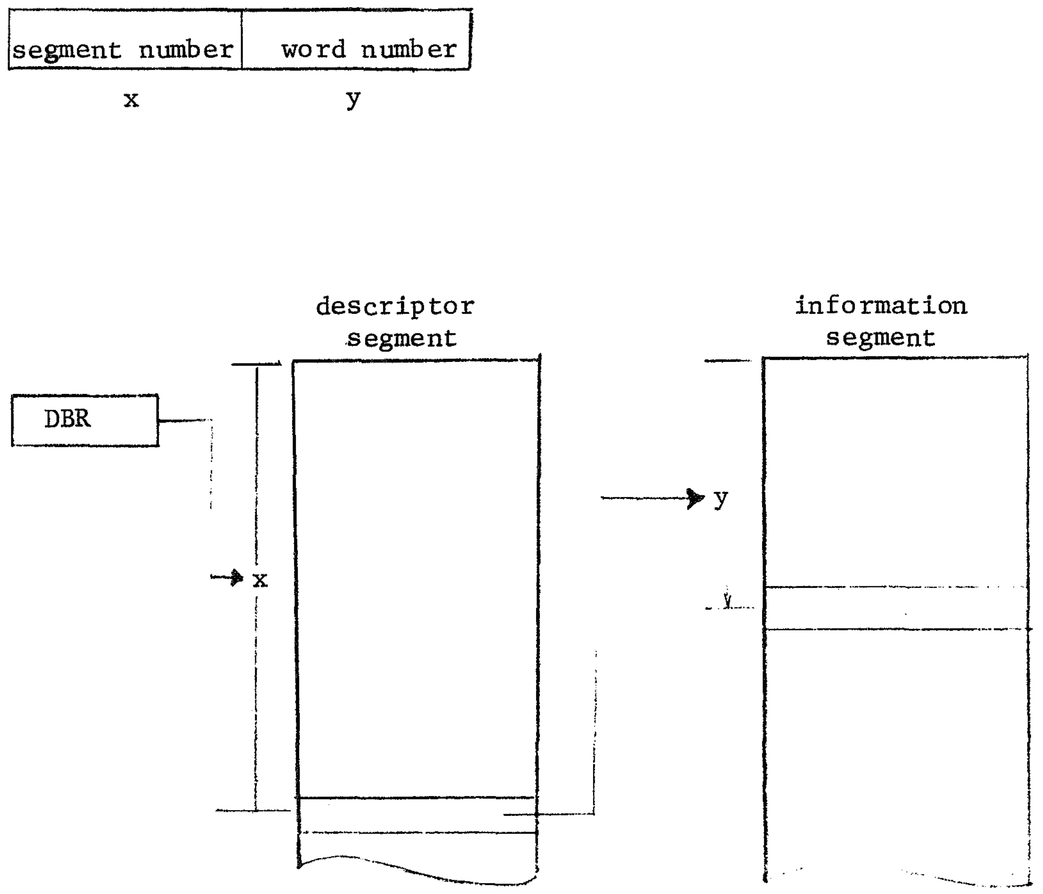


Figure 2-1 MULTICS addressing (paging not shown)

memory. In order that references to page tables or descriptor segments may be by-passed, a small associative memory is used. The process of addressing consists of accessing the descriptor segment and the associative memory in parallel for the address of the information segment word. If all segments are indeed paged, and the associative memory does not contain the page address, then the DBR points to a page table for the descriptor segment and each location in the descriptor segment points to a page table for a particular information segment. However, a special register contains the address of the current segment and only on inter-segment transfers must four main memory accesses be made. Furthermore, in a large percentage of the cases, the address will exist in the associative memory, requiring only an associative memory access and a main memory access.

The IBM System 360-67 [18,20] employs a linearly segmented name space, and the addressing mapping mechanism is as shown in Figure 2-2. Virtual memory consists of 16 segments of 256 pages each. Each page consists of 4096 8-bit bytes. Each 24-bit address consists of a segment (s), page (p), and word (w) part. To speed memory access an 8-word associative memory is used that, according to the manual [20], "contains the most recent and/or most frequently used page addresses." (If this statement means that the least frequently referenced or least recently used address is always overlaid, it can be shown to be false.) Addressing occurs sequentially, (a) the associative memory is searched in parallel for the desired page, (b) if not found the segment table is accessed for the desired page table, and (c) the absolute address is located in the page table. It should be noted that this differs from the MULTICS system, in which steps (a) and (b) occur in parallel. However, as the associative search is accomplished in .15 micro-seconds, compared to ~1 micro-second for main memory access, this may be considered to be of little importance.

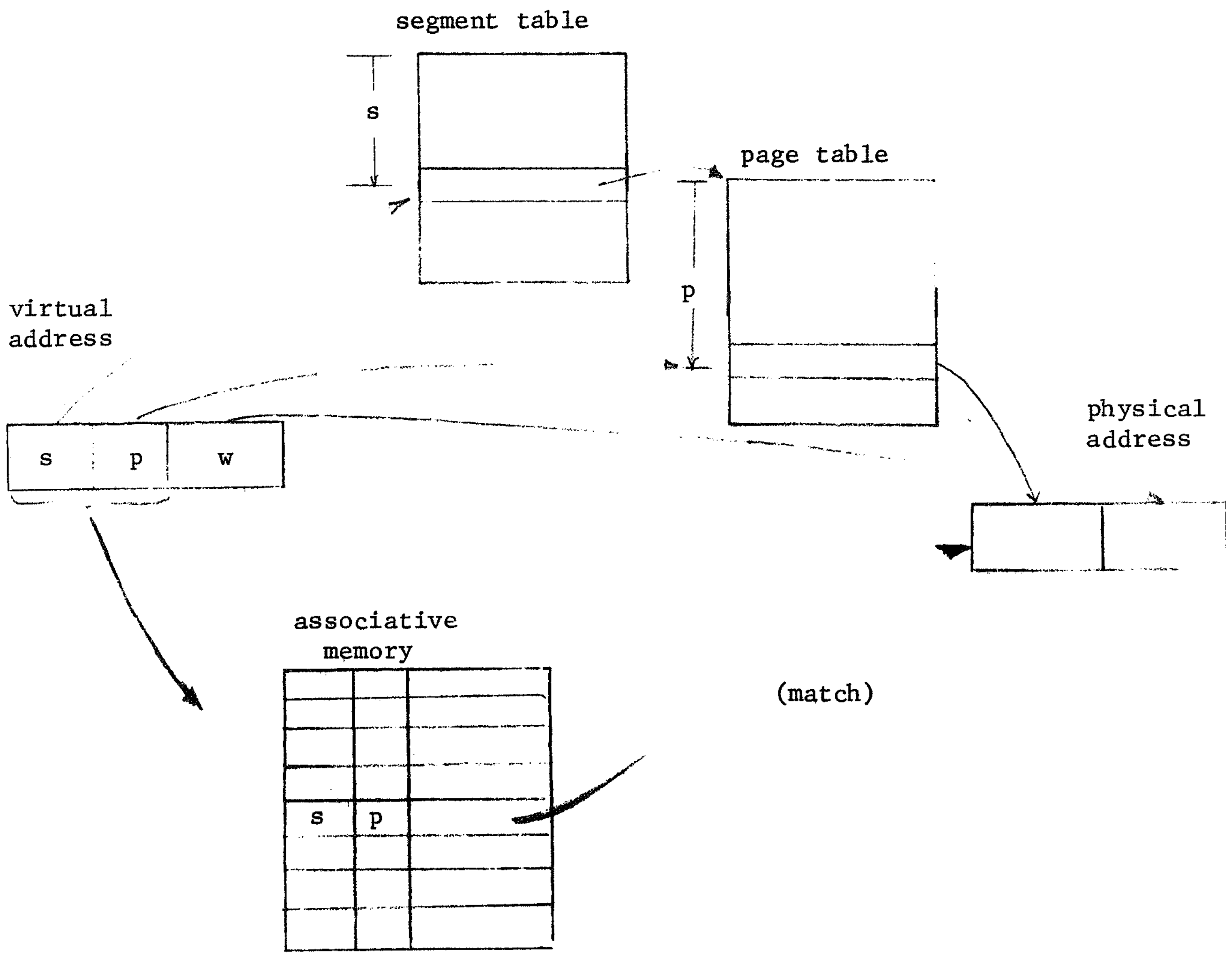


Figure 2-2 IBM 360/67 addressing

Again, if the associative memory contains the address, then only one main memory access must be made.

The RCA Spectra 70 System has two systems that employ paging, the 70/46 and 70/61. The Spectra 70/61 is an "enhanced" version of the 70/46. The addressing mechanism is essentially the same, and therefore references to both will be used [24,30,33]. Although it employs a 24-bit address (Figure 2-3), only 2 million bytes (8 segments x 64 pages/segment x 4096 bytes/page) of virtaul memory are available. Each user is allowed 256 pages of virtaul memory, pages 0 through 255. The system occupies pages 256 through 511 of all users' virtual memory. The addressing is performed through a 512 entry <u>translation memory</u> which has an 85 nanosecond access time. As a user task takes control of the processor, his copy of the lower 256 translation memory locations is loaded from main memory. At that point, all addressing occurs at a speed of $\sim$1 microsecond (85 nanoseconds for the translation memory plus 765 nanoseconds for the main memory [30]). When a task is removed from processing, those translation memory locations that have been altered are stored to main memory. The read-only memory contains the special subroutines that perform the functions of (a) load translation memory (b) scan translation memory and store, and (c) store translation memory. It should be noted that (a) only the largest tasks would require all 256 translation memory locations be loaded, and (b) the upper 256 translation memory locations are loaded only at system load time. The system portion of the virtual memory contains the copies of the translation memory, and all shared (re-entrant) code in addition to usual system functions. Therefore, while sacrificing some time to the loading and unloading of translation memory, all addressing involves only one main memory access.

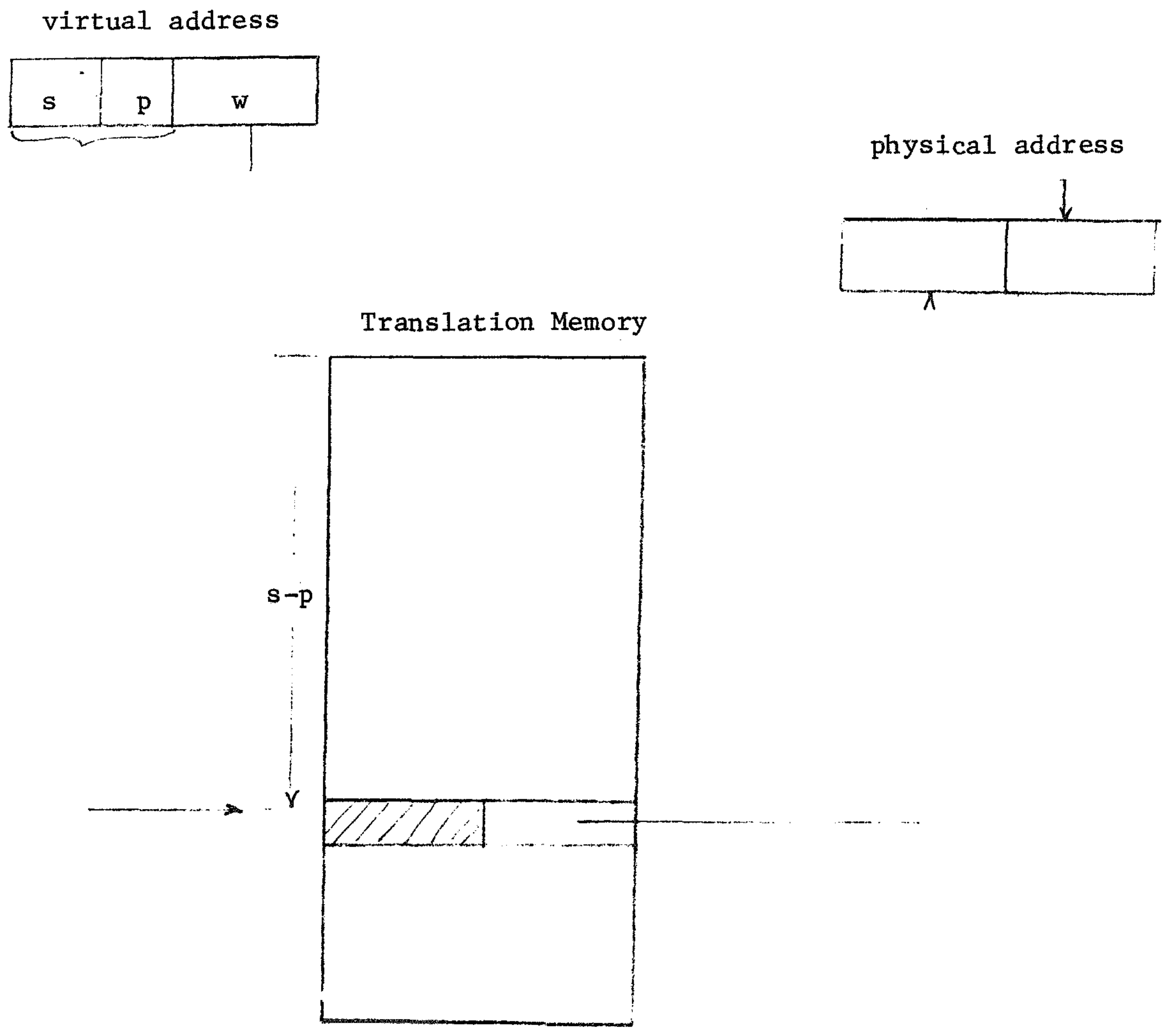


Figure 2-3 RCA Spectra 70/46 (70/61) addressing

Each translation memory location contains bits that indicate whether the page is in memory, whether it has been referenced, whether it has been changed (written into) since last being loaded, whether it is read-only, or whether it is locked-in core. The addressing operation proceeds in several steps:

(a) using the s-p portion of the logical address, access the translation memory;

(b) check to see if the page is in core;

(c) if so, then set the reference bit and, if this is a store operation, check to see if the write protect bit is off, and set the bit indicating the page has been altered; if not, issue a page interrupt, request the page be brought into main memory, unload the translation memory, and begin processing the next task;

(d) if part (c) is successful, form the actual address by concatenating the physical page address with the w-portion of the logical address.

2.2 Placement and Replacement Strategies - The Working Set

Paging systems can be characterized by the methods with which they make decisions to input or output pages from main memory. Demand paging refers to the method that only loads a page when it is referenced. Look-ahead paging attempts to predict which pages will be referenced in the near future and to load them before they are referenced. Although the latter method seems attractive, simple algorithms that perform well for a general set of processes have yet to be found [27]. The possibility of loading pages that may never be referenced contradicts one of the central premises of the paging system, to load only the set of pages that are to be used. For this reason, and that of simplicity, the system described herein will employ demand paging for page entry.

Experience shows that most of the problems in paging systems lie in the replacement strategy. If there are k tasks active, with $n_1, n_2, \ldots, n_k$ pages in memory, and task j makes a reference to a page not in core, one of two actions occur. If $\sum_{i=1}^{k} n_i < N$, where $N \equiv$ total pages in memory, then space exists and the page is loaded. However, if $\sum_{i=1}^{k} n_i = N$, then a decision must be made as to which page to remove. The obvious choice is to remove the page that is least likely to be used next. The strategies for determining the identity of this page are many.

These strategies can be divided into those that would consider all N pages likely candidates for removal (the <u>global strategies</u>) and those that consider only the n_j pages of task j (the <u>local</u> strategies). When demand paging is used, the task requesting a page will be suspended from processing, until the request is fulfilled, requiring that other tasks be processed in the interim. If a global strategy is used, it is possible that the page removed may be requested by a subsequent task. The extension of this problem is called <u>thrashing</u> [13], a condition in which the system is reduced to swapping pages to and from main memory, and useful computation is reduced to zero. Both the global and local strategies employ similar methods for choosing the page that is to be removed: least recently used, least frequently used, etc. For further details on replacement strategies and the resulting problems, the reader is referred to [2,13,23,24,27,28,29,31].

The solution chosen here, Denning's <u>working set strategy</u> [12,13], can best be understood as an attempt to provide the ideal condition: always enough room for the set of pages with which a task is working. If the pages referenced by a task are monitored, then the current <u>working set</u> of a task is defined as those pages referenced in the last τ time units, where τ is a fixed amount of time for all tasks, called the <u>working set parameter</u>. If an <u>active task</u>

is defined as a task competing for the central processor (CPU), and the cardinality of the working set is w_i, then the working set strategy is to allow only those K tasks to become active whose working sets $w_1, w_2, \ldots, w_k$ satisfy the property $W = \sum_{i=1}^{k} w_i \leq N$, where N is the total number of pages available. In order to make this strategy workable, several questions must be answered:

(a) How is the working set measured?

(b) What value is chosen for the task that has no history (i.e., has not been processed for τ time-units)?

(c) As the working set size appears to be dynamic, how are tasks made active when their working set w_j satisfies the property $W + w_j \leq N$, and, conversely, which task is removed when $W>N$?

The system described in this paper presents a workable implementation of the working set strategy and therefore provides solutions to these questions. A similar implementation was made by RCA in the Spectra 70 series and is described by Oppenheimer and Weizer [33]. As a final vote, it should be pointed out that this strategy assures that thrashing will never occur <u>between</u> tasks, as there will always be sufficient pages for each active process's working set (note that a single active task could grow to include all of available core in an active period and then thrash with itself--however, this is unlikely).

2.3 A Memory Management System

Describing a memory management system in any detail requires that more be known of the host operating system than merely the addressing mechanism. For example, to discuss demand paging requires some idea of what happens (besides the passage of time) while the page is being loaded into memory; to specify that several tasks are competing for processor time, issuing page requests and, at the same time, releasing pages, raises the question of I/O scheduling. Therefore, the memory management system presented here will be considered to be a part of a multi-programmed operating system. The task scheduling, memory management, and mass storage subsystems will be presented in detail.

The scheduling implementation for this computer is shown in Fig. 2-4 [12]. A series of tasks are ordered on a ready list, indicating they are to be processed. As main memory space becomes available, the task with the highest priority (task i) is assigned a time quantum, q_i, and entered into the running list (a circular list). Each task on the running list is processed for a burst of time β ($\beta \leq q_i$). At the end of the β seconds of process time the task is returned to the end of the running queue, unless:

(1) the task's running time exceeds the assigned quantum time ($t_i \geq q_i$), in which case the task is placed back on the ready list and t_i is set to zero;

(2) the task is blocked, waiting for completion of another task (e.g., I/O activity), in which case it is placed upon a blocked list (and then returned to the ready list when it is unblocked);

(3) the task is completed, in which case the supervisor is notified;

(4) a page fault occurs, in which case the task incurs a wait period while the page is entered into memory, and then is placed back onto the running list.

The virtual memory described here is similar to the memory system employed by the RCA Spectra 70-46 and 70/61. Each task has 2N pages of M words each in its virtual memory space. However, only the first N pages are available for the user programs as the last N pages always contain the system (e.g., processors, tables, re-entrant programs). When a task takes control of

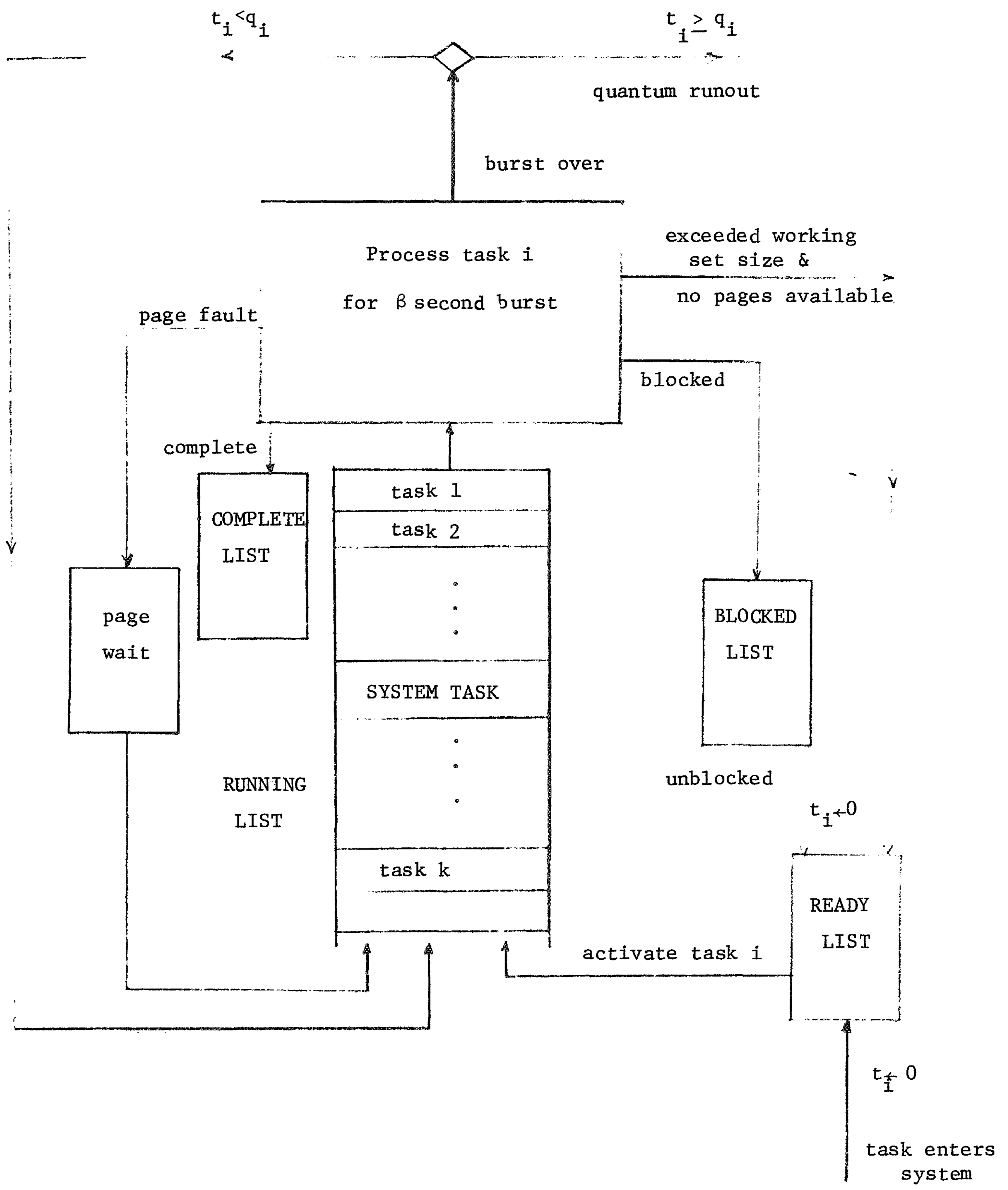


Figure 2-4   A scheduling implementation

the central processor (CPU) its virtual memory map is loaded into a special translation memory. When a task loses control of the processor, those entries of the translation memory that were altered are written to the task's buffer area. All addressing that the task performs occurs through the translation memory. If the page exists in memory, the address translation occurs, else a page fault occurs (case 4 above).

Page are swapped in and out of core according to the working set strategy. Pages are loaded to memory upon demand and attached to the requesting task. At this point they are active. Associated with each active page in memory is a count of the amount of the task's processing time that has elapsed since it was referenced. Every time a page is referenced the count is reduced to 0. If a page is not referenced over the period of the burst β, the count is increased. When this values becomes $>\tau$ then the page is disassociated from the task and becomes available. Available pages are of three types:

(a) altered, requiring that the page be swapped out before being reused,

(b) queued for swap-out, because of (a), and

(c) immediately available.

Categories (a) and (b) may be one and the same in some implementations. The reason for the distinction in this case will be explained later.

Every task's virtual memory exists in total on a mass storage device called the paging drum. Only its working set exists within the main memory. The paging drum concept is described in more detail in the section 2.4.

At this point, the working set principle will be discussed in relation to the scheduling system in order to answer the questions raised in section 2.3. A user job, consisting of several tasks, is entered into the system. As facilities become available tasks are entered into the ready list and:

(a) a task descriptor block is assigned;

(b) the task's pages are loaded to the paging drum, and the corresponding drum addresses are loaded into the task descriptor block;

(c) the task is assigned a priority and an initial working set value.

In addition, the ready list contains tasks that have already begun processing and have either run-out their time quantum or are blocked, awaiting completion of another task. In any case these tasks retain the <u>size</u> of the working set that they had when they were removed from the running list.

The running list contains a circular list of tasks that have not run-out their time quantums, are not waiting for a page, have not been blocked, and have not reached completion. These tasks are being processed for bursts of time β. In this circular list is a resident system task. The <u>system task</u> manages memory by removing those pages from the working sets that have not been referenced in τ seconds, and placing them on one of the available lists. It keeps count of those pages that are not currently a part of a working set. If the ready list contains a task of high enough priority whose working set would "fit" into main memory, the task is entered onto the running list and its working set size is deducted from the available page list. The task enters its pages on demand and a current working set size is computed. If after the first τ seconds of processing, the task has not reached its specified size, the <u>reserved pages are returned to the available list</u> (simply by adjusting the count). If in processing, a task requests a page that, exceeds its working set size, it is allowed to grow as long as pages are available; otherwise, it is removed from the running list.

Pages become inactive and available for one of two reasons: (a) they have not been referenced in τ seconds (have left the working set) or (b) the task to which they are attached becomes inactive. A task becomes inactive due to quantum time-out, task blocking, or exceeding the working set size when no pages are available.

Therefore, the system maintains dynamic working sets for active tasks. In order to reduce page swapping to a minimum, pages that are to be swapped out (because they have been altered) are queued together in a special available list (swappable). Those that do not require swapping are queued in another list. Each time a page is swapped in on demand, if the swappable queue is not empty, then a page is swapped out. As these are queues, the last entries on the queues will be the most recently referenced. Therefore, if a task references a page that has recently been removed from its working set, there is a good chance that it will still be in memory. If the available page count is non-zero, it can be re-attached immediately.

The rest of this paper details the memory management system and the scheduling algorithm. The remainder of section 2 will discuss the principles of the paging drum and parallel processing. Section 3 presents the hardware components of the system. Section 4 presents the overall logical structure of the system in the form of detailed ALGOL procedures.

2.4 The Paging Drum

Due to the fact that each task is given the impression that it is the sole user of a computer with an enormous address space and as several tasks may be competing for services, the set of pages representing each task's address space must be stored on some form of mass storage device. In addition, as the main memory is of a size such that only a relative few of each task's pages may reside in main memory at a time, this memory must be as efficient as possible.

For this task, a large rotating drum with a fixed head per track is chosen. The layout of the drum is as shown in Figure 2-5. The drum circumference is divided into equal parts called sectors, and the drum length is divided into equal numbers of tracks called channels (or fields). The division is made such that the area bounded by a sector and a field contains one page of information. Information is transferred to and from the drum in pages and therefore it is called a paging drum.

A simple implementation of a paging drum to consider it as an arbitrary storage device (i.e., take no advantage of its structure) and queue requests for it first come-first served (FCFS). However, if a queue is established for each sector (Figure 2-6) then the performance of the system is enhanced considerably. For the case of N sectors and revolution time T, the expected access time for the FCFS queue is the expected access time for one request

$$E[a] = \frac{N-1}{2N} T.$$

For the multiple-queue, Denning [14] establishes the expected access time as

$$E[a] = \frac{T}{n+1} (1 - \frac{1}{2N})^{n+1}$$

where n is the number of requests waiting in the queues (considering no penalty for switching from read to write). It should be noted that this is expected access time for the next request and not expected service time for each request. However, as far as drum utilization is concerned, the second result shows that under a heavier request load (n>>1) the multiple-queue drum performs better, while the single queue drum has a constant utilization. In addition, in a page-on-demand system, the order of request service is unimportant. The performance of the paging drum has been given extensive study, and

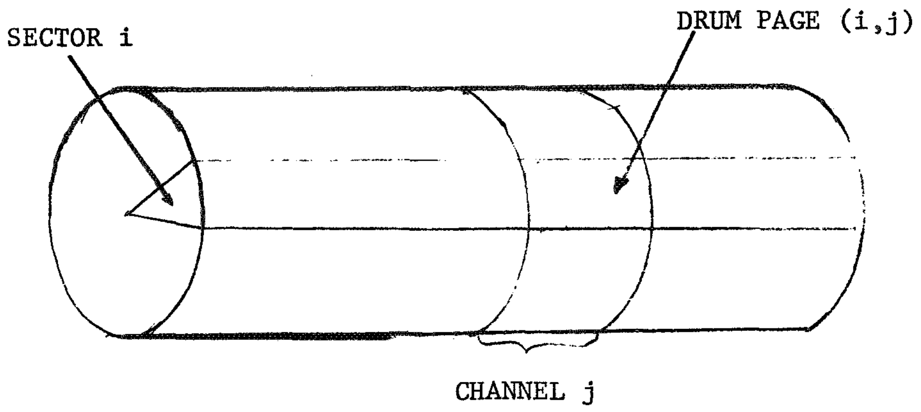


Figure 2-5   Paging drum diagram

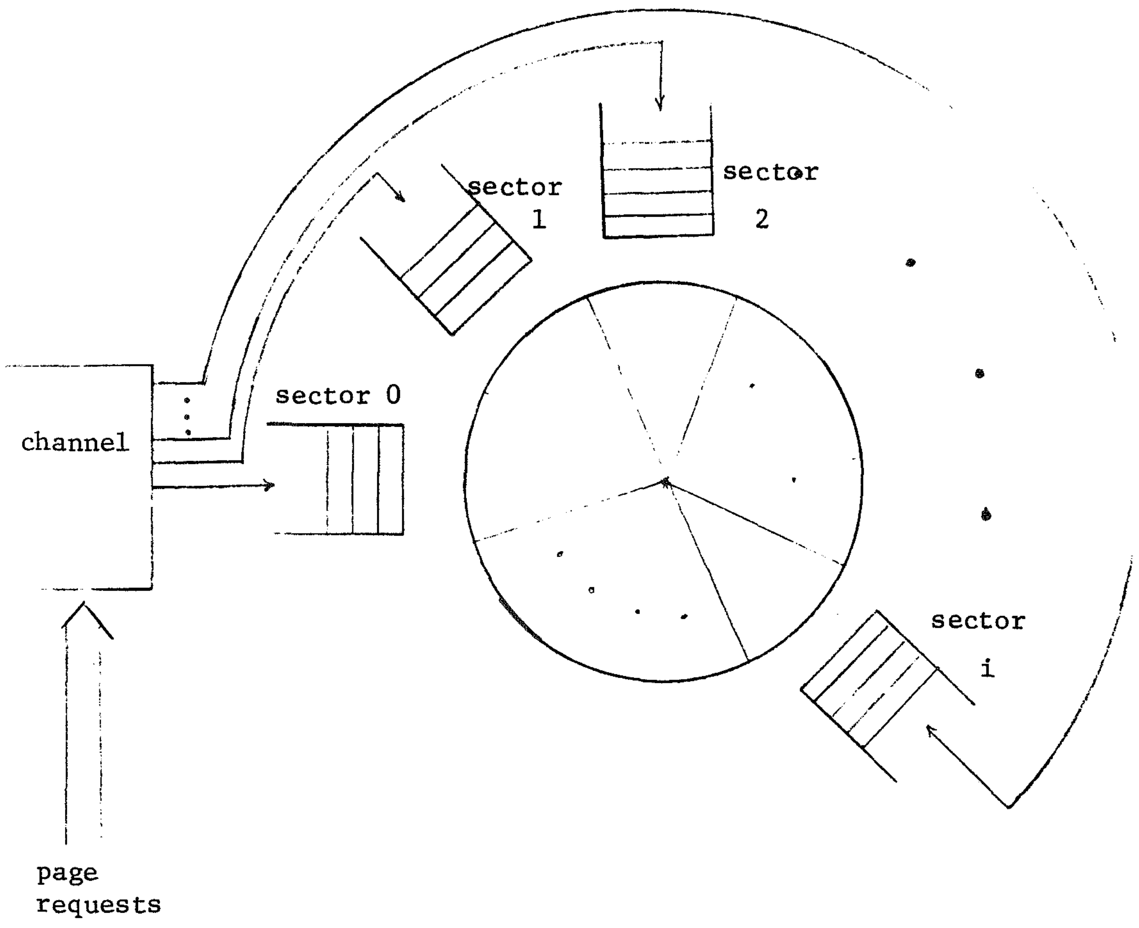


Figure 2-6 Organization of the paging drum sector queues

the reader is referred to a paper by Weingarten [32], Denning [14], Coffman [8], and Abate and Dubner [1].

The result obtained above depends upon the qualification that a switch from drum read to drum write requires no significant delay. As noted by Coffman [8], "after reading (writing) a given sector it is not possible to instantaneously switch the status of the heads so as to commence writing (reading) on the next sector." He goes on to suggest several ways of effecting a "read/write" (i.e., no delay) drum, including a gap between sectors, and two sets of heads (one read, one write). A particularly interesting method is called the precessing drum. Although there are N sectors on the drum, the drum reads or writes to every Mth page (where the greatest common divisor for M and N is 1). Therefore the drum in effect becomes M times slower but contains no wasted space. The point of this method is that the "gap" of skipped pages allows the heads time to switch states.

The paging drum defined in section 3.4 is considered to be a "read/write" drum. This will be satisfactory for the purposes of this paper, as the logical description is the same.

2.5 Concurrent Processes

Before continuing on to the implementation, one last topic deserves mention. The system presented in the next two sections specifies three processors which operate asynchronously of one another: the CPU, the paging drum channel, and the I/O channel. However, at specific times in the processing, each must notify the other of events that require reciprocal action, and, at other times, they must use facilities (e.g., memory tables) and possibly instruction sequences that are accessed by the other processors.

It is imperative in these operations that information not be lost, that deadlocks do not occur, and that all processors are served in a reasonable time. These problems have been studied by Dijkstra [15,16] and Wirth [34]. The reader is referred to these papers for further discussion of this topic.

3. HARDWARE DESCRIPTION

This section presents, in detail, the four major system components which provide the basis for the memory management system:

(a) main memory. A subsystem that allows three processors to share the same core memory.

(b) translation memory. The subsystem that performs the address translation from the logical (virtual) address to the physical address, resulting in either a fetch/store of main memory, or a page fault.

(c) page table. The subsystem that the memory management is based upon. It consists of a fast memory that contains one entry per physical page, containing protection information, information as to the page status, and drum location.

(d) paging drum channel. The subsystem that swaps the pages to and from memory.

Figure 3-1 shows the overall configuration of the subsystems in relation to the CPU. It includes those registers through which the subsystems communicate, which are explained further below.

There are three processors in Fig. 3-1: the central processing unit (CPU), the paging drum channel processor, and the I/O channel processor. This paper will confine itself to presenting the addressing and interrupt processing of the CPU and the operation of the paging drum channel. Although the operation of the I/O channel is not detailed, its points of contact with the rest of the system (i.e., main memory access, and CPU interrupts) are included. The main memory is shared by the three processors; therefore, each has its own read/write register (RWi), storage buffer (SBRi), and address register (MADRi).

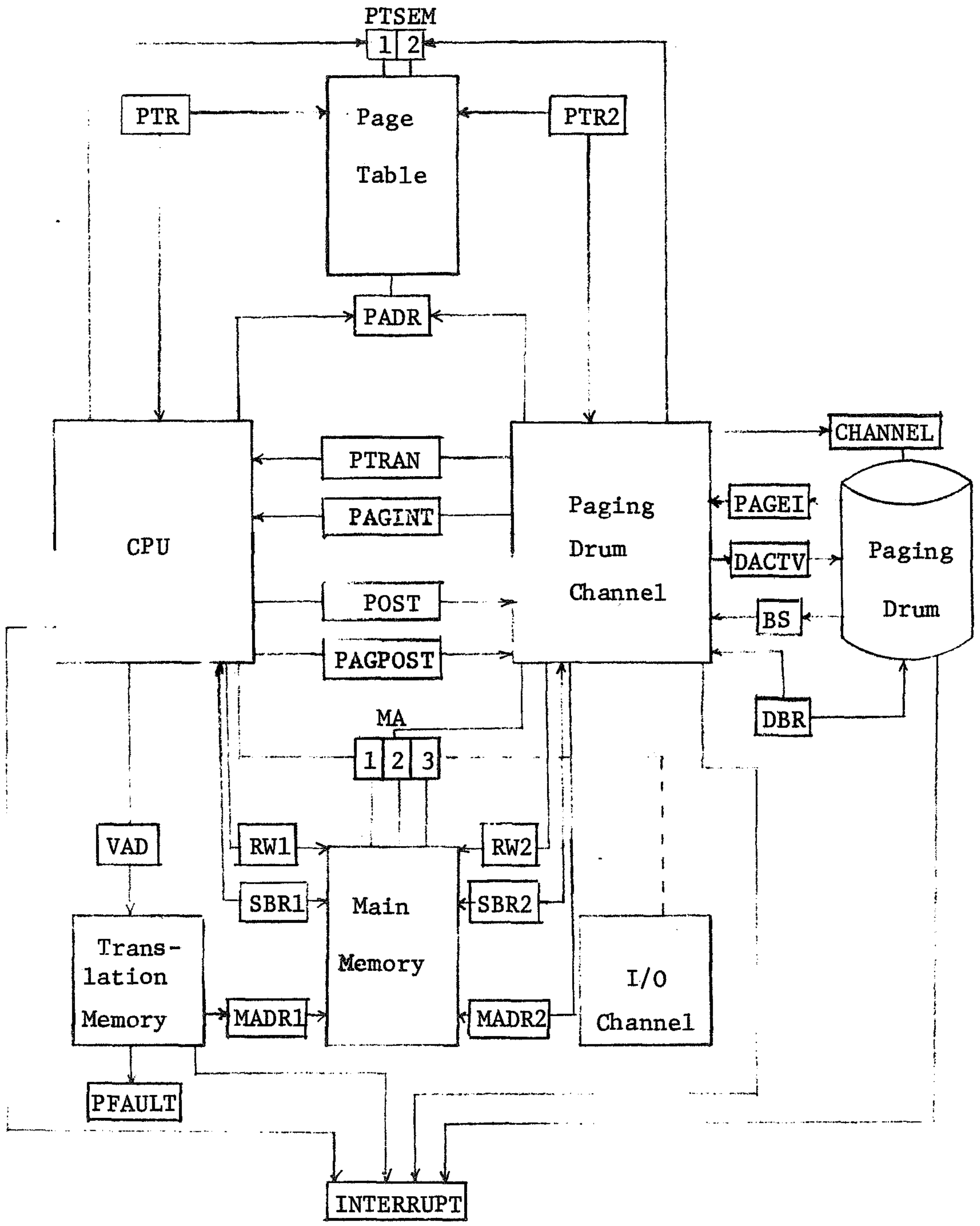


Figure 3-1 Basic Configuration

In addition, each processor is assigned an access bit (MA(1)) with which to signal the main memory of desired access. The main memory uses this register (MA) to resolve conflicts of access between processors, to assure each processor of service within three memory cycles, and to post the completion of the desired memory operation to the appropriate processor.

The CPU addresses the main memory through the translation memory in order to effect virtual addressing: given a virtual address (VAD), either a physical address (MADRi) or page fault (PFAULT=1) results.

In order that the CPU can keep track of physical page assignments (e.g., available pages), and in order that the paging drum channel can keep a list of pages to be swapped, the special "page table" memory is introduced. Consisting of one memory location for each physical page, it contains the lists of each active task's working set, the list of available pages, and the list of pages to be swapped. The "page table" memory is shared by the CPU and paging drum channel. Simultaneous access is prevented by supplying an access (semaphor) register PTSEM.

The CPU and the paging drum channel communicate to each other through the page table memory and five registers:

(a) INTERRUPT, one bit of which signals completion of a page transfer to the CPU,

(b) PTRAN, which signals the direction of transfer, or error in transfer,

(c) PAGINT, which specifies the page transferred,

(d) POST, which signals the paging drum channel that a page is to be transferred,

(e) PAGPOST, which specifies the page to be transferred.

The paging drum channel is a dedicated processor, linked only to the paging drum. It addresses the drum through the address register CHANNEL and writes to/reads from the drum word-serial through the buffer register DBR. The paging drum signals the passing of a page boundary through register PAGEI.

The four subsystems will be presented simply without dwelling on physical problems. For example, a single-bank shared memory as described in section 3.1 may be severely handicapped under normal CPU, page channel, and I/O activity. This problem is easily met through interleaving but the additional description necessary would merely complicate the overall system description. The paging drum description presents a similar problem. The drum has a capacity of 4096 pages of 1024 words each (at 48 bits per word). Although the drum circumference (16 K bits) is reasonable, its apparent length (256 x 48 bits) may be excessive. However, although the drum is logically described as a unit it could physically be several drums (with the additional complications of synchronization). Therefore, the reader is cautioned to recognize that implementation of such a system requires much attention to the timing and physical limitations of each component.

Complete description of the system will be deferred to section 4. In that section a task scheduling system will be presented in order to more completely describe the addressing and memory management algorithms. For clarity, the task scheduling system is presented as a series of ALGOL procedures which operate on the subsystems presented in this section through a series of primitive procedures. For this reason, with each subsystem, procedure calls and the corresponding CDL description will be presented. In the same vein, subregisters will be accessed through function procedures of the same name as the subregister, with the register name as argument.

3.1 Main Memory Subsystem

The main memory subsystem consists of a core memory, address registers, storage buffer registers, read/write control registers, and a control section that performs timing functions and handles conflicts between competing requesters. A requester may be any of the system processors (e.g., the CPU or an I/O channel). For simplicity, only three processors will be defined in this implementation as competing for main memory: (a) the CPU, (b) the paging drum channel, and (c) a general I/O channel. In addition, the memory will consists of one bank (i.e., no interleaving).

Each competing processor i will have a memory register MADRi, storage buffer register SBRi, and read/write register RWi and will be unaware that it is competing for the main memory. To access main memory, the processor loads the address register and, if writing, the storage buffer register, sets RWi to 1 for a read or to 0 for a write, and sets a bit of the memory access register, MA(i), to 1 to indicate a memory access. The memory subsystem consists of the memory access register MA, the memory address register MAR, the memory buffer register MBR, a memory read terminal READ, a memory write terminal WRITE, and 65K of 48-bit core memory, MEM(MAR). The CDL description of the memory appears below.

```
Comment, main memory subsystem
Memory, MEM(MAR)=MEM(0-65535,1-48)     $main memory
Register,   MAR(1-16),        $main memory address register
            MBR(1-48),        $main memory buffer register
            MA(1-3),          $memory access register
            MADR1(1-16),      $CPU address register
            MADR2(1-16),      $paging drum channel address register
            MADR3(1-16),      $I/O channel address register
            SBR1(1-48),       $CPU storage register
            SBR2(1-48),       $paging channel storage register
            SBR3(1-48),       $I/O channel storage register
```

```
                RW1,         $CPU read/write register
                RW2,         $paging channel read/write register
                RW3,         $I/O channel read/write register
Terminal,       READ,        $read terminal
                WRITE,       $write terminal
```

The sequence chart for the main memory subsystem is shown in Figure 3-2. The actual read or write is performed by loading MAR from and loading/storing MBR from/to the appropriate processor's registers depending upon the value of RWi and MA(i), and setting either the READ or WRITE terminal. If READ is set to 1, then the transfer

$$MBR \leftarrow MEM(MAR)$$

occurs. If WRITE is set to 1, then the transfer

$$MEM(MAR) \leftarrow MBR$$

occurs. It should be noted that the logic in the sequence chart resolves all conflicts and that any one processor must be serviced within three main memory cycles of the request.

Table 3-1 presents the two procedures for accessing main memory: (a) loadmm, read from main memory, and (b) storemm, write to main memory. Both of these procedures use the storage buffer register SBR as the source or destination of the information transferred. The actual memory access is accomplished by setting bit MA(1) to 1, and then waiting for completion (MA(1)=0) before returning control.

3.2 Translation Memory Subsystem

Virtual memory, in this implementation, consists of up to 1024 pages of 1024 48-bit words each. Each address field consists of 24 bits as shown in Figure 3-3. The D field indicates that this is either a virtual (D=0) or physical (D=1) address. The I field indicates whether this is an indirect

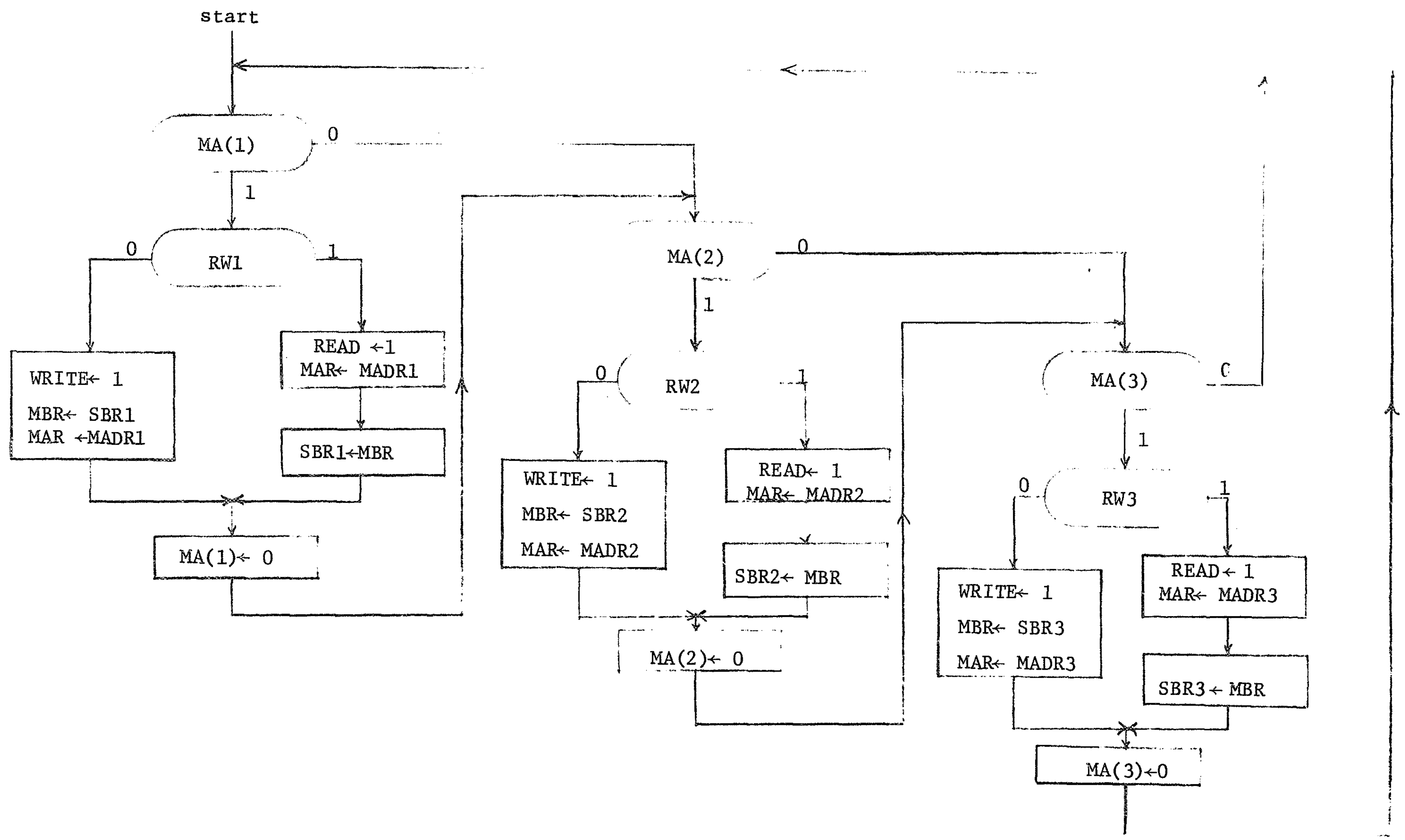


Figure 3-2 Sequence chart for main memory subsystem

procedure call	CDL description	explanation
loadmm (i,j,SBR)	MADR1(1-6)←i;	load SBR from
	MADR1(7-16)←j;	main memory location
	RW1←1;	page i, word j
	MA(1)←1;	
	IF(MA(1)=0) THEN(return)	
storemm (i,j,SBR)	MADR1(1-6)←i;	store SBR to
	MADR1(7-16)←j;	main memory
	RW1←0;	location page i,
	MA(1)←1;	word j
	IF(MA(1)=0) THEN(return)	

TABLE 3-1 Main memory procedures.

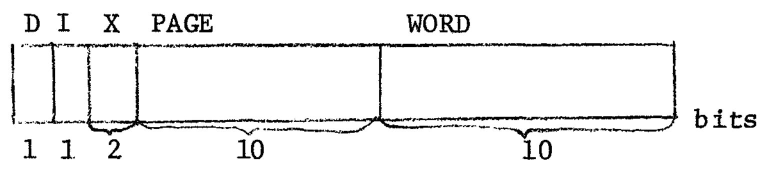
address (I=1) or not (I=0). The X field indicates whether indexing is to be applied to this address ($1 \leq X \leq 3$) or not indexing (X=0), and which of 3 index registers to use. The 20-bit ADR field either contains a 20-bit virtual address (D=0) or a 16-bit physical address (D=1).

The correspondence between the virtual name space and physical name space is achieved through a 1024 word translation memory. As the virtual memory is split between the user task (pages 0 through 511) and the system (pages 512 through 1023), each task has its own copy of the portion of translation memory it is using (between 1 and 512 pages). When a task gains control of the CPU, its copy of translation memory is loaded. When the task relinquishes control of the CPU, those locations in the translation memory that were altered are copied.

When addressing the virtual name space, the high-order 10 bits are used to address the translation memory. Each word in the translation memory, as shown in Figure 3-3, consists of 16 bits. The ACT field indicates whether or not the page is currently active (i.e., a member of the task's working set); the REFD field indicates whether or not the page has been referenced; the CHGD field indicates whether or not the page has been written into; the WP field indicates whether or not the page is write protected; the WKEY field is the protection key for the page; and the BLK field contains the 6-bit physical page address for the virtual page if it is in main memory.

The CDL description of the translation memory configuration appears below. In addition to those registers mentioned above, the configuration consists of the translation memory address register TADR, the read error register RFLAG, the write error register WFLAG, the page fault register PFAULT, and the main memory register MADR1, SBR1, RW1, and MA. The main memory registers were explained previously. The register RFLAG is set to 1 if an attempt is made to read from a page whose key does not match the task's key. The register WFLAG

Virtual memory address format



Translation memory address format

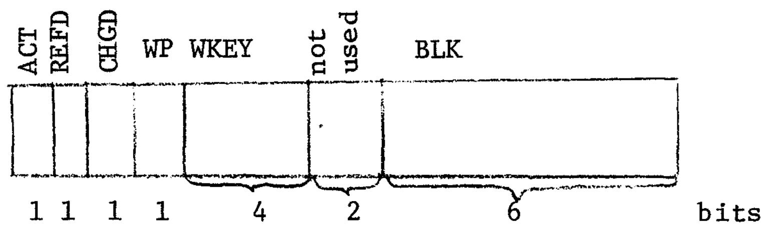


Figure 3-3

is set to 1 if an attempt is made to write to a page that is either write protected (TMR(WP)=1) or whose key does not match the key of the current task. The register PFAULT is set to 1 if a virtual page is addressed whose physical page is not currently a part of the task's working set (TMR(ACT)=1).

```
Comment, translation memory subsystem
Memory, TMEM(TADR)=TMEM(0-1023,1-16)    $translation memory
Register,   TADR(1-10),      $translation memory address register
            TMR(1-16),       $translation memory buffer register
            VAD(1-24),       $virtual address register
            MADR1(1-16),     $main memory address register (for CPU)
            RW1,             $read/write register (for CPU)
            RFLAG,           $read error flag
            WFLAG,           $read error flag
            PFAULT,          $page fault register
            MA(1-3),         $main memory access register
            SBR1(1-48),      $storage buffer register (for CPU)
Subregister,TMR(ACT,REFD,CHGD,WP,WKEY,NOTUSED,BLK)=TMR(1,2,3,4,5-8,9-10,11-16)
            VAD(D,I,X,PAGE,WORD)=VAD(1,2,3-4,5-14,15-24),
            MADR1(BLOCK,WRD)=MADR1(1-6,7-16)
```

Table 3-2(a) presents the two primitive procedures for accessing translation memory: (a) loadtmr, reading from the translation memory, and (b) storetmr, writing to the translation memory. Table 3-2(b) lists thirteen function procedures that are used to reference subregisters.

3.3 Page Table Subsystem

Memory management is enhanced by the addition of a special fast memory called the page table. The page table has one entry for each physical page in memory and is not to be confused with the translation memory, which has one

procedure call	CDL description	explanation

loadtmr(i,TMR)	TADR←i,
	TMR←TMEM(TADR)
		load the buffer register TMR
		from translation memory addr. i

storetmr(i,TMR)	TADR←i, TMEM(TADR)←TMR	store the buffer register TMR into translation memory add. i

(a) translation memory procedure calls

function	CDL description	explanation
act(TMR)	TMR(ACT)	active page indicator
refd(TMR)	TMR(REFD)	page reference indicator
chgd(TMR)	TMR(CHGD)	page change indicator
wp(TMR)	TMR(WP)	write protect indicator
wkey(TMR)	TMR(WKEY)	page key
blk(TMR)	TMR(BLK)	memory address of page
d(VAD)	VAD(D)	physical/virtual address indicator
i(VAD)	VAD(I)	indirect addressing indicator
x(VAD)	VAD(X)	indexing indicator
page(VAD)	VAD(PAGE)	virtual page address part of VAD
word(VAD)	VAD(WORD)	word address part of VAD
block(MADR1)	MADR1(BLOCK)	physical page addr. part of MADR1
wrd(MADR1)	MADR1(WRD)	word address part of MADR1

(b) translation memory field functions

Table 3-2

entry for every page in virtual memory. The page table consists of 64 66-bit words, called page descriptors, that represent the current status of the page, the task it is or was attached to, its protection bits, utilization information, the corresponding virtual address, the drum address, and list linkage information.

The CDL description for the page table appears below. The page table consists of the fast memory PAGETABLE, its address register PADR, its buffer register PTR, counters CAVPA, CAVPV, and AVPC, and list registers PTLIST, LSP, LAVP.

```
Comment, page table configuration
Memory, PAGETABLE(PADR)=PAGETABLE(0-63,1-66)
Registers,   PADR(1-6),      $page table address register
             PTR(1-66),      $page table buffer register
             CAVPA(1-6),     $count of immediately available pages
             CAVPV(1-6),     $count of total available pages
             AVPC(1-7),      $reserved page count (for working sets)
             PTLIST(1-12),   $general page table list register
             LSP(1-12),      $list of swappable pages (pointer register)
             LAVP(1-12),     $list of available pages (pointer register)
             GPTL(1-12),     $general page table list register
             PTSEM(1-2),     $page table semaphore
Subregisters, PTR(USE,LB,LF,WKEY,WP,CHGE,RES,UTIL,TID,VP,DP,ROW)=
              PTR(1-2,3-8,9-14,15-18,19,20,21,22-27,28-43,44-53,54-65,66),
              AVPC(SIGN,MAGNITUDE)=AVPC(1,2-7),
              PTLIST(FP,LP)=PTLIST(1-6,7-12),
              LSP(FP,LP)=PTLIST(1-6,7-12),
              LAVP(FP,LP)=LAVP(1-6,7-12),
              GPTL(FP,LP)=LAVP(1-6,7-12),
```

Table 3-3 presents the format of the page table buffer register, PTR. Field PUSE defines the current usage:

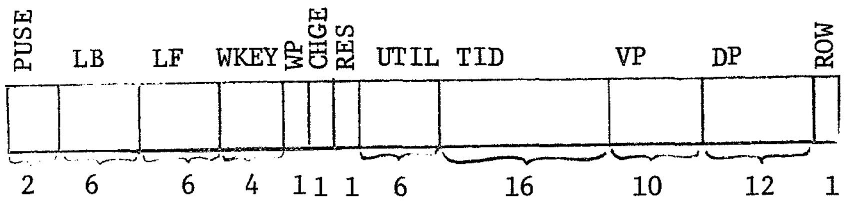
(a) *active* (PUSE=0). The page is attached to a task's (TID) working set and linked to it's list of active pages through LB and LF. The page's key WKEY, write protect bit WP, virtual address VP, and drum address DP are loaded with the corresponding values. Field CHGE reflects whether the page has been altered; field RES indicates whether it is resident; and field UTIL is a counter reflecting the time since the page was last referenced (every time the entry for a *referenced* page is removed from translation memory its UTIL field is loaded with zero).

(b) *queued for swap-out* (PUSE=1). The page is attached to the list of swappable pages (pointed to by the LB and LF fields of register LSP). Field ROW is set to 1. All other fields remain as they were just before being placed on this list.

(c) *unavailable* (PUSE=2). The page is undergoing swapping (either in or out) and may not be accessed by the CPU (except by procedure *queue request*, to be described in section 4).

(d) *available* (PUSE=3). The page is attached to the list of available pages (pointed to by the LB and LF fields of register LAVP). Field ROW is set to zero; field CHGE is set to zero. All other fields remain as they were before being placed in this list.

The four list registers LSP, LAVP, GPTL, and PTLIST point to the first and last entries in a list through the FP and LP subregisters. LSP always points to the list of swappable pages; LAVP always points to the list of available pages; register GPTL and PTLIST are used as general pointer registers to point to any of the other several lists that may come into question (e.g.,



PUSE=0	,	page is active (attached to an active task)
=1	,	page queued for swap-out
=2	,	page is unavailable (I/O in process)
=3	,	page is available

LB and LF, forward and backward links for a chain of pages

WKEY	,	read/write protection key
WP	,	write protect (FP=1)
CHGE	,	bit indicating page has been altered (CHGE=1)
RES	,	bit indicating the page is resident (RES=1)
UTIL	,	utilization counter. UTIL<τ if it belongs in a working set
TID	,	pointer to the task identifier for the task: (a) the page is attached to(PUSE=0) (b) the page was attached to(PUSE=1,3) (c) the page will be attached to (PUSE=2)
VP	,	virtual page address for this page in task TID
DP	,	drum address of this page
ROW,	,	read/write indicator

Table 3-3 Page table buffer register

there is one list for each active task). Three counters monitor the available pages:

(a) CAVPA is a count of the numbers of pages contains on the list of available pages;

(b) CAVPV is a count of the number of pages available to the active tasks, either on the list of available pages, the list of swappable pages, or being swapped out;

(c) AVPC is a count of the number of pages available that are not already assigned to some active task's working set.

Three counts are necessary in order to quickly determine at any one time how many pages are available and how many are available immediately.

The page table is shared by the CPU and the paging drum channel. As these are asynchronous processors, some method of control must be included in all procedures that access the page table to insure that the two processors do not access the page table simultaneously. In addition, as both processors may alter the list structures contained within the page table any list operation initiated by one processor must be completed before control can be given to the other processor. To accomplish this, the page table access register, PTSEM is defined, which allows the two processors mutually exclusive access to the procedures (see Figure 3-4, 3-5, and 3-6).

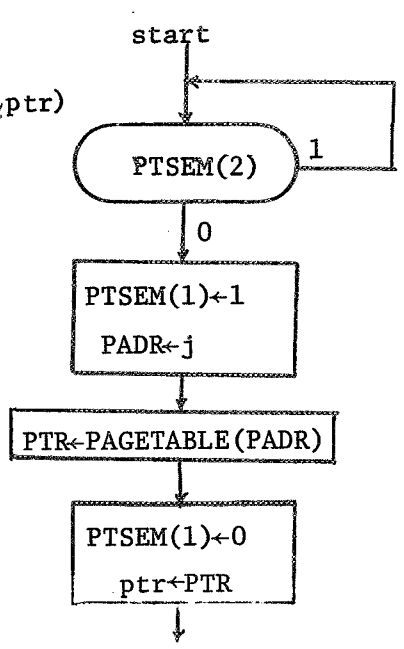
Table 3-4 lists the procedures for accessing the pagetable which load and store page descriptors, and queue (putpt), dequeue (getpt), and detach (detachpt) page descriptors from specified lists. These are complex enough to warrant separate descriptions by sequence chart in Figure 3-4, 3-5, 3-6.

Table 3-5 lists the functions that access the page descriptor fields in the PTR register and the list register fields.

Procedure call	CDL description (micro-operations are sequential)	Explanation
loadpagedescriptor(j,PTR)	see Figure 3-4(a)	load PTR from the jth location of PAGETABLE
storepagedescriptor(j,PTR)	see Figure 3-4(b)	store PTR into the jth location of PAGETABLE
putpt(page,ptl,PTR)	see Figure 3-5(a)	add page descriptor pointed to by <u>page</u> to page list <u>ptl</u>
getpt(page,ptl,PTR)	see Figure 3-5(b)	detach the first pagedescriptor from list <u>ptl</u>; place address in <u>page</u>; leave a copy in the PTR
detachpt(page,ptl,PTR)	see Figure 3-6	detach the page descriptor pointed to by <u>page</u> from list <u>ptl</u>; leave a copy in the PTR

Table 3-4 Page Table Procedures

(a) loadpagedescriptor(j,ptr)



(b) storepagedescriptor(j,ptr)

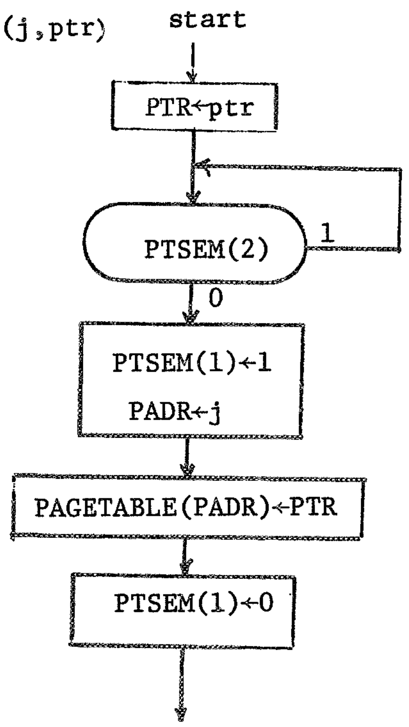
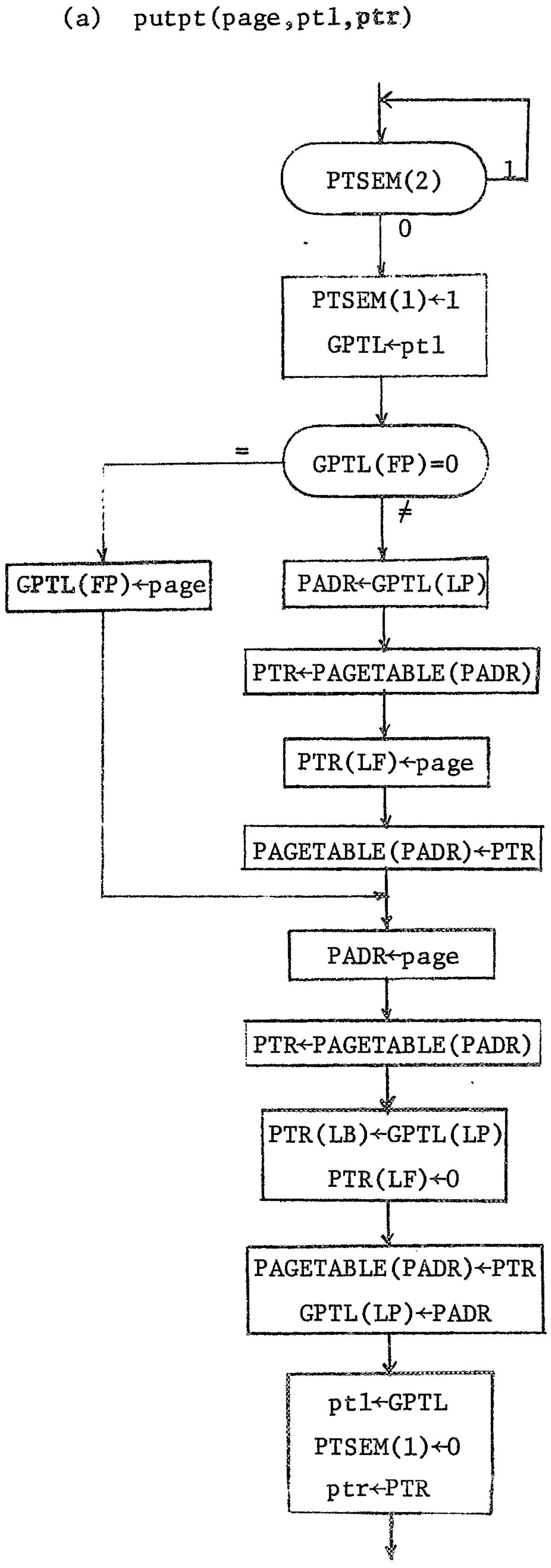


Figure 3-4


(a) putpt(page,ptl,ptr)
PTSEM(2)
1
0
PTSEM(1)←1
GPTL←ptl
GPTL(FP)=0
=
≠
GPTL(FP)←page
PADR←GPTL(LP)
PTR←PAGETABLE(PADR)
PTR(LF)←page
PAGETABLE(PADR)←PTR
PADR←page
PTR←PAGETABLE(PADR)
PTR(LB)←GPTL(LP)
PTR(LF)←0
PAGETABLE(PADR)←PTR
GPTL(LP)←PADR
ptl←GPTL
PTSEM(1)←0
ptr←PTR

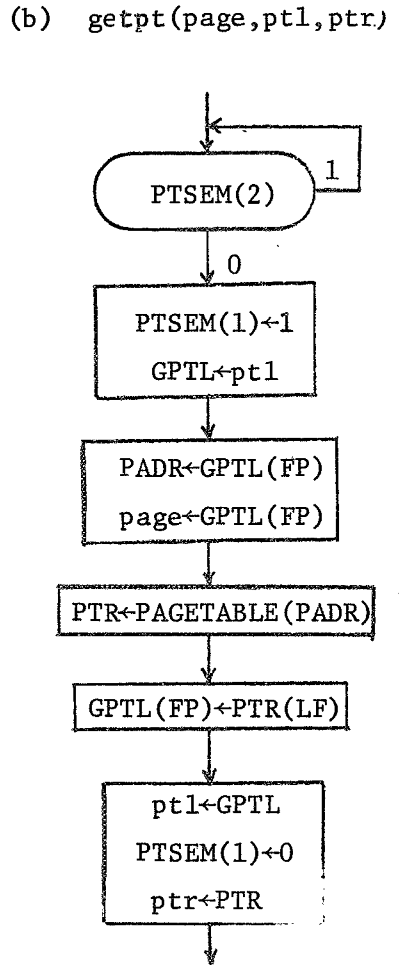

(b) getpt(page,ptl,ptr)
PTSEM(2)
1
0
PTSEM(1)←1
GPTL←ptl
PADR←GPTL(FP)
page←GPTL(FP)
PTR←PAGETABLE(PADR)
GPTL(FP)←PTR(LF)
ptl←GPTL
PTSEM(1)←0
ptr←PTR

Figure 3-5

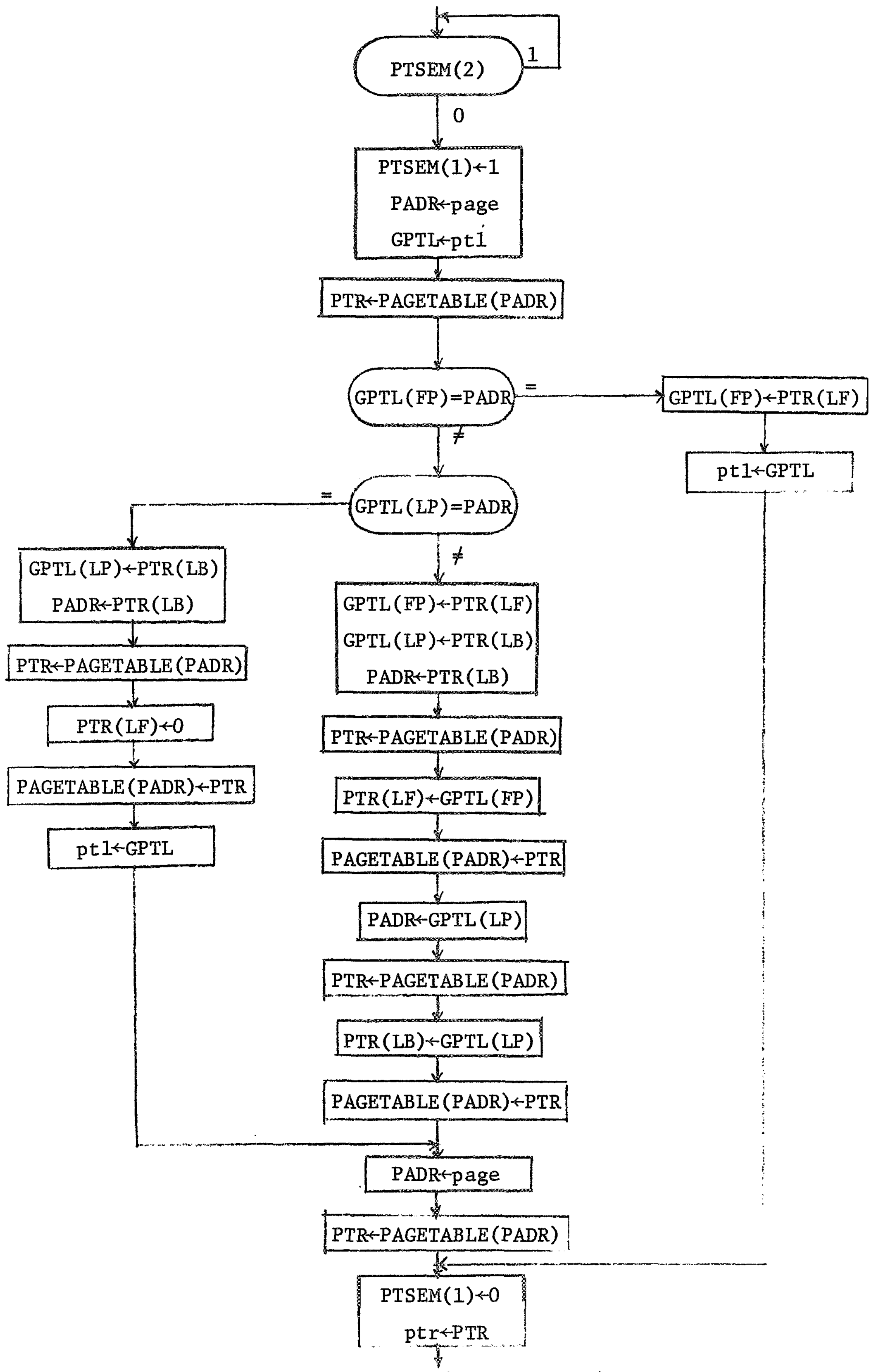


Figure 3-6   detachpt(page,ptl,ptr)

function	CDL description	explanation	
puse(PTR)	PTR(PUSE)	usage field	of PTR
lb(PTR)	PTR(LB)	link back field	of PTR
lf(PTR)	PTR(LF)	link forward field	of PTR
wkey(PTR)	PTR(WKEY)	page key	of PTR
wp(PTR)	PTR(WP)	page write protect bit	of PTR
chge(PTR)	PTR(CHGE)	page altered bit	of PTR
res(PTR)	PTR(RES)	page reserved bit	of PTR
util(PTR)	PTR(UTIL)	utilization field	of PTR
tid(PTR)	PTR(TID)	task identifier	of PTR
vp(PTR)	PTR(VP)	virtual page	of PTR
dp(PTR)	PTR(DP)	drum address	of PTR
row(PTR)	PTR(ROW)	read/write indicator	of PTR
fp(ptl)	ptl(FP)	first page descriptor	of list ptl*
lp(ptl)	ptl(LP)	last page descriptor	of list ptl*

*ptl must be (a) PTLIST
(b) LAVP
(c) LSP
(d) GPTL
(e) PTL

Table 3-5 Page Table Functions

3.4 Paging Drum Channel Subsystem

The paging drum channel subsystem consists of a large drum memory and a dedicated channel processor. The two combine to provide the backing store for the virtual memory system. The drum provides storage for 4096 pages, arranged 16 pages to a drum circumference and 256 pages to the drum width with fixed heads for each track. The drum channel queues requests in 16 separate queues, one queue for each set of 256 pages that come under the read/write heads at once. This allows for optimum average page access times under heavy page traffic conditions.

PAGING DRUM

The paging drum PDRUM is a paging drum containing 4096 pages stored word-parallel in 256 bands, called channels, around the circumference of the drum, 16 pages to a band. Each page occupies one-sixteenth of the drum circumference, called a sector. As the pages are stored word parallel, there are 16K bits per track, and 48 read/write heads per channel. At any one time, the drum address can select one of the 256 channels but must wait for the appropriate sector to come under the heads in order to read the desired page.

The paging drum is described below as a two dimensional memory PDRUM addressed by registers CHANNEL and CWORD and returning a 48-bit word in drum buffer register DBR. Each time DBR is available for reading or writing the buffer status register BS is set to 1, every time a page has been transferred, the register PAGEI is set to 1. Read/write register RW indicates a drum read if RW is set to 0, a drum write if RW is set to 1. If no page is to be transferred, DACTV is set to 0, else it is set to 1.

```
    Comment, paging drum configuration
    Memory,         PDRUM(CHANNEL,CWORD)=PDRUM(0-255,0-16383,1-48)
    Register,       CHANNEL(1-8),       $channel address
```

```
          CWORD(1-14),   $drum channel address
          DBR(1-48),     $drum buffer register
          DACTV,         $drum active register
          RW,            $drum read/write register
          BS,            $buffer status indicator
          PAGEI          $page complete indicator
Subregister, CWORD(SECT)=CWORD(1-4),
          CWORD(PCOUNT)=CWORD(5-14)
```

Figure 3-7 shows the sequence chart for the paging drum. As the drum operation is circular, so the sequence of operations in the chart is circular. Although no transfers are made if DACTV is 0, the drum keeps rotating, represented by the counting operation on CWORD. When reading from drum, BS set to 1 indicates DBR is full, while when writing to drum, BS set to 1 indicates DBR is empty. When CWORD(PCOUNT) becomes zero, a page has been swapped in or out, and the register PAGEI is turned on to indicate this. At this time DACTV is set to zero; therefore, in the time it takes the drum to rotate between words, the paging drum channel must reset CHANNEL, RW, and DACTV and possibly load DBR.

PAGING DRUM CHANNEL

The paging drum channel is defined here as a dedicated processor that, in addition to swapping pages to and from memory, maintains queues of requests in the page table memory -- one for each drum sector, and, as a drum sector comes under the read/write heads, it initiates the read/write indicated by the first request in that sector queue. Upon completion of a page transfer, an interrupt is issued to the CPU in order to release the task and/or page awaiting the request.

The channel configuration, described below in CDL, is centered around two small fast, 16-word memories: the command buffer COM, addressed by address

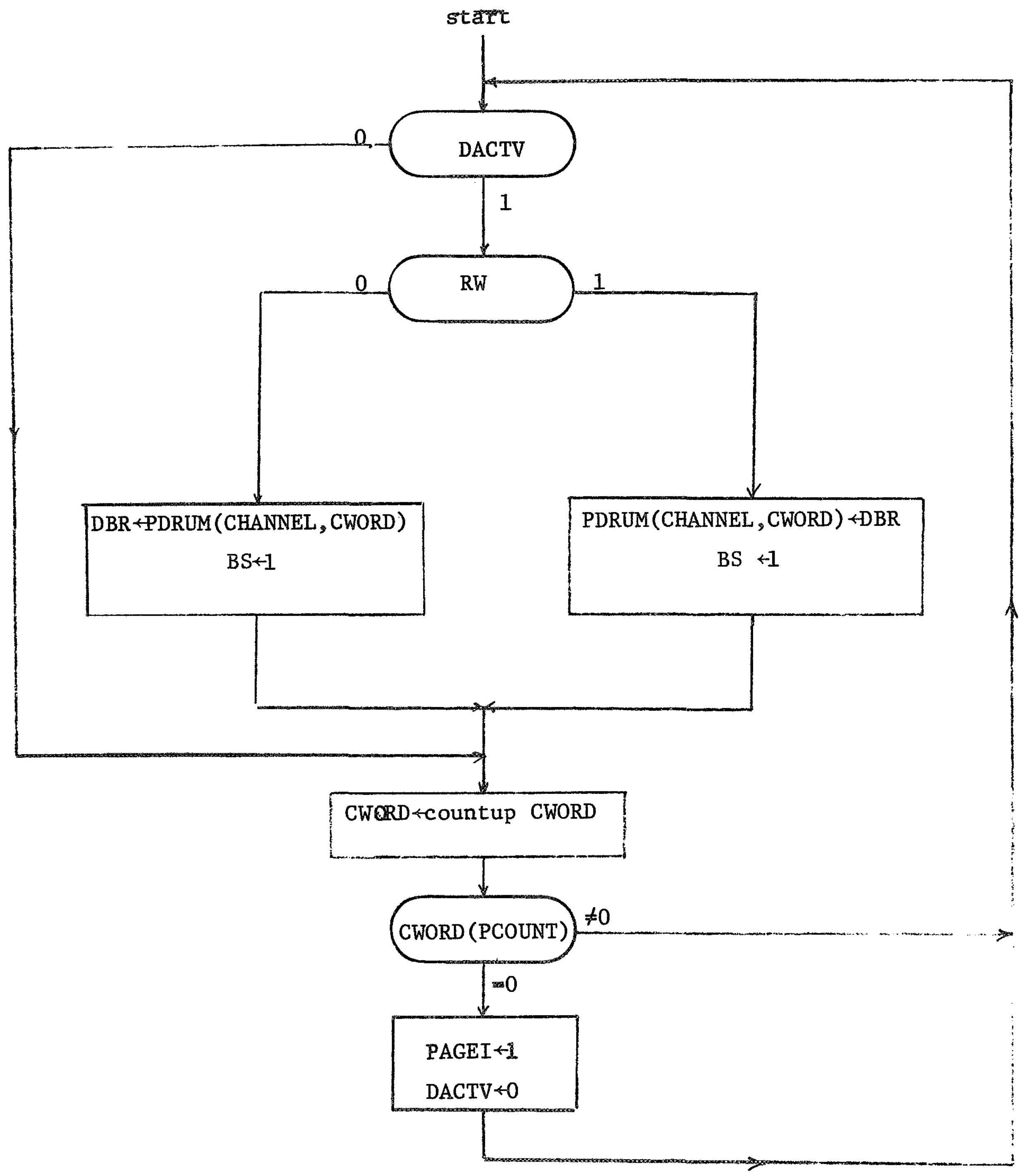


Figure 3-7   Sequence chart for paging drum

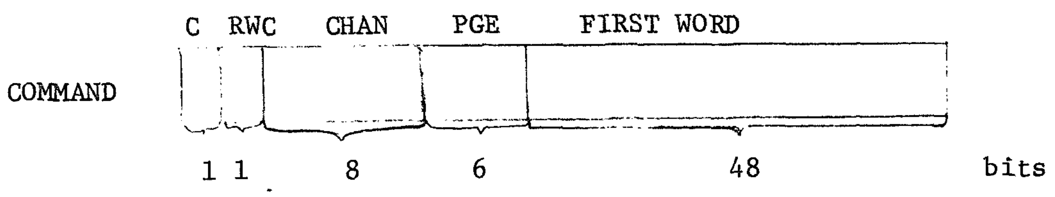
register SEC, and the listhead buffer LISTS, addressed by register SECTORS.

```
   Comment, paging drum channnel configuration

   Memory,     COM(SEC)=COM(0-15,1-64),          $command memory

               LISTS(SECTORS)=LISTS(0-15,1-12)   $pointers to queued requests

   Register,   SEC(1-4),          $address register for command memory

               SECTORS(1-4),      $address register for list-head memory

               COMMAND(1-64),     $buffer register for command memory

               PTR2(1-66),        $page table buffer register (for paging channel)

               PTL(1-12),         $page table  list  register

               PTSEM(1-2),        $page table semaphor

               POST,              $page posting indicator

               PAGPOST(1-6),      $page posted

               CHANNEL(1-8),      $channel address register

               CWORD(1-14),       $channel word address register

               COUNT(1-10),       $word counter

               MADR2(1-16),       $main memory address register (for paging channel)

               SBR2(1-48),        $main memory buffer register (for paging channel)

               DBR(1-48),         $drum buffer register

               RW2,               $main memory read/write register (for paging channel)

               RW,                $drum read/write register

               DACTV,             $drum active register

               BS,                $buffer status register

               PAGEI,             $page complete register

               PTRAN(1-2),        $page transfer direction

               PAGINT,            $page for which interrupt occurred

               MA(1-3),           $main memory access register

               INTERRUPT(1-10)    $CPU interrupt register
```

Subregister, COMMAND(C,RWC,CHAN,PGE,FIRSTWORD)=COMMAND(1,2,3-10,11-16,17-64),
PTL(FP,LP)=PTL(1-6,7-12),
MADR2(BLOCK,WRD)=MADR2(1-6,7-16),
INTERRUPT(PAGE)=INTERRUPT(4),
CWORD(SECT,COUNT)=CWORD(1-4,5-14),
INTERRUPT(DRUMPAGE)=INTERRUPT(10),
PTR2(CH,SEC,ROW)=PTR2(54-61,62-65,66),

Each drum sector has an associated queue of page requests. The first request on the queue (the next to be serviced) is maintained in a command buffer location. The rest of the requests are linked in a doubly-linked list in the pagetable memory PAGETABLE, and this list is identified by a 12-bit list head which points to the first and last entries on the list and is stored in the listhead memory LISTS. Buffer register COMMAND and address register SEC are used to access the command memory. Address register SECTORS, listhead buffer register PTL, page table access register PTSEM and pagetable buffer register PTR2 are used to access the pagetable and listhead memories. Address registers CHANNEL and CWORD, buffer register DBR, and status registers PAGEI, RW, DACTV, and BS are used to access drum memory. Address register MADR2, buffer register SBR2, read/write register RW2, and access register MA are used to access main memory. Registers INTERRUPT, PAGINT, and PTRAN are used to notify the CPU of a completed page transfer and/or error. Registers POST and PAGPOST are used by the CPU to post a page request with the drum channel. Counter COUNT is used to compute the memory pageword address and to check for complete page transfers.

Figure 3-8 displays the format of the COMMAND and PTL buffer registers. If the command word contains in COMMAND is useful (i.e., has not been used), then C is set to 1, RWC indicates the read/write, CHAN contains the drum channel (the command word address is the sector number), PGE contains the main memory page address, and in the case of a drum write FIRSTWORD contains the



C=0 if page has been swapped

=1 if page has not been swapped

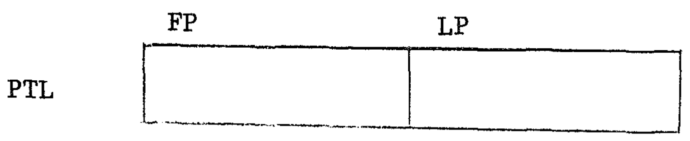
RWC=0, if page is to be swapped in (read)

=1, if page is to be swapped out (written)

CHAN= drum channel address of page

PGE= memory page address to be swapped

FIRSTWORD= first word of page to be swapped (if RWC=1)



FP= first page of queue of this sector

LP= last page of queue for this sector

Fig. 3-8 Format of COMMAND and PTL registers

first word of the page to be swapped. Register PTL has the standard page list-head format.

Table 3-6 presents the procedure calls associated with the paging drum channel. The first three (putpt, getpt, loadpagedescriptor), are used as shorthand for the sequence chart in Figure 3-11. They represent slightly altered forms of sequence charts already presented, in Figure 3-5a, 3-5b, and 3-4a, respectively. The queuerequest procedure will be used in section 4.3, but is useful here in indicating how the CPU posts a page request with the drum channel. The sequence chart appears in Fig. 3-9. It should be noted that the logic does not allow a page request to be lost if the channel cannot handle a previous request before the next request is made. In this case, the CPU "hangs up" (i.e., loops until the previous request is handled).

Figure 3-10 presents a descriptive flow chart of the sequence chart presented in Fig. 3-11. The reader is urged to study the details of 3-11 carefully. In particular, it should be noted that once a new channel command is initiated, two processes occur in parallel: the left-hand subsequence (beginning with the DACTV branch) handles the page swapping, and the right-hand subsequence (beginning with SECTORS←SEC) loads the command buffer with the next command for the current sector, if there is one, and then cycles and waits for page requests to be posted by the CPU.

In the right-hand subsequence, the first step is to check the list-head of the current sector to see if the list is empty (PTL(FP)=0). If it is, then the completion bit (COMMAND(C)) is set to zero, indicating that there is no command for this sector. However, if the list is not empty, then the first entry on the list pointed to by PTL is removed (getpt(PG,PTL,PTR2)), and the command buffer is loaded with the memory page address (PG), channel address (PTR2(CH)), and read/write indicator (PTR2(ROW)). If this is a write to drum (PTR2(ROW)=1) then in addition the first word of the page is loaded to the

procedure call	CDL description	explanation
putpt(page,PTL,PTR2)	Figure 3-5(a)*	place page page on the queue PTL using PTR2 as a buffer register
getpt(page,PTL,PTR2)	Figure 3-5(b)*	get the first page page on queue PTL and leave a copy in PTR2
loadpagedescriptor(page,PTR2)	Figure 3-4(a)*	load PTR2 with a copy of page descriptor page
queue request(page)	Figure 3-9	queue page page for swapping

* except substitute PTR2 for PTR; switch PTSEM(1) and PTSEM(2)

Table 3-6 Procedure Call for Paging Drum Channel

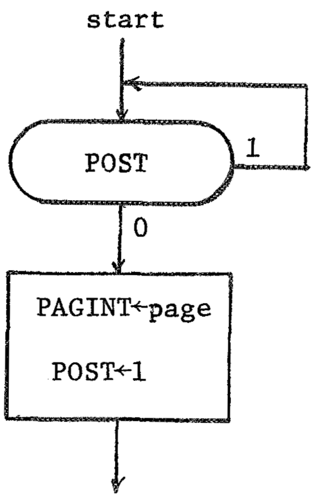


Figure 3-9 queue request(page)

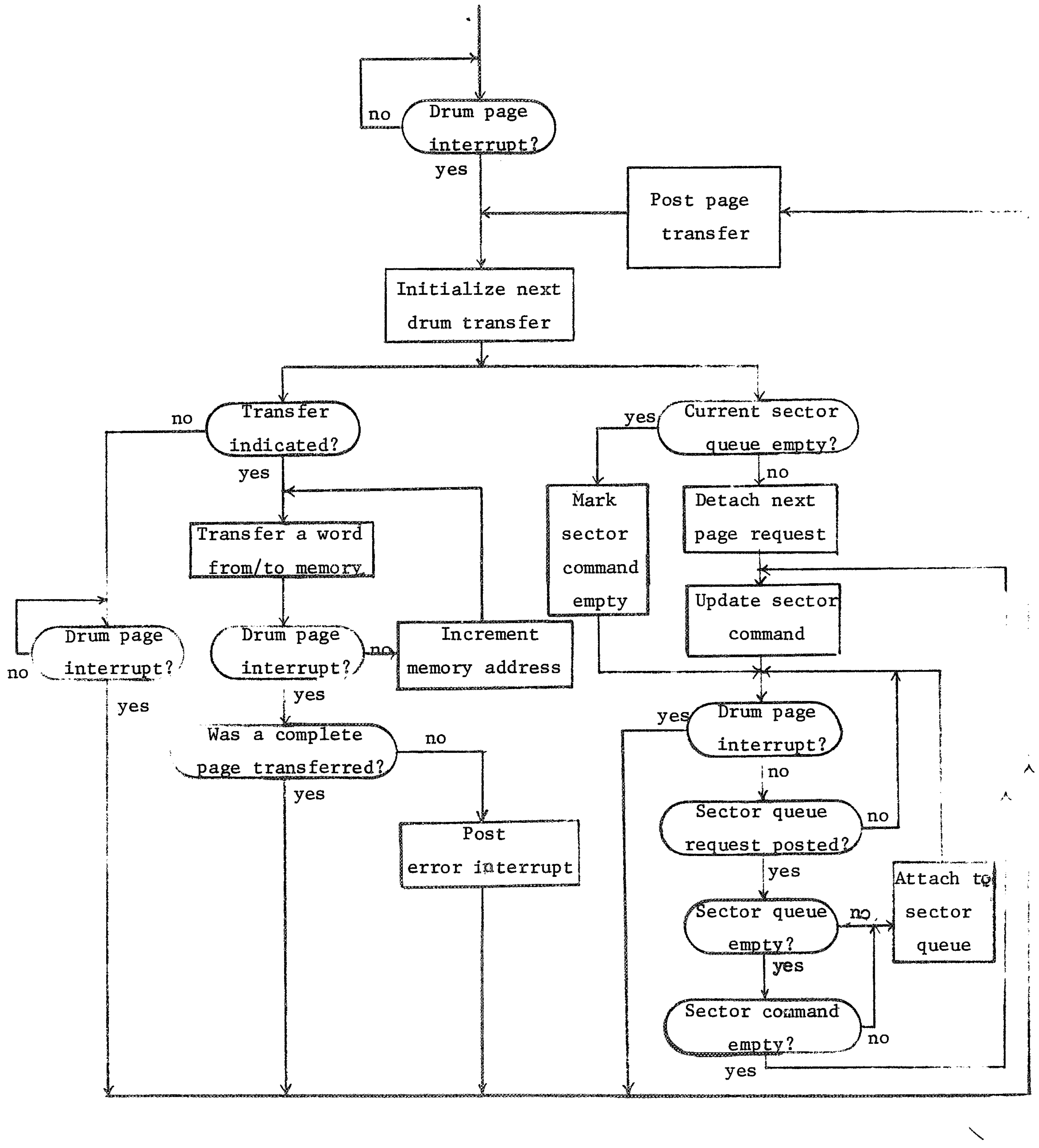


Figure 3-10 Descriptive flow chart of paging drum channel

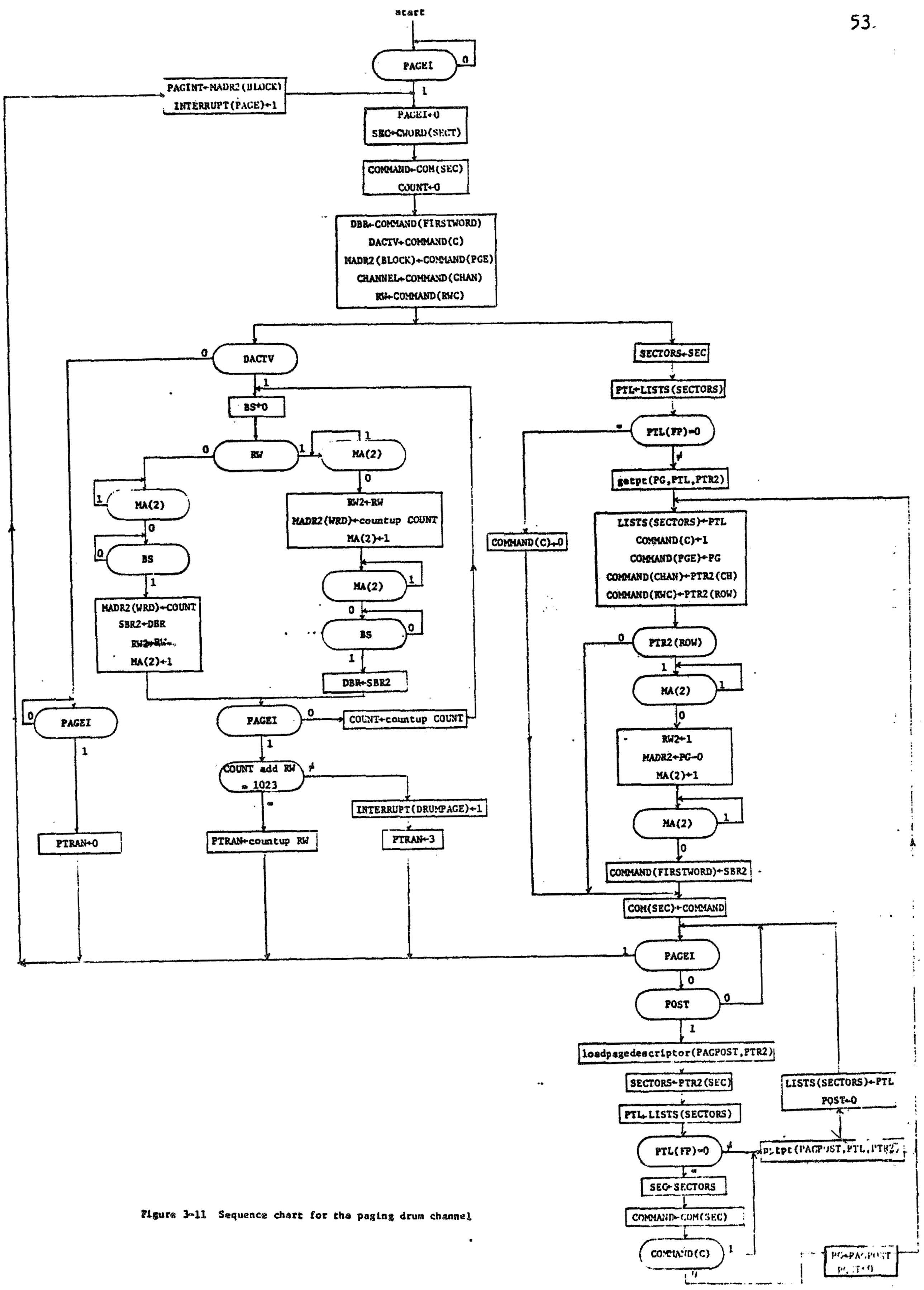


Figure 3-11 Sequence chart for the paging drum channel

command buffer. This reduces the amount of time required to initiate the page swap on the next drum revolution. Once the command word has been reloaded to the command buffer memory, the subsequence monitors the POST register. If the CPU posts a page request, the subsequence checks to see if the command buffer entry for that sector has its C subregister set to zero. If so, then this new page request is loaded directly to the command buffer, else it is added to the list of page requests for this sector.

The left-hand subsequence interfaces with the sequence chart in Figure 3-7 (the drum sequence). If a page is to be transferred (DACTV≠0), then if RW=0 the transfer is from drum to memory, else (RW=1) the transfer is from memory to drum. In the former case (drum to memory), when the drum signals the buffer (DBR) has been loaded (BS=1), then the word is written to the main memory. The earlier test on MA(2) assures us that the previous memory write was compiled. In the case of a transfer from memory to drum, the first word is already in DBR and the sequence signals the memory to load SBR2 with the next word, waits for the drum to signal the completion of the transfer (BS=1) and then reloads DBR. At the completion of each one-word transfer, the PAGEI register is checked to see if the drum has reached a page boundary. If not, the COUNT register is incremented and another word is transferred. If so, then the subsequence tests to see if a complete page was transferred (COUNT add RW = 1023). This curious equation is a result of the fact that on a transfer from memory to drum (RW=1), the subsequence passes through the loop one time less (as DBR is loaded before the loop is entered).

Upon completion of a page swap, register PTRAN is set to indicate the status: (a) no swap (PTRAN=0), (b) page read to main memory (PTRAN=1), (c) page written to drum (PTRAN=2), or (d) incomplete swap (PTRAN=3); register PAGINT is loaded with the page address of the page swapped, and interrupt subregister INTERRUPT(PAGE) is set to one.

3.5 Interrupt and the CPU

The physical configuration of the central processing unit (CPU) is purposefully left undefined, except for the addressing and interrupt handling connected with task scheduling. In defining the CPU further, the topic would be broadened and, at the same time, restricted unnecessarily. Indeed, the address format described in section 3.2 makes assumptions about indirect addressing and indexing, although merely intended to be exemplary!

Table 3-7 lists the INTERRUPT single-bit subregisters and their interpretation. These include an interval timer interrupt bit, blocking and unblocking interrupt bits, a page interrupt bit, a task completion bit, task suspension bits and various error interrupt bits.

At this point it is instructive to regard Figure 3-1 once again, which displays the registers with which the subsystems inter-communicate. The relationship of the pagetable, main memory and paging drum with the paging drum channel has now been defined. In addition, the processors' mutually exclusive sharing of the pagetable and main memories via the MA and PTSEM access registers has been outlined.

The primitive procedure calls defined in this section will be used in the next section, in which the logical structure of the memory addressing, memory management, and task scheduling will be presented as a series of ALGOL procedures.

name	CDL description	explanation
timer(INTERRUPT)	INTERRUPT(TIMER)=INTERRUPT(1)	indicates interval timer has gone to zero
blocked(INTERRUPT)	INTERRUPT(BLOCKED)=INTERRUPT(2)	indicates the current task has been blocked
unblocked(INTERRUPT)	INTERRUPT(UNBLOCKED)=INTERRUPT(3)	indicates a task may be un-blocked
page(INTERRUPT)	INTERRUPT(PAGE)=INTERRUPT(4)	indicates a page has been swapped (in or out)
complete(INTERRUPT)	INTERRUPT(COMPLETE)=INTERRUPT(5)	indicates the current task has completed
deactpw(INTERRUPT)	INTERRUPT(DEACTPW)=INTERRUPT(6)	indicates current task should be deactivated to the page wait list
deactrl(INTERRUPT)	INTERRUPT(DEACTRL)=INTERRUPT(7)	indicates current task should be deactivated to ready list
susp(INTERRUPT)	INTERRUPT(SUSP)=INTERRUPT(8)	indicates current task should be suspended but not deac-tivated
addressfault(INTERRUPT)	INTERRUPT(ADDRESSFAULT)= INTERRUPT(9)	indicates an address fault (error condition)
drumpage(INTERRUPT)	INTERRUPT(DRUMPAGE)= INTERRUPT(10)	indicates an incomplete page transfer (error condition)

Table 3-7 Interrupt subregisters

4. SYSTEM DESCRIPTION

This section presents the detailed description of the virtual memory addressing, task scheduling, and memory management. The description is presented in ALGOL. Except for the description of the system task (Procedure 4-27), the ALGOL descriptions represent microprograms and/or hardwired sequences that control the fetch-execute cycle, address translation, memory access, and interrupt processing. Specifically, the hardware sequences described in section 3 are considered to be procedures and are referred to in standard ALGOL fashion. Fields of registers are referenced as function procedures with the register appearing as an argument of the procedure statement. Notationally, three conventions are adopted:

(1) ALGOL delimiters are represented lower-case and underlined (e.g., procedure, begin, end, if, then, else, etc.),

(2) procedure names (in both declarations and statements) are in lower-case,

(3) variables are in upper-case.

The data structures for the task lists and procedures that manipulate the data structures are presented first, followed by descriptions of miscellaneous procedures, and finally the system algorithm.

4.1 Data Structures

The data structures used in the virtual memory subsystem consists of two types: lists of task descriptors and lists of page descriptors. The lists of page descriptors have been described previously in the memory management description (section 3.3). Task descriptors reside in main memory, and consist of a block of words containing pointers to the previous and following

4. SYSTEM DESCRIPTION

This section presents the detailed description of the virtual memory addressing, task scheduling, and memory management. The description is presented in ALGOL. Except for the description of the system task (Procedure 4-27), the ALGOL descriptions represent microprograms and/or hardwired sequences that control the fetch-execute cycle, address translation, memory access, and interrupt processing. Specifically, the hardware sequences described in section 3 are considered to be procedures and are referred to in standard ALGOL fashion. Fields of registers are referenced as function procedures with the register appearing as an argument of the procedure statement. Notationally, three conventions are adopted:

(1) ALGOL delimiters are represented lower-case and underlined (e.g., procedure, begin, end, if, then, else, etc.),

(2) procedure names (in both declarations and statements) are in lower-case,

(3) variables are in upper-case.

The data structures for the task lists and procedures that manipulate the data structures are presented first, followed by descriptions of miscellaneous procedures, and finally the system algorithm.

4.1 Data Structures

The data structures used in the virtual memory subsystem consists of two types: lists of task descriptors and lists of page descriptors. The lists of page descriptors have been described previously in the memory management description (section 3.3). Task descriptors reside in main memory, and consist of a block of words containing pointers to the previous and following tasks in the list, working set information, priority information, scheduling information, a copy of the task's portion of the translation memory, a buffer for its copy of the registers, a pointer to its set of active pages on the page table list, and, if the task is blocked, a pointer to its set of locked-in pages. There are five task lists, each paired with a hardware register containing a pointer to the first task descriptor in the list and a count of the number of task descriptors on the list:

(a) read list (pointer register READYLIST). A doubly-linked list of tasks to be activated when space exists in memory for their working sets.

(b) running list (pointer register RUNLIST). A circular, doubly-linked list of tasks that are being processed in round-robin fashion (i.e., active tasks).

(c) blocked list (pointer register BLOCKLIST). A doubly-linked list of tasks awaiting completion of another task (e.g., an I/O request).

(d) page wait list (pointer register PAGEWAIT). A queue of tasks awaiting completion of a page write so that they may access the page.

(e) task complete list (pointer register COMPLETELIST). List of complete tasks.

The CDL description of the pointer registers and subregisters appears below.

```
Comment, task descriptor list pointers

Register,     READYLIST(1-24),     $ready list

              RUNLIST(1-24),       $running list

              PAGEWAIT(1-24),      $page wait list

              BLOCKLIST(1-24),     $blocked list
```

```
                COMPLETELIST(1-24)     $completed task list
  Subregister,  READYLIST(NUMBER,F)=READYLIST(1-8,9-24),
                RUNLIST(NUMBER,F)=RUNLIST(1-8,9-24),
                PAGEWAIT(NUMBER,F)=PAGEWAIT(1-8,9-24),
                BLOCKLIST(NUMBER,F)=BLOCKLIST(1-8,9-24),
                COMPLETELIST(NUMBER,F)=COMPLETELIST(1-8,9-24),
```

The page list pointers have a subregister NUMBER (e.g., READYLIST(NUMBER)) which contains a count of the number of entries on the list, and a subregister F (e.g., READYLIST(F)) which contains the physical address of the first task descriptor on the list. The function procedures for accessing the contents of these subregisters are shown in Table 4-1.

Rather than describe the format of the task descriptor block, a list of its contents is shown in Table 4-2. The list consists of a function procedure name with which to access a field of the task block, the minimum size of the field (in bits), and an explanation. These fields are concerned with timing (_active time_, _quantum_, _interval value_), working set sizes (_wssize_, _wsold_), page table lists (_tasklist_, _lockedinlist_), translation memory (_tmcount_, _sact_, _swkey_, _swp_, _sres_, _sblk_, _sdp_), and task and page status (_blkd_, _key_, _resident_, _first request_, _desired virtual page_). Two fields link the task to other tasks in the list (_nexttask_, _previoustask_).

All task lists are stored as circular, doubly-linked lists and are thought of as queues (i.e., having a front and a back). The task list pointer points to the front of the list and the _previous task_ field of the first task descriptor points to the back of the list. Four procedures (Procedures 4-1, 4-2, 4-3, 4-4) display procedures for operating on the task lists. Procedure _gettask_ (4-1) detaches the first task descriptor from a specified list and returns the address of the task descriptor. Procedure _puttask_ (4-2) attaches a specified task to the end of a specified list. Procedures _getnexttask_ (4-3)

function	CDL description	explanation
number(t)	t(NUMBER)	number of entries in the task list identified by register t*
f(t)	t(F)	first task on the task list identified by register t*

* t ≡ READYLIST, RUNLIST, PAGEWAIT, BLOCKLIST, or COMPLETELIST

Table 4-1 Function procedures for task list pointers

function procedure name	min. size (bits)	explanation
1) active time (T)	-	elapsed processing time since task was made active
2) quantum (T)	-	quantum of processing time allocated to the task
3) interval value (T)	-	remaining time in the current burst interval
4) wssize (T)	6	current working set size for task
5) wsold (T)	6	working set size at termination of last active period
6) translation memory		a set for each translation memory entry
sact(P,T)	1	active flag } for virtual page P of task T
swlay(P,T)	4	page key } for virtual page P of task T
swp(P,T)	1	write protect flag } for virtual page P of task T
sres(P,T)	1	resident flag } for virtual page P of task T
sblk(P,T)	6	memory page } for virtual page P of task T
sdp(P,T)	12	drum address } for virtual page P of task T
7) tasklist (T)	12	page list pointers for task T
8) desired virtual page (T)	8	most recently requested virtual page
9) locked in list (T)	12	list pointers for pages locked-in during a process block
10) first request (T)	8	first virtual page necessary to activate task T
11) resident (T)	1	flag signalling that the task is resident
12) key (T)	4	page key for task T
13) blkd (T)	1	flag indicating the task has active pages but is suspended from processing awaiting a page or task unblocking
14) tmcount (T)	10	number of translation memory locations
15) nexttask (T)	16	next task in the list
16) pervious task (T)	16	previous task in the list

Table 4-2 Task field functions

points the tasklist pointer for a specified list at the next task in the task list and returns the address of that task descriptor. Procedure detachtask (4-4), detahces a specified task from a specified list.

4.2 Primitive Procedures and Global Variables

The hardware components of the virtual memory system are presented in section 3 in CDL. In order to make use of the hardware registers and micro-instruction sequences presented there, suitable ALGOL counterparts are assigned. Registers are defined as global variables, and appear in Table 4-3. Micro-instruction sequences are assigned procedures names, with most of the relevant variables (registers) appearing as format parameters. Table 4-4 summarizes those procedures previously defined. Subregisters are defined as function procedures. A list of registers for which function procedures are defined appears in Table 4-5.

In addition, there are a number of procedures that perform simple functions that operate on the contents of main memory and use the procedures defined in Table 4-4 or perform basic operations on pieces of hardware not described completely (e.g., the interval timer) or perform necessary operations that can be described without specifying detailed operation. These procedures are listed in Table 4-6. Procedures gettask, puttask, getnexttask, and detach-task were presented in section 4-1 and manipulate the task descriptors in the various lists. Two procedures, load state vector (4-5) and unload state vector (4-6), load and unload a task's register set and translation memory. They are each made up of two other procedures. Unloadtm (4-7) scans the translation memory for entries that have been referenced and changes the corresponding page descriptor and the task's copy of the entry to reflect the changes found in the entry. It should be noted that this procedure in effect unloads the trans-

name	definition
TASK	address of descriptor block for current task
SYST	address of descriptor block for the resident system task
READYLIST	pointer to the list of tasks ready for activation
RUNLIST	pointer to the list of active tasks
PAGEWAIT	pointer to the list of tasks waiting for a page assignment
BLOCKLIST	pointer to the list of tasks blocked, awaiting completion of another task
COMPLETELIST	pointer to the list of complete tasks
INTERRUPT	interrupt register
PAGINT	register that indicates the page which has just been swapped
PTRAN	register that indicates the type of swap which just occured
TASKINT	register that indicates the task that was just unblocked
PTR	page table descriptor register
TMR	translation memory buffer register
LSP	pointer to the list of pages that must be swapped out
LAVP	pointer to the list of available pages
CAVPA	count of available pages
CAVPV	count of pages in LSP and LAVP
AVPC	count of available pages not yet assigned to working sets
AFLAG	flag indicating a memory access error (illegal key)
WFLAG	flag indicating an attempt to write a read-only page
β	length of time allowed for one burst of processing
τ	working set parameter (in units of time)
MTKEY	value of protection key which indicates the page is outside program limits

Table 4-3 Global variables

procedure call	description	explanation
loadmm(i,j,SBR)	Table 3-1	load SBR from main memory page i, word j
storemm(i,j,SBR)	Table 3-1	store SBR to main memory page i, word j
loadtmr(i,TMR)	Table 3-2	load TMR from translation memory location i
storetmr(i,TMR)	Table 3-2	store TMR to translation memory location i
loadpagedescriptor(j,ptr)	Figure 3-4(a)	load ptr from the jth location of the page table
storepagedescriptor(j,ptr)	Figure 3-4(b)	store ptr to the jth location of the page table
putpt(p,ptl,ptr)	Figure 3-5(a)	add the page descriptor at address P to the page list ptl
getpt(p,ptl,ptr)	Figure 3-5(b)	detach the first page descriptor at page list ptl, place address in p, contents in ptr
detach(p,ptl,ptr)	Figure 3-6	detach the page descriptor at address P from list ptl, leave contents in ptr
queue request(p)	Figure 3-9	queue page p for swapping

Table 4-4. Summary of primitive procedure defined in section 3

register	location of field descriptions	buffer name
TMR	Table 3-2	translation memory buffer
VAD	Table 3-2	virtual address register
MADR1	Table 3-2	main memory address register
PTR	Table 3-5	page table buffer (for CPU)
PTR2	Table 3-5	page table buffer (for paging channel)
PTLIST	Table 3-5	page list pointer (for CPU)
LAVP	Table 3-5	list of available pages pointer
LSP	Table 3-5	list of swappable pages register
PTL	Table 3-5	page list pointer (for paging channel)

Table 4-5 Summary of registers that are referenced by subregister name

procedure call	description	explanation
gettask(T,LIST)	procedure 4-1	removes the first task on list LIST, places address in T
puttask(T,LIST)	procedure 4-2	places task T into list LIST
getnexttask(T,LIST)	procedure 4-3	moves list pointer LIST to point to the next task on the circular LIST, $T \leftarrow f(LIST)$
detachtask(T,LIST)	procedure 4-4	detachtask T from list LIST
loadstatevector(TASK)	procedure 4-5	load registers and translation memory for task TASK
unloadstatevector(TASK)	procedure 4-6	unload and store registers and translation memory for task TASK
unloadtm(TASK)	procedure 4-7	unload translation memory for task TASK
loadtm(TASK)	procedure 4-8	load translation memory for task TASK
store registers(TASK)	*	store registers for task T
restore registers(TASK)	*	load registers for task T

*not defined

Table 4-6 Other primitive procedures

procedure call	definition	explanation
load interval timer (INTVAL)	*	load interval timer with the value of INTVAL
store interval timer (INTVAL)	*	store the contents of the interval timer into INTVAL
suspend interval timer	*	suspend the interval timer
restore interval timer	*	restart the interval timer
cannabalize(PAGE,TASKLIST)	*	TASKLIST is searched for the best candidate for swapping, this page is detached, its address placed in PAGE, and the page is queued as available of for swapping. CAVPA, CAVPV and AVPC are adjusted.
search ready list for task(T) whose working set size(WSIZE) is less than (AVPC) and note (FAILURE)	*	the ready list is searched for the highest priority task T. If the working size WSIZE is less than the available page count AVPC, then FAILURE←0, else FAILURE 1
errorstop (TASK)	*	processes address and hardware errors

*not defined

Table 4-6 (continued)

lation memory, while at the same time recording the page utilizaton. Loadtm (4-8) loads the translation memory with a task's copy of the translation memory. The entries for the pages that are not used are loaded with a key that will generate an "exceeded bounds" fault if they are addressed. The procedures store registers and load registers save a task's register set and restore the task's registers, respectively. These procedures are not shown in any detail for two reasons: all registers are not defined and the operation is obvious.

The interval timer is not described in any detail but plays an important role in the task scheduling implementation. Four procedures are provided to save and restore its contents (procedures store interval timer and load interval timer, respectively) and to suspend and restore its operation (suspend interval timer and restore interval timer, respectively).

Three procedures perform necessary tasks but are better left vague until the complete operating system is defined. Procedure cannabolize is a special procedure that is used to select the best candidate for swapping from a task's working set. It is only called when one task, other than the system task, resides on the running list and no pages are available. The choice of a best candidate is not simple. Procedure search ready list for task searches the ready list for a task whose working set would fit into the available space in memory. This requires knowledge of the priority algorithm, and as determination of priority is a subject in itself, the details are not given here. Procedure errorstop is defined as the system addressing error and hardware error handler. Again, this is considered to be outside the range of this paper and, therefore, details are not given.

All procedures defined above are called primitive procedures. Although each may perform a complex task, the functions of each are self-contained. Their definition as procedures will make the complete system description more general.

4.3 Memory Management - Task Scheduling Algorithm

The memory management, task scheduling algorithm consists of three parts: address translation, interrupt processing, and the system task. Address translation occurs during the CPU's control (fetch-execute) cycle. As the author wishes to avoid presentation of instruction formats, the details of the control cycle will not be presented. It will be sufficient to state that at some, possibly several, points in the control cycle instruction or operand fetchs occur. A possible control cycle is shown in Figure 4-1 that has cascading indirect addressing for operand fetch/store and an instruction fetch.

Each address to the user's virtual name space must be translated. This address translation may be successful, may generate a pagefault, or may generate an error. Figure 4-2 presents the flow chart for the major CPU cycle (procedure 4-9). The control cycle appears as a procedure call. In this procedure it is assumed that a request is made for address translation, to a procedure called vmtranslate. Procedure vmtranslate (4-10) described in detail later, performs the function of address translation.

If the reader refers back to Figure 2-4, the scheduling implementation diagram, he can see that task scheduling is primarily interrupt driven. The task is removed from the processor under one of five conditions: (a) after an interrupt from the interval timer, (b) after a page fault, (c) after task completion, (d) after blocking to await completion of another task, or (e) after exceeding the working set size with no pages available (working set overflow). All except the first may be considered to be generated by the task itself, but they are interrupts to normal task execution. Any of these interrupts cause task rotation or task deletion in the running list and possibly task addition to the completed list, ready list, blocked list, or page wait list.

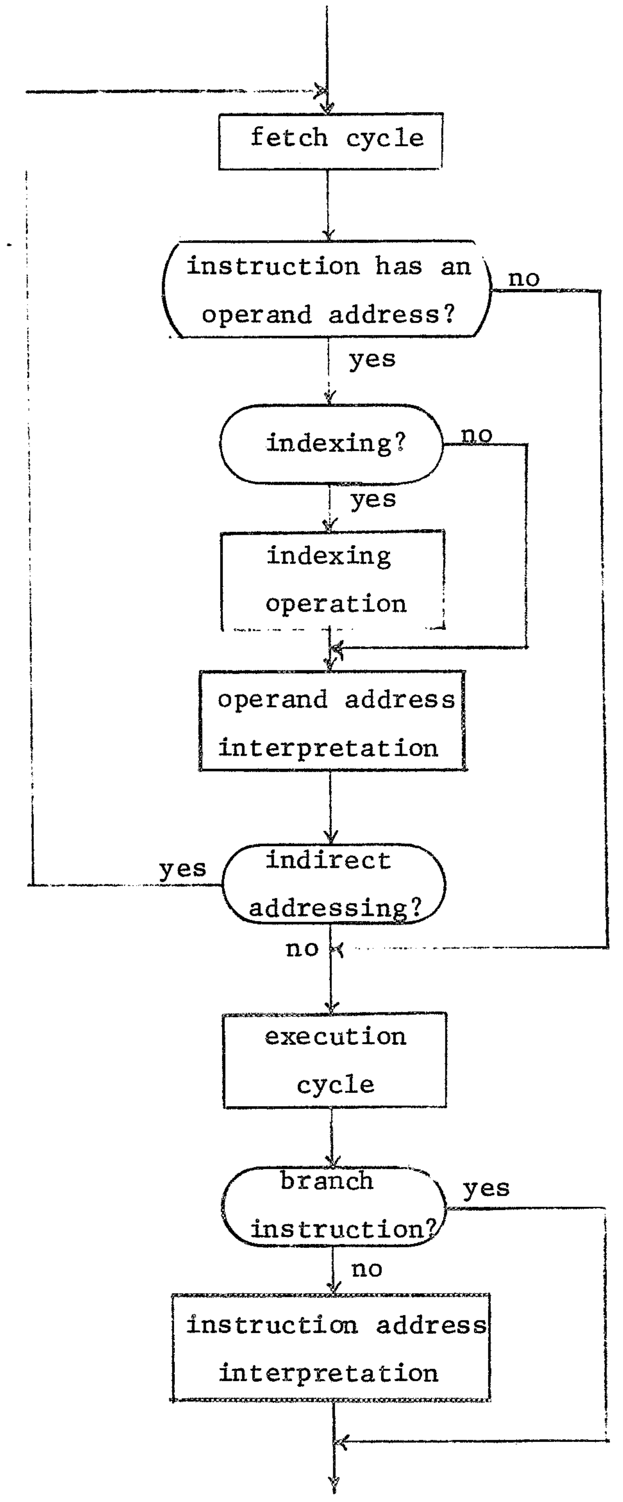


Figure 4-1 A possible control cycle

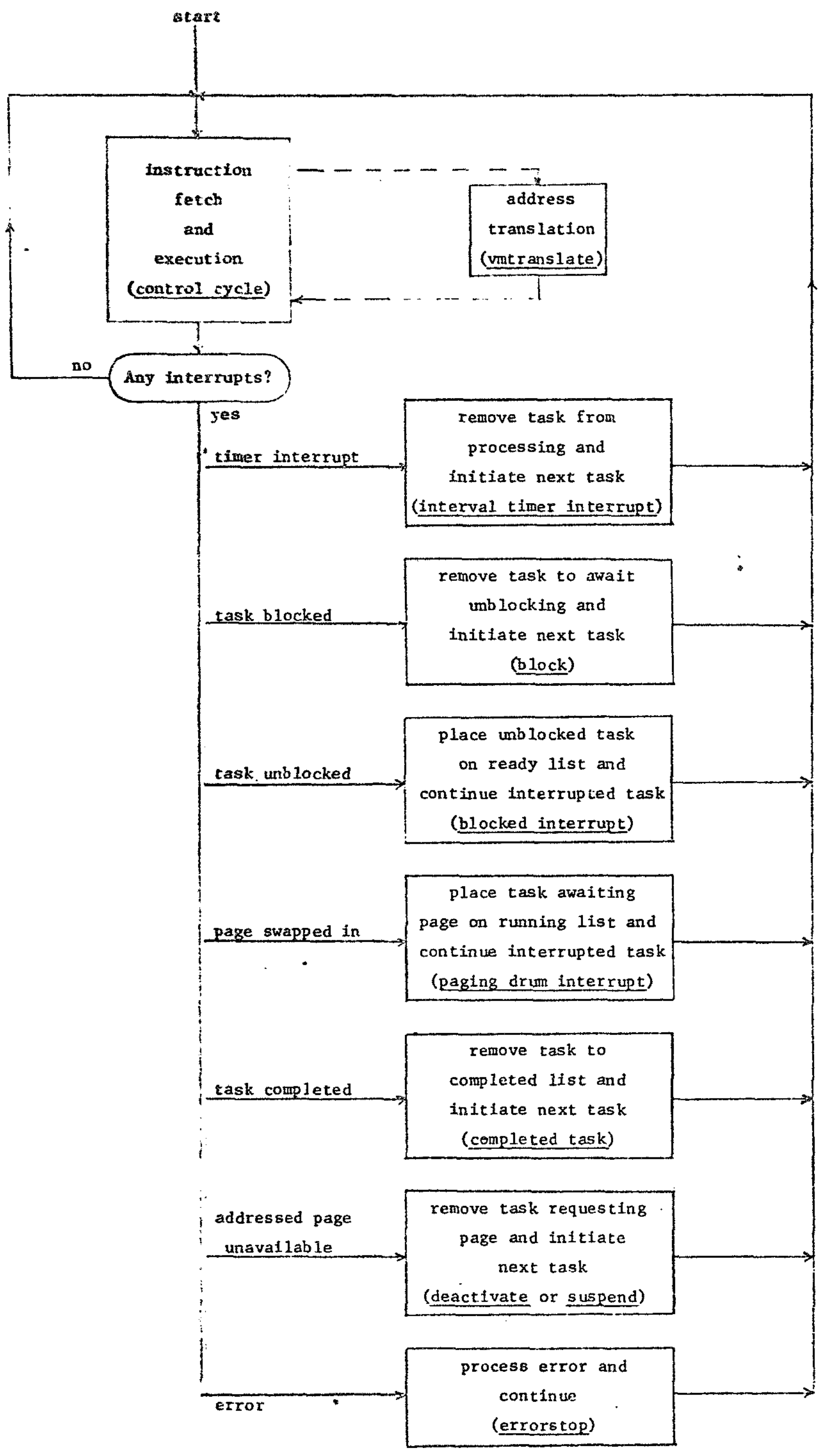


Figure 4-2 Flow chart showing the functions of the major CPU cycle

Tasks move from page wait to the running list after an interrupt from the paging drum. Tasks leave the blocked list and move to the ready list after an interrupt signals that block has been removed. Figure 4-2 shows that the second part of the major CPU cycle consists of testing for and handling the various interrupts.

In Figure 2-4, only one task movement remains, from the ready list to the running list. This function is performed by the resident system task. The resident system task performs a portion of the memory management task, namely the monitoring and adjustment of the working sets. Therefore, only the system task can recognize when space exists for activating a task and its working set.

The remaining portion of section 4 presents the ALGOL procedural descriptions of the address translation, interrupt handling, and system task.

MAJOR CPU CYCLE

The major CPU cycle (procedure 4-9) has been just outlined. It consists of a program loop, apparently endless (system shutdown is left undefined). The control cycle is executed and re-executed unless an interrupt occurs (INTERRUPT≠0). When an interrupt is recognized by the major CPU cycle the interval timer is suspended, the interrupt is processed, the interval timer is restored, and the cycle continues. There are ten possible interrupts that can occur:

(a) <u>interval timer</u>, in which case procedure interval <u>timer interrupt</u> (4-16) causes the task to be removed from processing;

(b) <u>task blocked</u>, in which case procedure <u>block</u> (4-20) places the task on the blocked list;

(c) <u>task unblocked</u>, in which case procedure <u>blocked interrupt</u> (4-21) places the task on the ready list;

(d) page interrupt, in which case procedure paging drum interrupt (4-23) determines the action to be taken with the released page;

(e) task completion, in which case procedure completed task (4-25) deactivates the task to the completed task list;

(f) page wait(1), in which case procedure deactivate (4-17) places the task on the list PAGEWAIT;

(g) working set overflow, in which case procedure deactivate (4-17) places the task on the ready list;

(h) page wait(2), in which case procedure suspend (4-26) removes the task from the running list;

(i) address fault, in which case procedure errorstop handles the error;

(j) drum page fault.

Several comments should be made at this point. In the above scheme, interrupts a, b, e, f, g, h, and i must occur mutually exclusive of each other, as each involves an action with the current task (and each changes the current task pointer). In all other cases, some other uniquely specified task is involved. For this reason, the following clarifications are made about the control cycle operation:

(a) the posting of interrupt b, e, f, g, h, or i cause the resetting of the timer interrupt (off);

(b) the timer interrupt will not be set while interrupt b, e, f, g, h, or i are on;

(c) once any of one of interrupts b, e, f, g, h, or i are set the instruction counter is reset so that this instruction can be re-executed and the rest of the control cycle is by-passed.

ADDRESS TRANSLATION

Each time the control cycle must translate an address from the virtual name space to the physical name space it calls the procedure <u>vmtranslate</u> (4-10) with the virtual page VP, the page word WORD, the read/write indicator RWM, the word to be filled stored SBR, and the task identifier TASK as arguments. The procedure loads variable TMR from the VP-th location of the translation memory.

If this page is active (act(TMR)=1), then the page is in memory and attached to the task's working set. In this case the protection key and write protect bit are checked to see if this is a proper fetch/store. If so, the read (RWM=1) or write (RWM=0) takes place and the "reference bit" and, if a write was performed, the "changed bit" for the page are turned on. If an improper fetch/store was attempted then the access flag AFLAG and/or the write protect flag WFLAG are set to 1, and the address fault interrupt is posted.

If the page was not active, then the procedure <u>page fault</u> (4-11) is called. This procedure has as arguments the virtual page VP, physical page PAGE, task descriptor address TASK, and copy of the translation memory location (address VP) TMR. The first thing the procedure does is to assume that the page is not in memory: the working set size for the task (wssize) is incremented and the value of the interval timer is stored. Then it checks the page table (procedure <u>checkpagetable</u> (4-12)) to see if the page indicated by the translation memory address is in memory. If it finds it (SUCCESS=1) then it replaces the task on the task's working set list (if it is in the user's virtual memory) or the system task's working set list (if it is in the system's virtual memory). In addition, it decrements the non-assigned page count AVPC, the unattached page count CAVPV and, if the page was on the list of available pages, the available page count CAVPA.

If the page was not found to be in memory then a page must be brought into memory (procedure queue for input (4-13)). However, if no pages are unattached (CAVPV=0), then the task is deactivated to the ready list unless it is the only task other than the system task. In the latter case, the task selects a suitable page for overlaying from its own working set (procedure cannabolize (Table 4-6)) and queues the page for input.

This completes the description of procedure pagefault. It should be noted that the three page counts are essential to the proper operation of the algorithm. Counter AVPC represents the number of pages that have not been assigned to active tasks. It is used solely to determine when other tasks may be added to the running list. Counter CAVPA represents the number of pages that are currently available for immediate overlay (i.e., the length of the list of available pages). Counter CAVPV represents the number of pages in memory not attached to working sets, either on the list of available pages (LAVP) or on the list of swappable pages (LSP) or in the process of being swapped out. This counter is used to determine if there are available pages to assign. Especially notable is the fact that counter AVPC is signed (i.e., it can go negative). This is due to the fact that during the first τ seconds of a task's active period the working set size is being re-evaluated. However, its old working set size is reserved from the AVPC until it has been re-evaluated. Therefore, the condition could arise when all pages are reserved according to AVPC but not currently assigned according to LAVP. A task that wished to increase its working set size at this point would be allowed to but would cause AVPC to go negative

Procedure pagefault called two procedures that have not previously been described:

(a) procedure checkpagetable (4-12). This procedure checks to see if the physical page that appeared in the addressed translation memory location is still assigned to thatpage, although it is, not in its working set (i.e., has not been reassigned). In addition it makes certain that the page is not being swapped out (puse(PTR)=2) and if not loads variable PTLIST with list pointer of the list on which it resides (either the LAVP or the LSP).

(b) procedure queue for input (4-13). If the LAVP is not empty (CAVPA≠0), the first entry of the LAVP is queued for input (queue request) and the suspension interrupt (susp(INTERRUPT)) is posted. If it is empty, then the deactivate-to-the-page-wait-list interrupt is set (deactpw). In either case, if the LSP is not empty the first entry is queued for swap-out. This last action assures that a new page will be added to the LAVP.

Procedure queue for input called procedure remove trace (4-14) before queueing a page for input. This procedure removed any pointers to the physical page that may have previously existed in another task's translation memory. In addition it called procedure load ptr (4-15) to load the page table with necessary information about the page's new identity (protection key, write protect bit, resident bit, task identity, virtual page, and drum address). It should be noted that the task identity is either the user task (if $P \leq 511$) or the system task (if $P > 511$). All active pages from the system virtual memory are attached to the system task, allowing these pages to be shared by all user tasks.

INTERVAL TIMER INTERRUPT

The interval timer is an automatic counter that is decremented by a clock cyle. It is only in operation during the control cycle. When it reaches zero, if no other interrupts have been posted that effect the current task, then the interval timer interrupt is posted. The interval timer interrupt is processed by procedure interval timer interrupt (4-16).

The procedure updates the active time count for the task by the burst amount β. It resets the interval time for the task to β. If the active time for the task is greater than or equal to the assigned quantum, the task is deactivated to the ready list by procedure deactivate (4-17), else the next task on the running list is assigned the processor by procedure circulate (4-19). If this marked the end of the task's first τ (the working set parameter) seconds, then the available page count AVPC is adjusted to reflect the true working set size (wssize) rather than the assigned working set size (wsold).

(4-17) removes a task from the running list to the specified task list. If the task is non-resident, it zeroes the active time, stores the current working set size as wsold, releases the tasks pages to the LAVP or LSP (procedure release page (4-18)), unloads the task's state vector, and initiates the next task. If the task is resident, it merely unloads the state vector and initiates the next task. The next state is initiated by loading its state vector and the interval timer with the new task's current timer value.

Procedure circulate (4-19) is called in order to rotate the circular running list and initiates processing of the next task on the list. This is accomplished by unloading the last task's state vector, rotating the list, loading the new task's state vector and interval timer.

BLOCK AND UNBLOCKED INTERRUPT

A block interrupt is handled by the block procedure (4-20). It consists of setting the task's interval time value to zero and deactivating the task to the block list (procedure deactivate (4-17)).

An unblock interrupt is handled by the blocked interrupt procedure (4-21). Posting of the unblock interrupt only interrupts the processing task, it does not remove it from processing. Therefore, the procedure must first store the task's register set and, at the end of the procedure, restore the register set. In between it releases the locked-in pages of task just unblocked and places the task on the ready list.

A task that is blocked (procedure block) awaiting completion of another task, must have already locked-in the pages that will be required by the task (e.g., for a non-paged I/O, the block to be stored or loaded must be in memory). Pages are locked-in by the procedure lockin (4-22) which removes the pages from the task's active list and places them on the task's locked-in list.

PAGING DRUM INTERRUPT

The paging drum interrupt also interrupts a processing task temporarily. Therefore, the procedure must store and restore the register set. The procedure paging drum interrupt (4-23) handles the interrupt. If the page for which the interrupt was issued was swapped-in (PTRAN=1), then there is a task awaiting the arrival of the page. This task is placed back on the running list, and the appropriate translation memory location is loaded. If the page was swapped-out, then it is added to the list of available pages. At this point, if the page wait list is not empty, then there was a task waiting for that physical page. In this case, the virtual page the task desires is restored and queued for input (procedure queue for input (4-13)).

Procedure set tm entry loads the translation memory location that will reference the page just read in. The task's translation memory copy is changed to reflect the new status of the page. The procedure sets the active bit and stores the physical page address in the entry.

COMPLETED TASK INTERRUPT

Procedure completed task (4-25) handles the completed task interrupt by deactivating the task to the completed task list (procedure deactivate (4-17)).

DEACTIVATE TO PAGEWAIT AND READY LIST INTERRUPTS

These interrupts are posted when a page fault occurs. If the task cannot immediately be assigned a page for swap-in, it is deactivated to the page wait list to await a page. If the task has exceeded its working set size and no pages are available (AVPC=0) then the task is deactivated to the ready list. Both interrupts are handled by procedure deactivate (4-17).

SUSPEND TASK INTERRUPT

The suspend task interrupt is posted when a task must await the swapping-in of a page. The procedure suspend (4-26) handles the interrupt by removing the task from the running list, storing its state vector, and initiating the next task on the running list. As the page table entry corresponding to the page being swapped in contains the task descriptor address, the task descriptor does not have to be attached to any list.

SYSTEM TASK

Procedure system task (4-27) performs the major portion of the memory management. It scans the page table for non-resident pages that have not been referenced in the last τ second (util(PTR)$\geq\tau$). If it finds any, it detaches the page from the tasklist it is on, reduces the working set size of the task that it was attached to, and releases the page (procedure release page (4-18)). The procedure then searches the ready list for tasks whose working set will fit. If it finds any, it removes them from the ready list, finds the first page necessary for their processing, and queues this page for input.

NON-PAGED INPUT/OUTPUT

The problem of non-paged input/output will only be briefly be considered here. The problem manifests itself when an I/O command is given that involves the transfer of a block of words, the addresses of which include one or more page boundaries. As the pages would not be expected to be physically contiguous, the I/O channel must either use virtual addresses (requiring the task's translation table to access the pages) or it must avoid transfers of blocks of information that cross page boundaries. This latter method is adopted in many paging systems (e.g., RCA Spectra 70/46 and 70/61). However, the former method is more elegant in that it follows the virtual memory philosophy.

Without going into too much detail, this may be effected with an I/O channel that itself had a translation memory and the capability of demand paging. Only the first page would be locked-in core and as others would be required they could be paged on demand. The channel would suspend I/O for that task and begin I/O for another while the page was being swapped-in. However, if this method was used then the memory management subsystem would have to be revised to satisfy the added complications (i.e., tasks blocked for I/O would have active working sets).

FUNCTIONAL RELATIONSHIP

Figure 4-3 shows the functional relationships between the procedures. The major procedures appear on the left while the procedures that perform simple functions appear on the right. It should be noted that the references

```
control cycle . . . vmtranslate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . loadtmr
                                          .                                                                             loadmm
                                          .                                                                             addressfault(INTERRUPT)+1
                                          .                                                                             storemm
                                          .                                                                             storetmr
                                          . pagefault. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . store interval timer
                                                       .                                                                detachpt
                                                       .                                                                putpt
                                                       .                                                                storetmr
                                                       .                                                                deactrl(INTERRUPT)+1
                                                       . check page table . . . . . . . . . . . . . . . . . . . . . . . load page descriptor
                                                       . cannabalize  . .  (not defined)
                                                       . queue for input  . . . . . . . . . . . . . . . . . . . . . . . getpt
                                                                           .                                            loadptr
                                                                           .                                            queue request
                                                                           .                                            store page descriptor
                                                                           .                                            deactpw(INTERRUPT)+1
                                                                           .                                            susp(INTERRUPT)+1
                                                                           .remove trace . . . . . . . . . . . . . . . loadtmr
                                                                                                                        storetmr
interrupt testing . interval timer interrupt.deactivate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . detachpt
                  .                         .            .                 .                                            load interval timer
                  .                         .            .                 . . . . . . . . . . . get task
                  .                         .            .                 .                      put task
                  .                         .            .                 .                      get next task
                  .                         .            .                 .unload state vector. .store registers
                  .                         .            .                 .                      storetm
                  .                         .            .                 .load state vector. . .load registers
                  .                         .            .                                        loadtm
                  .                         .            .release page . . . . . . . . . . . . . . . . . . . . . . . . putpt
                  .                         .circulate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . load interval timer
                  .                                                        . . . . . . . . . . . get next task
                  .                                                        .unload state vector*
                  .                                                        .load state vector*
                  . block . . . . . . . . . .deactivate*
                  . blocked interrupt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . detachpt
                  .                                      . . . . . . . . . . . . . . . . . . . . store registers
                  .                                      .                                        detach task
                  .                                      .                                        put task
                  .                                      .                                        restore registers
                  .                                      .release page*
                  . paging drum interrupt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . load page descriptor
                  .                                      .                                                              putpt
                  .                                      . . . . . . . . . . . . . . . . . . . . .store registers
                  .                                      .                                        put task
                  .                                      .                                        get task
                  .                                      .                                        restore registers
                  .                                      . . . . . . . . .  set tm entry
                  .                                      .queue for input*
                  . completed task . . . . . deactivate*
                  . deactivate*
                  . suspend. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . load interval timer
                  .                                       . . . . . . . . . . . . . . . . . . . . get task
                  .                                                                             . unload state vector
                  .                                                                             . load state vector
                  . errorstop  . . . . . . . (not defined)

                                                                                                                   * defined above
```

Figure 4-3 Functional relationship between procedures

form a tree structure, with the only circularity at the top level (the left-hand side).

This completes the presentation of the virtual memory system.

```
procedure  gettask (T,LIST);
        integer  T,LIST;
    begin  comment  remove the first task descriptor on list LIST and place the
                    address in T;
        integer  TSK;
        T←f(LIST);
        number(LIST)←number(LIST)-1;
        if  number(LIST)=0  then  f(LIST)=0
                              else
                begin         f(LIST)←nexttask(T);
                              TSK←f(LIST);
                              previous task(TSK)←previous task(T);
                              TSK←previous task(T);
                              nexttask(TSK)←nexttask(T)
                end
    end  gettask
```

Procedure 4-1. gettask

```
procedure  puttask (T,LIST);
           integer  T,LIST;
     begin comment  place task T in list LIST;
           integer TSK
           if  f(LIST)=0  then
                   begin  f(LIST)←T;
                          nexttask(T)←T;
                          previous task(T)←T;
                   end
                          else
                   begin  TSK←previous task(f(LIST));
                          previous task(f(LIST))←T;
                          previous task(T)←TSK;
                          nexttask(T)←f(LIST);
                          nexttask(TSK)←T;
                   end;
           number(LIST)←number(LIST)+1
     end  puttask
```

Procedure 4-2. puttask

```
procedure  getnexttask(T,LIST);
           integer  T,LIST;
    begin  comment  the circular list LIST is circulated one position (the
                    head becomes the tail) and the address of the new task
                    descriptor is placed in T;
           T←nexttask(f(LIST));
           f(LIST)←T
    end
```

Procedure 4-3. getnexttask

```
procedure  detachtask(T,LIST);
           integer  T,LIST;
     begin comment  task descriptor T is removed from list LIST;
           integer  TSK;
           TSK←previoustask(T);
           nexttask(TSK)←nexttask(T);
           TSK←nexttask(T);
           previoustask(TSK)←previoustask(T);
           number(LIST)←number(LIST)-1;
           if  f(LIST)=T then
                 begin if  number(LIST)=0 then  f(LIST)←0
                                          else  f(LIST)←TSK
                 end
     end  detachtask
```

Procedure 4-4. detachtask

```
procedure  load state vector(TASK)
           integer  TASK;
    begin  comment  load all registers and translation memory for task TASK;
           restore registers(TASK)
           loadtm(TASK)
    end    load state vector
```

Procedure 4-5. load state vector

```
procedure  unload state vector(TASK);
           integer  TASK
    begin  comment  unload all registers and those altered translation
                    memory locations for task TASK;
           store registers(TASK);
           storetm(TASK)
    end    unload state vector
```

Procedure 4-6. unload state vector

```
procedure  store tm(TASK)
        integer  TASK; global  TMR,PTR;
   begin  comment  unload those translation memory locations that have been
                   referenced to task TASK's storage area;
        integer  I;
        for  I=0 step 1  until  tmcount(TASK)-1 do
            begin  loadtmr(I,TMR);
                 if  (ref(TMR)=1) then
                     begin  loadpagedescriptor(blk(TMR),PTR);
                            util(PTR)←0;
                            if  chgd(TMR)=1  then  chge(PTR)←1;
                            store page descriptor (blk(TMR),PTR);
                            sact(I,TASK)←act(TMR);
                            sblk(I,TASK)←blk(TMR)
                     end
            end
   end  store tm
```

Procedure 4-7 store tm

```
procedure  loadtm(TASK);
           integer  TASK;  global MTKEY,TMR;
     begin  comment  load task TASK's copy of the translation memory;
           integer  I;
           for  I=0 step 1  until  tmcount(TASK)-1 do
                 begin  act(TMR)←sact(I,TASK);
                        ref(TMR)←0;
                        chgd(TMR)←0;
                        wp(TMR)←swp(I,TASK);
                        wkey(TMR)←swkey(I,TASK);
                        res(TMR)←sres(I,TASK);
                        blk(TMR)←sblk(I,TASK);
                        storetmr(I,TMR)
                 end
           for  I=tmcount(TASK) step 1 until  511 do
                 begin  wkey(TMR)←MTKEY;
                        act(TMR)←1;
                        storetmr(I,TMR)
                 end
     end  loadtm
```

Procedure 4-8. loadtm

```
begin comment major CPU cycle;
      integer TASK,READYLIST,RUNLIST,PAGEWAIT,BLOCKLIST,COMPLETELIST,INTERRUPT,
              PAGINT,PTRAN,TASKINT,PTR,TMR,LSP,LAVP,CAVPV,CAVPA,AVPC,AFLAG,WFLAG,
              SYST,β,τ,MTKEY;
 MAIN: controlcycle(TASK); if INTERRUPT=0 then go to MAIN;
      suspend interval timer;
      if timer(INTERRUPT) then
         begin timer(INTERRUPT)←0; interval timer interrupt(TASK) end;
      if blocked(INTERRUPT) then
         begin blocked(INTERRUPT)←0; block(TASK) end;
      if unblocked(INTERRUPT) then
         begin unblocked(INTERRUPT)←0; blocked interrupt(TASK,TASKINT) end;
      if page(INTERRUPT) then
         begin page(INTERRUPT)←0; paging drum interrupt(PTRAN,PAGINT,TASK)
         end;
      if complete(INTERRUPT) then
         begin complete(INTERRUPT)←0; completed task(TASK) end;
      if deactpw(INTERRUPT) then
         begin deactpw(INTERRUPT)←0; deactivate(TASK,PAGEWAIT)end;
      if deact rl (INTERRUPT) then
         begin deactrl(INTERRUPT)←0; deactivate(TASK,READYLIST) end;
      if susp(INTERRUPT) then
         begin susp(INTERRUPT)←0; suspend(TASK)end;
      if addressfault (INTERRUPT) then
         begin errorstop(TASK) end;
      if drumpage(INTERRUPT) then
         begin errorstop(TASK) end;
      restore interval timer;
      go to MAIN
end
```

Procedure 4-9 Major CPU Cycle

```
procedure vmtranslate (VP,WORD,RWM,SBR,TASK);
          integer VP,WORD,SBR,TASK; boolean RWM; global AFLAG,WFLAG;
    begin comment virtual memory addressing sequence;
          integer TMR;
          loadtmr(VP,TMR);
          if (act(TMR)=1) then
           begin if (wkey(TMR)=key(TASK))∧(¬wp(TMR)∨RWM)
                         then
                 begin ref(TMR)←1;
                         if RWM then loadmm(blk(TMR),WORD,SBR)
                                 else
                         begin storemm(blk(TMR),WORD,SBR);
                               chgd(TMR)←1
                         end;
                         storetmr(VP,TMR)
                 end
                         else
                 begin AFLAG←(wkey(TMR)≠key(TASK));
                         WFLAG←wp(TMR)∧¬RWM;
                         addressfault(INTERRUPT)←1;
                 end
           end
                 else pagefault(VP,blk(TMR),TASK,TMR)
    end vmtranslate
```

Procedure 4-10 vmtranslate

```
procedure  pagefault(P,PAGE,TASK,TMR);
        integer  P,PAGE,TASK,TMR; global  PTR,LAVP,SYST,CAVPA,CAVPV,READYLIST,RUNLIST;
    begin  comment  This procedure is called when virtual page P is not found in the work-
        ing set of task TASK.  In this case, the procedure checks to see if the page
        PAGE resides in main memory from an earlier request and, if so, attaches it
        to the working set and sets FLAG to 1.  Otherwise, a page request is made
        and the task is made to wait while the page is swapped in;
        integer  SUCCESS,PTLIST;
        wssize(TASK)←wssize(TASK)+1; store interval timer(interval value(TASK));
        if  PAGE≠0  then  checkpagetable(PAGE,P,TASK,PTLIST,SUCCESS);
        if  SUCCESS=1 then
               begin  SUCCESS←0; detachpt(PAGE,PTLIST,PTR); AVPC←AVPC-1;
                      if  PTLIST=LAVP  then  CAVPA←CAVPA-1; CAVPV←CAVPV-1;
                      if  P>511 then  putpt(PAGE,tasklist(SYST),PTR);
                                  else  putpt(PAGE,tasklist(TASK),PTR);
                      act(TMR)←1;storetmr(P,TMR)
               end
                      else
               begin if  CAVPV=0 then
                        begin if  number(RUNLIST)≤2 then
                                    begin  cannabolize (PAGE,tasklist(TASK));
                                              wssize(TASK)←wssize(TASK)-1
                                    end
                                                      else  deactrl(INTERRUPT)←1
                        end
                     if  INTERRUPT≠0 then
                        begin  queue for input(TASK,P);
                                 CAVPV←CAVPV-1; AVPC←AVPC-1
                        end
               end
    end  pagefault
```

Procedure 4-11 pagefault

```
procedure  checkpagetable (B,VP,T,PTLIST,S);
           integer  B,VP,T,PTLIST,S;  global PTR;
    begin  comment  if page table location B corresponds to virtual page VP
                    of task T and it is not being swapped out, then the list
                    head of the list it is on is stored in PTLIST, and S is
                    set to 1, else S is set to 0;
           loadpagedescriptor(B,PTR);
           if  (VP<511)∧(puse(PTR)≠2) then
               begin  if  (VP=vp(PTR))∧(T=tid(PTR)) then  S←1
                                                     else  S←0
               end
                      else  if  (VP=vp(PTR) then  S←1
                                            else  S←0;
           if  S=1 then
               begin  if  puse(PTR)=1 then PTLIST←LSP
                                      else PTLIST←LAVP
               end
    end  checkpagetable
```

Procedure 4-12 checkpagetable

```
procedure  queue for input(TASK,P);
         integer  P,TASK; global CAVPA,LAVP,PTR,LSP,SYST;
    begin  comment  this procedure queues page P of task TASK for input from the
                    drum and writes a page to drum if one is available;
         if  p>511 then  TSK←SYST else  TSK←TASK;
         if  CAVPA≠0 then
             begin  getpt(PAGE,LAVP,PTR);  CAVPA←CAVPA-1;
                    removetrace(vp(PTR),tid(PTR,TASK));
                    load ptr(0,0,0,swkey(P,TASK),swp(P,TASK),0,sres(P,TASK),
                                      0,TSK,P,sdp(P,TASK),0);
                    store page descriptor(PAGE,PTR);
                    queue request(PAGE);
                    susp(INTERRUPT)←1
             end
                    else
             begin  deactpw(INTERRUPT)←1;
                    desired virtual page(TASK)←P
             end;
        if  fp(LSP)≠0then
             begin  getpt(PAGE,LSP,PTR); row(PTR)←1;
                    store page descriptor (PAGE,PTR); queue request(PAGE)
             end
    end  queue for input
```

Procedure 4-13 queue for input

```
procedure  remove trace(P,TSK,TASK);
           integer P,TSK,TASK; global TMR;
    begin  comment  remove any pointers to physical page P from task TSK
                    so that task TASK may make use of it;
           if  TSK=TASK then
                    begin  loadtmr(P,TMR);
                           blk(TMR)←0;
                           storetmr(P,TMR)
                    end;
           sblk(P,TSK)←0
    end
```

Procedure 4-14 remove trace

```
procedure  load ptr(PUSE,LB,LF,WKEY,WP,CHGE,RES,     UTIL,TID,VP,DP,ROW);
           integer PUSE,LB,LF,WKEY,WP,CHGE,RES,     UTIL,TID,VP,DP,ROW; global  PTR;
   begin   comment this procedure loads the page table buffer register PTR;
           puse(PTR)←PUSE;
           lb(PTR)←LB;
           lf(PTR)←LF;
           wkey(PTR)←WKEY;
           wp(PTR)←WP;
           chge(PTR)←CHGE;
           res(PTR)←RES;
           util(PTR)←UTIL;
           tid(PTR)←TID;
           vp(PTR)←VP;
           dp(PTR)←DP;
           row(PTR)←ROW
   end  loadptr
```

Procedure 4-15 load ptr

```
procedure  interval timer interrupt(TASK);
        integer  TASK;  global  $\beta$,READYLIST,$\tau$,AVPC;
        begin  comment  procedure to handle interval timer interrupts:  test
                        for quantum overflow and access next task on the run-
                        ning list;
            integer  TACTIVE;
            TACTIVE$\leftarrow$activetime(TASK)+$\beta$;  interval value(TASK)$\leftarrow\beta$;
            activetime(TASK)$\leftarrow$TACTIVE;
            if  TACTIVE$\geq$quantum(TASK)  then  deactivate(TASK,READYLIST)
                                        else
                 begin  if  (TACTIVE $<\tau+\beta$)$\wedge$(TACTIVE$\geq\tau$)  then
                                      AVPC$\leftarrow$AVPC+wsold(TASK)-wssize(TASK);
                        circulate(TASK)
                 end
        end  interval timer interrupt
```

Procedure 4-16 interval timer interrupt

```
procedure  deactivate(TASK,LIST);
        integer  TASK,LIST; global  RUNLIST;
    begin  comment  this procedure removes task TASK from list RUNLIST and places it
                    on list LIST;
        integer  TASKLIST,NEXTPAGE,I,PTR,TACTIVE;
        TACTIVE←activetime(TASK); activetime(TASK)←0;
        TASKLIST←tasklist(TASK);
        if  resident(TASK)=0  then
            begin  if  (TACTIVE≥τ)∧(wsold(TASK)<wssize(TASK))then wsold(TASK)←wssize
                                                                                (TASK);
                   wssize(TASK)←0; NEXTPAGE←fp(TASKLIST);
                   for  I←NEXTPAGE  until  NEXTPAGE=0  do
                        begin  detach pt(I,TASKLIST,PTR);
                               release page(I,PTR);
                               NEXTPAGE←lf(PTR)  end;
                   gettask(TASK,RUNLIST);
                   puttask(TASK,LIST);
                   unload state vector(TASK);
                   TASK←f(RUNLIST)
            end
                         else
            begin  unload state vector(TASK);
                   get next task(TASK,RUNLIST)
            end;
        tasklist(TASK)←TASKLIST;
        load state vector(TASK); load interval timer(interval value(TASK))
    end  deactivate
```

Procedure 4-17 deactivate

```
procedure  release page(I,PTR);
           integer  I,PTR; global  LSP,LAVP,CAVPA,CAVPV,AVPC;
    begin  comment  this procedure releases page I (the contents of which are in PTR)
                    to either the list of available pages     or the list of
                    swappable pages;
           if  chge(PTR)=1  then  putpt(i,LSP)
                                  else
                    begin  putpt(i,LAVP);
                           CAVPA←CAVPA+1
                    end;
           CAVPV←CAVPV+1;
           AVPC←AVPC+1
    end  release page
```

Procedure 4-18 release page

```
procedure  circulate(TASK);
           integer  TASK; global  RUNLIST;
    begin  comment  initialize processing of the next task in the running
                    list;
           unload state vector(TASK);
           get next task(TASK,RUNLIST);
           load state vector(TASK);
           load interval timer(interval value(TASK))
     end  circulate
```

Procedure 4-19 circulate

```
procedure  block  (TASK); comment  task must lock pages in memory before blocking;
           integer  TASK; global  BLOCKLIST;
     begin  comment  this procedure places task TASK on the blocked list;
           interval value(TASK)←0;
           deactivate(TASK,BLOCKLIST)
       end  block.
```

Procedure 4-20 blocked

```
procedure  blocked interrupt(TASK,TASKINT);
           integer  TASK;
     begin  comment  this procedure interrupts task TASK to release task TASKINT from
                     the blocked list to the ready list;
           integer  PTLIST,I,STORE;
           store registers(TASK); STORE←TASK; TASK←TASKINT,
           PTLIST←locked in list(TASK);
           NEXTPAGE←fp(PTLIST);
           for  I←NEXTPAGE until  NEXTPAGE=0 do
                  begin  detachpt(I,PTLIST,PTR);
                         releasepage(I,PTR);
                         NEXTPAGE←lf(PTR)
                  end
           detachtask(TASK,BLOCKLIST);
           puttask(TASK,READYLIST); TASK←STORE;
           restore registers(TASK)
     end  blocked interrupt
```

Procedure 4-21 blocked interrupt

```
procedure lockin(PAGE,TASK);
          integer PAGE,TASK; global TASKLIST;
    begin comment add page descriptor PAGE to the list of locked-in
                  pages for task TASK;
          TASKLIST←tasklist(TASK);
          detachpt(PAGE,TASKLIST,PTR);
          tasklist(TASK)←TASKLIST;
          TASKLIST←lockedin list(TASK);
          putpt(PAGE,TASKLIST,PTR);
          lockedin list(TASK)←TASKLIST
    end
```

Procedure 4-22 lockin

```
procedure  paging drum interrupt(PTRAN,PAGE,TASK);
           integer  PTRAN,PAGE,TASK;  global  PAGEWAIT,RUNLIST,LAVP,CAVPA;
    begin  comment  when the paging drum signals completion of a sector revolution it
           sets PTRAN, indicates the page transferred as PAGE, and interrupts task TASK.
           This procedure then issues the next page request and reactivates the waiting
           task;
           integer  PTR,TSK,VP;
           store registers(TASK);
           if  PTRAN=1  then
                 begin  load page descriptor(PAGE,PTR); settmentry(PAGE,PTR);
                        putpt(PAGE,tasklist(tid(PTR)),PTR);
                        puttask(tid(PTR),RUNLIST)
                 end;
           if  PTRAN=2  then
                 begin  putpt(PAGE,LAVP,PTR);
                        CAVPA←CAVPA+1;
                        if  f(PAGEWAIT)≠0  then
                            begin  gettask(TSK,PAGEWAIT);
                            VP←desired virtual page(TSK);
                            queue for input(TSK,VP)
                        end
                 end;
           restore registers(TASK)
    end  paging drum interrupt
```

Procedure 4-23 paging drum interrupt

```
procedure  set tm entry(PAGE,PTR);
           integer  PAGE,PTR;
   begin   comment  load the translation memory location that will address
                    the page PAGE;
           integer  VP,TSK;
           VP←vp(PTR);
           TSK←tid(PTR);
           sact(VP,TSK)←1;
           sblk(VP,TSK)←PAGE
   end  set tm entry
```

Procedure 4-24 set tm entry

```
procedure  completed task(TASK);
           integer  TASK; global  COMPLETELIST;
    begin  comment  this procedure places task TASK in the completed task list;
           deactivate(TASK,COMPLETELIST)
    end    completed task
```

Procedure 4-25 completed task

```
procedure  suspend(TASK);
           integer  TASK; global  RUNLIST;
    begin  comment  suspend task TASK by removing it from the running list
                    and initiate the next task on the running list;
           gettask(TASK,RUNLIST);
           unloadstate vector(TASK);
           TASK←f(RUNLIST);
           load state vector(TASK);
           load interval timer(interval value(TASK))
    end  suspend
```

Procedure 4-26 suspend

```
begin comment a portion of the resident system program (always in the RUNLIST);
      integer I,PTR,INTVAL,WSIZE,FAILURE,VP,TSK;
      global TASK,τ,AVPC,READYLIST;
      for I←0 step 1 until 63 do
          begin load page descriptor(I,PTR);
                if (use(PTR)=0)∧(blkd(tid(PTR))≠0) then
                   begin util(PTR)←util(PTR)+interval value(TASK);
                         if (util(PTR)≥τ)∧(res(PTR)≠1) then
                            begin wssize(tid(PTR))←wssize(tid(PTR))-1;
                                  detachpt(I,tasklist(tid(PTR)),PTR);
                                  releasepage(I,PTR)
                            end;
                            store page descriptor(I,PTR)
                   end
          end
LOOP: WSIZE←0; FAILURE←0;
      for AVPC←AVPC-WSIZE while FAILURE=0 do
          begin search ready list for task(TSK) whose working set size(WSIZE)
                is less than(AVPC) and note(FAILURE);
                if FAILURE=0 then
                   begin VP←first request(TSK); detachtask(TSK,READYLIST);
                         queue for input(TSK,VP)
                   end
          end
                if number(RUNLIST)=1 then go to LOOP
end systemtask
```

Procedure 4-27 system task

5. DISCUSSION

The aim of this paper is to present a detailed algorithm of a memory management system, the major components of which could be microprogrammed. This goal seems reasonable in that the bulk of the procedures described handle addressing and interrupts. Only the system task would necessarily be in software and that could be one special instruction that would initiate the appropriate micro subroutine.

However, the author would suggest that the algorithm first be simulated in its present form to check for logical flaws and to attempt to discover the performance of the memory management subsystem under random conditions. This simulation may demonstrate a need to modify the algorithm.

Once the algorithm has been simulated in this manner, microprogramming by CDL and simulation by the CDL simulator will allow the system overhead to be calculated. At this point evaluation of the system could be undertaken.

6. REFERENCES

1. Abate, Joseph and Harvey Dubner, "Optimizing the Performance of a Drum-Like Storage," IEEETC, November 1969, pp. 992-997.

2. Belady, L.A., "A Study of Replacement Algorithms for a Virtual Storage Computer," IBM Systems Journal, Vol. 5, No. 2 (1966), pp. 78-101.

3. Brotherton, D. and S. Domchick, Preliminary Programming Manual for RADC 2048-Word Associative Memory, Goodyear Aerospace Corporation (Akron, Ohio: 1966).

4. Chu, Yaohan, "Direct Execution of Programs in Floating Code by Address Interpretation," IEEETEC, June 1965, pp. 417-444.

5. Chu, Yaohan, "An ALGOL-like Computer Design Language," CACM, February 1966, pp. 72-76.

6. Chu, Yaohan, Introduction to Computer Organization, Prentice-Hall, 1970.

7. Chu, Yaohan, O.R. Pardo, and Yeh, Jeffrey, "A Methodology for Unified Hardware-Software Design," Technical Report 70-107, Computer Science Center, University of Maryland.

8. Coffman, E.G., Jr., "Analysis of a Drum Input/Output Queue Under Scheduled Operation in a Paged Computer System," Journal ACM, January 1969, pp. 73-90.

9. Coffman, E.G., and L. C. Varian, "Further Experimental Data on the Behavior of Programs in a Paging Environment," CACM, July 1968, pp. 471-474.

10. Corbato, F.J. and Saltzer, J.H., "Some Considerations of Supervisor Program Design for Multiplexed Computer Systems," Proceedings of IFIPS, pp. 66-71.

11. Daley, Robert C. and Jack B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS," CACM, May 1968, pp. 306-312.

12. Denning, Peter J., "The Working Set Model for Program Behavior," CACM, May 1968, pp. 323-333.

13. Denning, Peter J., "Virtual Memory", Princeton University, Computer Science Laboratory, Tech. Report 81 (January 1970).

14. Denning, "Effects on Scheduling in File Memory Operations," Proceeding ACM 22nd National Meeting, 1967, pp. 9-21.

15. Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control," CACM, September 1965, pp. 569.

16. Dijkstra, Edsger W., "The Structure of the 'THE' - Multiprogramming System," CACM, May 1968, pp. 341-346.

17. Falkoff, A.D., "Algorithms for Parallel-Search Memories," JACM, Vol. 9, No. 4 (October, 1962) pp. 488-511.

18. Gibson, Charles T., "Time Sharing in the IBM System/360 Model 67," Proceedings ACM 21st National Meeting, 1966, pp. 61-78.

19. Hellerman, H. and Smith, H.J., Jr., "Throughput Analysis of Some Idealized Input, Output, and Compute Overlap Configurations," Computing Surveys, June 1970, pp. 111-118.

20. IBM System/360 Model 67 Functional Characteristics, IBM Systems Reference Library, File No. S360-01 (Form GA27-2719-1).

21. Iliffe, J.K. and Jodeit, Jane G., "A Dynamic Storage Allocation Scheme," Computer Journal, October 1962, pp. 200-209.

22. Jacobson, David, "A Self Organizing Drum," IEEETEC, June 1964, pp. 302.

23. Jones, Robert M., "Factors Affecting the Efficiency of a Virtual Memory," IEEETC, November 1969, pp. 1004-1008.

24. Oppenheimer, G. and Weizer, N., "Resource Management for a Medium Scale Time-Sharing Operating System," CACM, May 1968, pp. 313-322.

25. Ramamoorthy, C.V. and K.M. Chandy, "Optimization of Memory Hierarchies in Multi programmed Systems," Journal of ACM, July 1970, pp. 426-445.

26. Randell, B. and Kuehner, C.J., "Dynamic Storage Allocation Systems," CACM, May 1968, pp. 297-306.

27. Randell, B., "A Note on Storage Fragmentation and Program Segmentation," CACM, July 1969, pp. 365-369.

28. Smith, John L. "Multiprogramming under a Page in Demand Strategy," CACM, October 1967, pp. 636-646.

29. Spectra 70: 70/61 Processor Reference Manual, RCA Information Systems (Camden, New Jersey: October 1969).

30. Wallace, V.L. and D.L. Mason, "Degree of Multiprogramming in Page-on-Demand Systems," CACM, June 1969, pp. 305-308.

31. Weingarten, Allen, "The Eschenbach Drum Scheme," CACM, July 1966, pp. 509-512.

32. Weizer, Norman and Oppenheimer, G., "Virtual Memory Mangement in a Paging Environment, Proceedings ACM 24nd National Meeting, 1969, pp. 249-256.

33. Wirth, Niklaus, "On Multiprogramming, Machine Coding, and Computer Organization," CACM, September 1969, pp. 489-498.