AN OPERATING SYSTEM AND A MICROPROGRAMMABLE MACHINE

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Louis Joseph Vogel

May 1970

ACKNOWLEDGMENT

The author would like to thank Mr. J. B. Wyatt who found time despite a very busy schedule to always be available to give advice and consultation to the author during the development of this material. He is thanked also for being the source of the notation scheme used in Chapter 1 to describe the system algorithms.

Thanks are extended to my mother for typing drafts. Special thanks are due my mother-in-law who spent much time, at night and on weekends, typing the final draft.

Finally, extra special thanks are extended to my wife, Shelia, who, in addition to typing draft after draft, proofreading, and advising, put up with the author through all the strain during the preparation of the thesis.

L.J.V.

Houston, Texas

May, 1970

AN OPERATING SYSTEM AND A MICROPROGRAMMABLE MACHINE

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Louis Joseph Vogel

May 1970

ABSTRACT

Multiprogrammed interactive operating systems are usually implemented with software. The growth of interest in microprogrammable central processors creates a new dimension in effective computer utilization. While the ability of microprogrammable machines to emulate other machines is well known, the use of microprogramming to aid in the implementation of operating systems is a new area of interest. To investigate this area, a specific operating system and a specific microprogrammable machine were needed. The availability of an INTERDATA Model 4 dictated the choice of machines, but there was not an operating system available for this machine.

A model for a multiprogrammed interactive operating system is developed and described. The system is intended for use on a hardware configuration of the general size and capability of the INTERDATA Model 4, but no special consideration is given to its microprogramming characteristics during the development of the system. Selected microprograms are developed to illustrate some implementation features of an emulated machine which has the properties desired by the operating system model.

The effect of microprogramming is considered with respect to speed of execution of the resulting system, utilization of primary storage, needs for control storage, and desirability of the use of microprogramming by the independent researcher or systems designer.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

AN OPERATING SYSTEM MODEL

Section 1 - Introduction

More and more today the use of computers is becoming commonplace in our society. Coupled with this growth in usage, is a growth in computer technology. Computer manufacturers are continually making advances in the equipment that can be made available to a company or individual that desires to use a computer.

The use of microprogramming [1], has been adopted by many manufacturers to facilitate implementing these technological advances. Sometimes the ability to modify the microprograms of the central processor is given to the user. This attitude reflects a desire to promote the adaptability of a given machine; thus, allowing the user to tailor the system to his particular needs. The opposite attitude is that microprogramming is a tool of the hardware designer and not the user. What is the relationship of these two attitudes?

The implementation of a microprogram may utilize a read-only memory (ROM) or a writable-control memory (WCM). The use of ROM to implement the "firmware" [2] is the more common method, but the use of WCM is growing in popularity [3]. What evaluation can be made of the need for WCM as compared to ROM? Could a system utilize both a ROM and a WCM?

The most important question that is to be asked by this thesis is concerned with the uses of microprogramming. A microprogrammable machine is usually noted for its ability to emulate the execution of another machine by changing the ROM or WCM so that a new instruction set is implemented. With the growth of large operating systems, however, a new use for microprogramming is created. Since an operating system often assumes a special hardware configuration, how can microprogramming help provide the desired special properties if they do not exist? What use can be made of the already established ability of microprogramming to extend an instruction set? Can special system primitives be developed to transfer the execution of the system into the ROM? What advantages would this type of implementation have over one that was simply a software feature? What disadvantages would have to be accepted if this manner of implementation were chosen? It is these questions which motivate this thesis.

Section 2 - Overview of the Model

The availability of an INTERDATA Model 4 dictated the machine to be used in the study of microprogramming. Because there is no operating system for the Model 4, the thesis shall first describe a model for a software system. This model will be developed without special regard to the microprogramming capabilities present in the hardware. Then an investigation will be made of the effect that the use of microprogramming a read-only memory might have if the model were going to be implemented. The model that is developed is one which has as a design objective the ability to serve multiple users in an interactive mode. The scope of the model will be limited to the basic elements of the system. The system design itself will be limited in power only to allow greater depth of study in the areas of hardware and memory multiplexing than could occur within the paper if a completely specified or more powerful model were developed. The features not modeled, or modeled only basically, could be expanded to provide state-of-the-art powers at a later time.

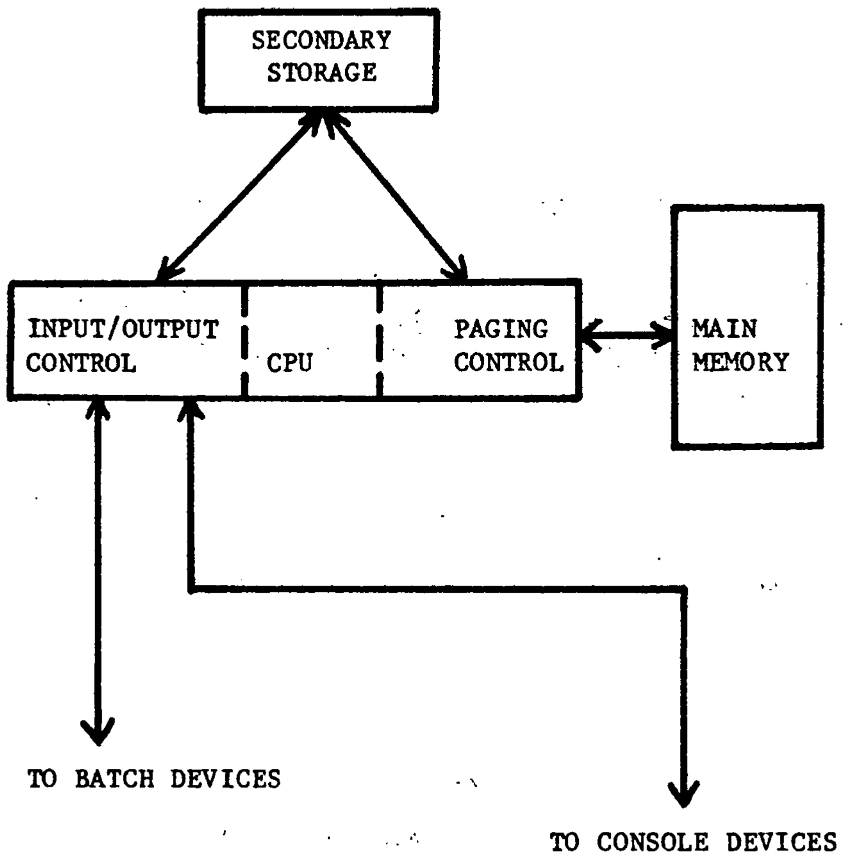
The system design is pieced together from parts of operating systems studied by the author at various times. It is not expected that this operating system model will contribute originally to the body of knowledge about sophisticated operating systems. Rather, as noted, it is to serve as a vehicle for the investigation of the use of microprogrammable read-only memories to ease the implementation of the model; and, at the same time, allow full investigation of desired special system properties before any money is spent to actually "hard-wire" a design.

The model is intended to be logically complete within the routines that specify the hardware and memory multiplexing. Although not always completely specified, such principles as Denning's working set concept [4], demand paging [5], two-component virtual memory addressing [5], process priority recognition [6], virtual processors [4], and interprocess protection [7] are intended to be utilized. Some omissions, such as multiple processes per user [8], intraprocess protection [7], and user execution overlapped with user I/O are not intended to imply a lack of appreciation of the need for study in these areas. Rather, such omissions indicate a desire on the part of the author to restrict the scope of the paper to such proportions that some detailed information may be examined in the included areas.

Section 3 - System Assumptions

It is expected that if this model were going to be implemented one of the implementation criteria would be that the response time of the system to a user be some reasonable amount of time. Almost certainly it should be less than three or four seconds; preferably, as much faster than that as possible. In order for this to be the case the hardware system upon which this model is to be implemented would have to have certain properties. (See Figure 1.) For example, it would have to be the case that there exist some relocation scheme for the system. This could take many forms: address modification, base register usage, or paging [5]. The model assumes that a paging capability is desired. To have this paging capability, memory would be addressed via a two-component addressing scheme so that the page component of the address would actually be a one level indirect address. (See Figure 2.) A second characteristic that the system would need to have would be the capability to overlap computation by the CPU with input and output. A third would be that the system have some moderately high speed and large volume secondary storage devices. The result of the absence of the first two characteristics would be very, very poor response time, but the system could still function. Without large volume secondary storage of at least moderate speed the system would not be at all feasible.

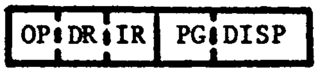
For the purpose of defining and discussing the model a process will be defined as a program together with the elements of the virtual processor on which it executes. For the purpose of this paper multiprocessing and multiprogramming will be synonymous. In this model there may be only one process per user at any given time. The address



Minimum I/O control demands some type of I/O-computation overlap.

Figure 1

Hardware Schematic

Instruction Format: 

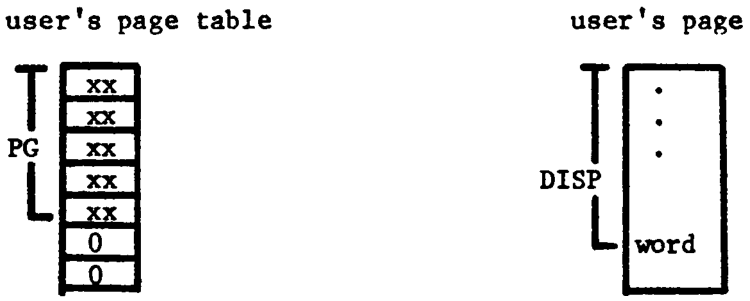
OP = operation code

DR = destination register

IR = index register

PG = page number

DISP = displacement



Memory must be referenced via a two number address. Whether this is used for associative memory referencing of the actual address or index referencing of that address is installation dependent. Thus any instruction addresses memory in such a way that the reference is at least one level indirect so that the address space may be larger than actual memory. The system keeps a directory of information in which the location of every page that belongs to a process is recorded as to its location, either in memory or in secondary storage. The first component of the address is the page reference and the second is the displacement within that page.

Figure 2

Memory Addressing

space of a process refers to those things in memory, either actual or virtual memory, that can be referenced in a single instruction by a statement in a program. Files, until they have been read, are not considered to be in the address space of a process. Any external references used must be resolved as routines are loaded into the virtual memory. To make any system routine, or any other routine, available to a process that routine must, of course, be brought into the address space of that process. This could mean many things. In MULTICS for example, it implies that the segment (unit of virtual memory) containing the desired routine be assigned a segment number and a segment descriptor word for the calling process. This is referred to as "making the segment known to the process" [9]. In this particular model it means copying the desired routine into an area belonging to that process.

Section 4 - Model Description

This is a model of a time-sliced multiprocess operating system. There may be several interactive users interfacing with the system and, as currently restricted by the input/output control system (IOCS), there may be one "batch" job in the system at a time. Batch jobs are treated as special cases of the interactive state. The major purpose of this model is to provide an interactive system, and any batch jobs are considered to be of secondary importance.

A process in the system may be in one of three states: ready active locked (RAL), ready active unlocked (RAU), or blocked (B). A process may change from any one of these states at any time. To which state it changes depends on the reason for the change. These state transitions are summarized by the drawing in Figure 3. The causes for the transitions will be discussed throughout this section. A process in the ready active locked state is a candidate for a time-sliced share of the processor. It is called locked because none of its pages in actual memory may be stolen by another process. A process in the ready active unlocked state is also a candidate for a time-sliced share of the processor but must be locked before it can actually receive control of the processor. It is called unlocked because its pages of actual memory may be stolen by needy processes. A process that is in the blocked state is not a candidate for the processor because it is waiting for some event to occur. For example, the IOCS that is supplied with the model does demand input and output. Therefore, when a process asks for either input or output it automatically goes blocked until that transfer has completed. Then IOCS "wakes up" the blocked process and it becomes ready. These states, although called by

C

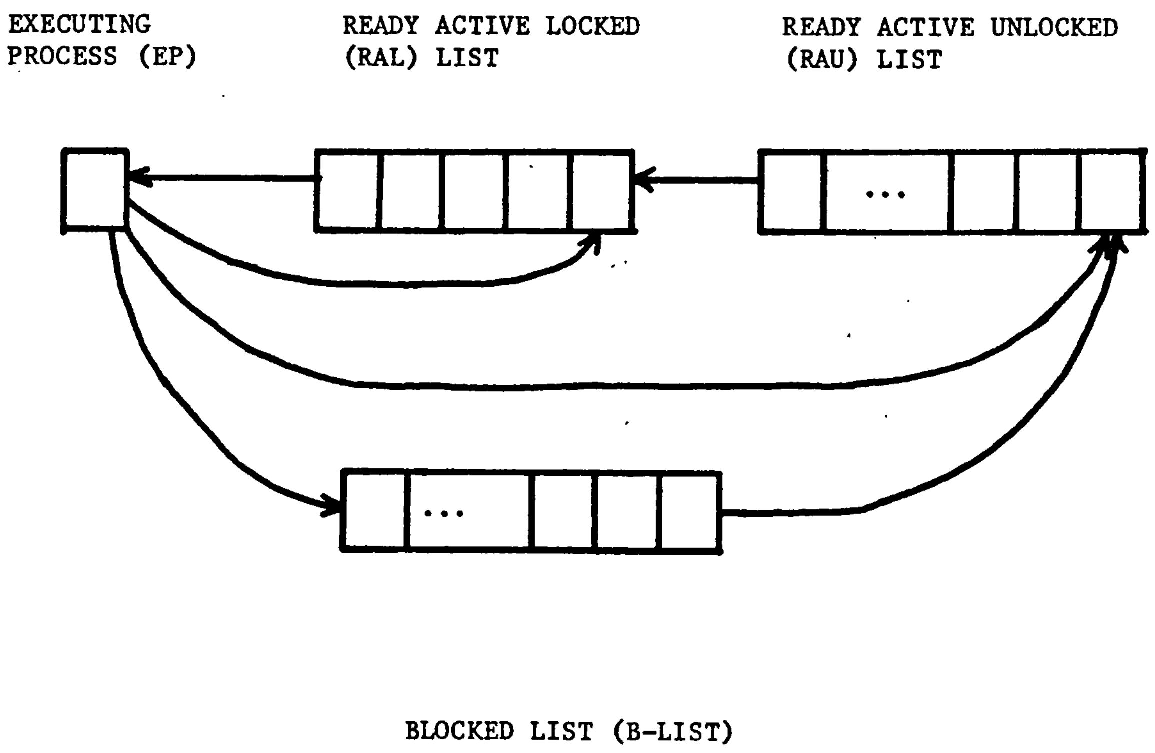

EXECUTING PROCESS (EP)
READY ACTIVE LOCKED (RAL) LIST
READY ACTIVE UNLOCKED (RAU) LIST
...
...
BLOCKED LIST (B-LIST)

Figure 3

Logical Schematic of the Process Lists

different names, were borrowed from the scheduler in the HITAC5020 system [10]. The purpose of having a "locked" state with the pages of a process tied down temporarily is to take a step toward minimizing system "thrashing" [4] where no work is done. The paging routines take other steps to help minimize this problem.

The only way a process may change from the blocked state to the ready state is by a system routine or some other process issuing either an "unblock" or a "ready". The only place that a newly readied process can go is on the end of the ready active unlocked (RAU) list. Similarly, the only way a process may move from the unlocked state to the locked state is by the system doing a "lock". The only place that a newly locked process may go is on the end of the ready active locked (RAL) list. From the RAL list, the process with the highest priority that has been RAL the longest and which is not involved in a paging operation is made the executing process (EP). When this process is to be disposed from EP it may go to the tail of any of the lists. It may not receive two quanta in a row. This was done to prevent one process from monopolizing the system. However, two high priority processes may together monopolize the system. A dynamic priority allocation idea is used to further aid the system in responding to more than one or two high priority processes. The approach taken is derived from the XDS Universal Time-sharing System (UTS) [11]. The job control language (JCL) interpreter of a process should be able to respond more rapidly to the user than any other part of the process. The creation takes place at level one, the interpretation of the JCL at level two, and then the JCL interpreter dynamically assigns the priority level for the procedure being used to some level below level two. When control returns to the JCL routine the priority is reassigned to level two. The basis for this scheme is that when inputing control commands the user expects and deserves rapid response from the system. However, when a user routine is in progress it should have a lower priority than that of some other process that is in the midst of accepting JCL commands.

The system maintains a system directory which contains the file information, the page map for the process's pages, the accounting information for the process, a pointer to the computation activity block (CAB), and any other information that an implementation might need. (See Figure 4.) The directory information may be stored in any convenient manner. For the purpose of this model we may assume that there is a caretaker program which performs all manipulation of data within the directory and that look ups or entries of data are done automatically. This omission will allow the directory to be stored in a machine and installation dependent way, thus allowing for maximum speed and system versatility. It also allows the thesis model to access this information and not be forced to delve into details which are superfluous to the goals of the thesis.

The system has a vector of page locations called the current memory map (CMM). (See Figure 5.) Each process has its own copy of the CMM which is loaded into the actual CMM when that process is given control. This is done by accessing the process's map stored in the system directory. This map is then moved to the CMM which is used for the indirect referencing feature. Loading the CMM is an overt action of the system. When the operating system is given control via a trap or interrupt, one of the first things that it must do is save the CMM

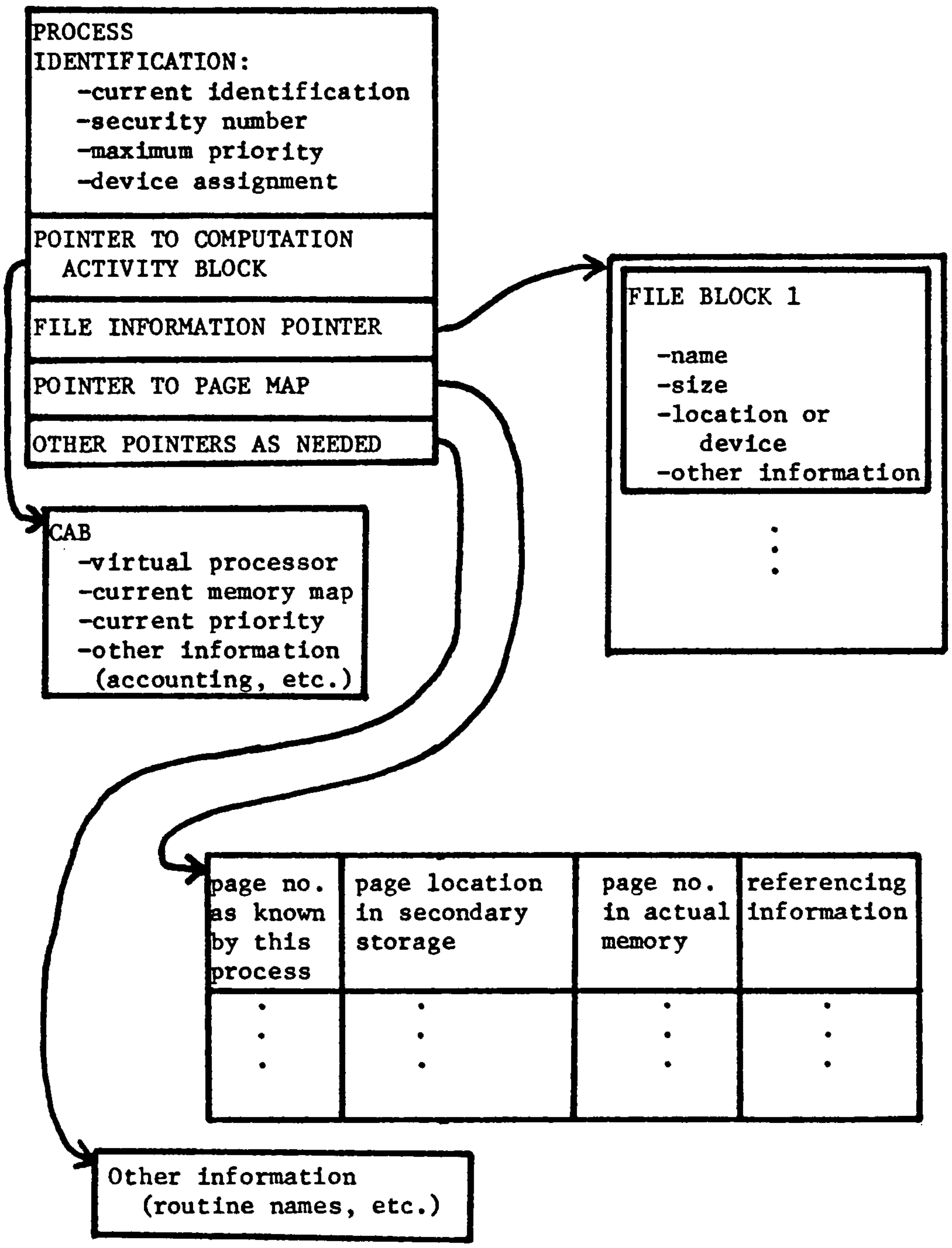
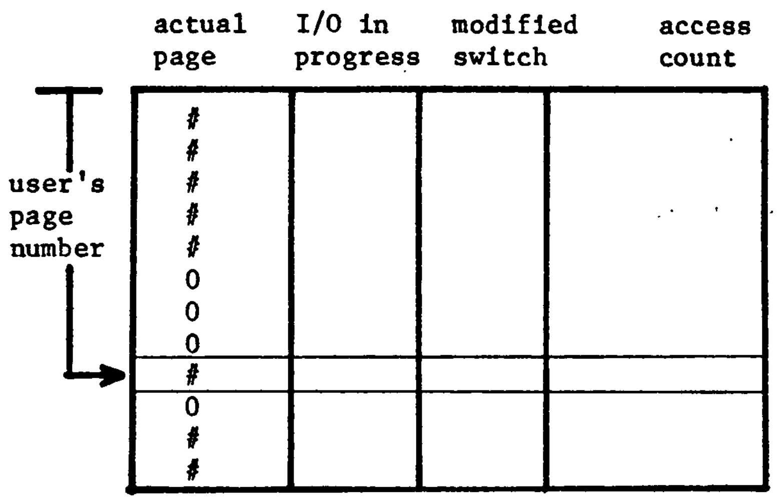


Figure 4

System Directory Structure



Presence of zero in actual page field may mean different things to supervisor and user programs. The map itself may be hardware, firmware, or software.

Figure 5

Current Memory Map (CMM)

of the interrupted process in the directory. Then the system loads its own copy of the CMM. A user's CMM may have several zero entries. Such entries are used to indicate missing pages that either are not in memory or are not in the address space of the process. The current memory map of the operating system reflects the actual configuration of memory elements. The user process's CMM contains the actual location of the logical page referenced, whether that page has been changed since being loaded into memory or not, and a count field of the number of accesses. The system CMM contains the page map for the system routines which are in actual memory while the full page map shows an indicator (perhaps a pointer) to the CAB to which that page currently belongs. The system CMM also contains a switch for all pages currently involved in I/O operations. It should be noted at this point that this type of memory map implies an indexive mapping as available on the XDS Sigma 7 rather than the associative mapping that is utilized by some other computer systems [10,12]. If the ultimate implementation is to be on a machine with associative memory mapping, then the CMM should be considered to be an information vector (possibly similar to the known segment table in MULTICS) and the letters could denote current memory management.

The hardware multiplexor routines are:

QTICK	Clock Interrupt Routine
QTIMEOUT	Quantum Exhaustion Routine
ENTER	Process Execution Routine
EXEC	Process Selection Routine
READY	Process Readying Routine
PRIORITY	Priority Setting Routine

LOCK	Locking Selector Routine
UNBLOCK	Ready Invoking Routine
BLOCK	Blocking Routine
RALKIKOUT	Unlocking Selector Routine

The memory multiplexor routines are:

PAGER	Paging Handler
PAGEFAULT	Page Fault Interrupt Handler
FINDPAGE	Page Freeing Routine
FOUND	Paging Input Initiator

The routines of the IOCS and file maintenance are:

OUTPUTER	Output Controller
INPUTER	Input Controller
CREATE	File Creation Routine
DELETE	File Deletion Routine
ASSIGN	Device Assignment Routine
FREE	Device Releasing Routine
READ	Input Initiation Routine
WRITE	Output Initiation Routine
I/O INTERRUPT	Input/Output Interrupt Handler

The routines associated with process creation and maintenance are:

CREATOR	Process Creation Routine
SWITCHER	Activation Character Decoder
KILL	Process Deletion Routine

The names were chosen to reflect the duties of each routine. On some systems the hardware will perform the indicated functions and these will then be primitive commands rather than system routine calls.

Exact algorithmic details of the duties of these routines appear as figures in the next section. Before seeing the details of the routines, following the progress of a process through the system, from beginning to end, would be instructive.

First, it is assumed that a person sits down at a console device and turns that device on. It is then necessary to be sure that the device is physically connected to the system. This may or may not be anything other than seeing that the plug is plugged in or may involve dialing a telephone. Eventually, when the connection is made, a key is depressed on the device. Either the processor is one that has polling hardware and thus senses this character, or the depression of the key causes an interrupt in the processor. In either case, the character is read and IOCS notices that there is no entry in the INPUT queue for this device. IOCS then activates the routine called SWITCHER if the character that was received was an "attention-getting" character. If not, the input is ignored. The user must, therefore, press a character which has been chosen by the system as an "attention-getting" character, probably from among those characters not usually used for anything else. When one of these is transmitted, the IOCS calls the routine SWITCHER. SWITCHER decides if the user is interrupting a process in progress or wishing to enter the system. If no process already exists that is using this device then the system routine CREATOR, which is treated as a system process, is readied at priority level one with the interrupting device entered in the CREATION queue. CREATOR will request a log-on from the user to verify that he has permission to utilize the system. The user has some number of tries, which can be determined as a system parameter, before the connection is broken and he must reconnect. When the log-on is verified, CREATOR uses a template of a CAB and copies this CAB into an available area in secondary storage. The data in the CAB is initialized in the blocked state. Then a copy of the JCL interpreter is made into the user's address space. The caretaker of the system directory is requested to insert the new items in the directory. A process identification (PROC ID) is assigned and transmitted to the user. The process is then readied by CREATOR, at priority level two, so that the JCL routine may begin to interpret what the user wishes to do. When this task is finished CREATOR looks for further creation tasks by looking in the CREATION queue. If it is empty, this process blocks itself.

The user will now communicate with his process. This process takes its turn getting CPU time as decided by the hardware multiplexor. Via the JCL routine, the programs in the system library are made available to the user by copying system files and executing them as routines of his process at various levels of priority. To get out of the system the user causes control to be returned to the JCL routine and indicates that it is to call the routine KILL which deletes the executing process, the caller in this case, from the system.

How a user gets in and out of the system should now be clear. How he fares while in the system is not yet fully clear. For the remainder of the descriptions a specific process, call it P, shall be followed. The computation activity block of P is referred to by CAB (P). After the log-on procedure described previously, P begins by having a CAB created in the blocked state. This is done so that secondary storage may be used for the creation process and the CAB (P) may be simply moved to the blocked list by the caretaker program.

A READY is performed for the CAB (P) by CREATOR and the CAB (P) is moved to the end of the RAU list with a priority level of two. Eventually, P will become either the highest priority process or the level two process that has been in the RAU list for the longest time. It is then selected to be locked by the LOCK routine and the paging routines initiate paging-in of the page containing the entry point of the JCL routine. This entry point is the address pointed to by the location counter (LC) of CAB (P). The process resides on the RAL list until it is the oldest process or the highest priority process on the list and paging has completed. Then, it is selected by the EXEC routine to be given control of the CPU. Its time quantum is set by a formula which is installation dependent. For the sake of discussion, the following formula for the time quantum may be assumed. Let QUANTUM = (M*10)/(N*PR) where M is the maximum number of processes to be allowed in the system, N is the number of users currently in the system, and PR is the priority level of the process receiving the quantum. The process is now given control of the CPU. One of the first things that the JCL routine must do is execute the PRIORITY primitive to assure that the process is at level two since, later, it can be entered by being called by a user routine.

Eventually, the quantum will pass and this process will be moved to the end of the RAU list or the RAL list depending upon the presence or absence, respectively, of other processes with the same or higher priority. It is possible, however, that while the process was executing it tried to do I/O in which case it would have gone blocked to wait for the I/O to complete. IOCS then takes the responsibility of issuing a READY when the I/O has completed. Thus the cycle is begun again. If the quantum has elapsed, and the process moved to the RAU list or the tail end of the RAL list, then its turn will come again, and the process will continue to receive execution time. The other possibility that could occur while P is executing is the page fault. The occurrence of a page fault could move P to the tail of the RAL list with its paging switch on. When the paging operation has completed, the paging routines turn off that switch and P is not overlooked in receiving a quantum of execution time.

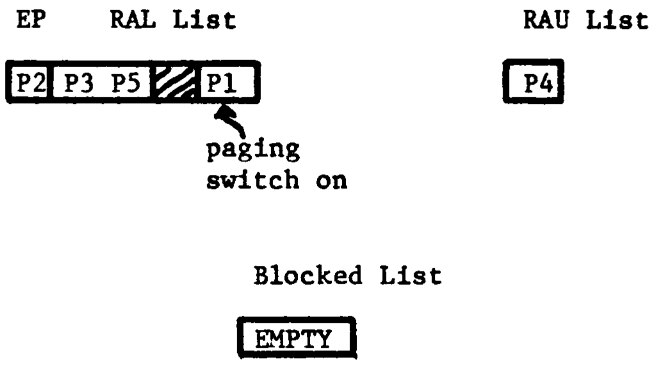
In order to follow the system through its paging philosophy, suppose there are five processes, P1, P2, P3, P4, and P5, and that all are of the same priority. Suppose that P1 is the EP and all are RAL.

EP	RAL List	RAU List
P1	P2 P3 P4 P5	EMPTY

Blocked List

EMPTY

Suppose that during the quantum assigned to P1, it has a page fault. The paging routines try to find a blocked process from which to steal a page. But, none exist so the RAU list is searched for the lowest priority process that has been RAU the longest. But none of these exist either. Therefore, the routine responsible for finding a page for P1, FINDPAGE, has a problem. All pages of memory are occupied by locked processes. The only course open to the system is to unlock one of the processes. RALKIKOUT is the routine responsible for the selection of one of the RAL processes other than P1 (since there are some) which is then made RAU and FINDPAGE can then select one of its pages. This routine, RALKIKOUT, chooses the lowest priority process in the RAL list and among those of equally low priority it selects the process that has been in the system the longest. For example, P4.



P4 is then made an unlocked process. This is done because it is felt that a process that has been in the system for a long time (relative to the other processes in the system) has had the most opportunity to get work done. Therefore, newer processes should have an opportunity to get some work accomplished. This attitude is derived from the philosophy adopted for the early time-sharing system (TSS) at MIT as reported by Corbató and others [13]. A similar attitude was chosen for the XDS UTS system [11]. P1 then gets a page from that process. The system resumes operation and selects some process other than P1 to get an execution time quantum. This is done to allow time for the paging operation to be completed.

Section 5 - System Algorithms

This section contains the figures which show the algorithms of the routines and primitives of the system. Double asterisks at the end of a statement indicate a terminal exit. No return is ever executed to the next statement. The number of processes that may be in the locked state is limited as a parameter of the system implementation. Note that no attempt is made to provide for multiprocessor situations.

<u>QTICK</u>

```
    - save virtual processor

    - subtract one from QUANTUM of EP

    - is QUANTUM equal to zero?

         - no  - ENTER EP at interrupted point **

         - yes - call QTIMEOUT **
```

Figure 6

Clock Interrupt Routine

<u>QTIMEOUT</u>

```
    - lock q timer

    - save virtual processor

    - is there a CAB in RAU list with a priority greater than or
      equal to process EP?

         - yes - place EP at tail of RAU list and unlock it

               - delete EP from RAL list

               - LOCK a process

               - call EXEC **

         - no  - place EP at tail of RAL list

               - LOCK a process

               - call EXEC **
```

Figure 7

Quantum Exhaustion Routine

ENTER

```
- is entry point of process in memory?
    - yes - restore virtual processor
          - unlock q timer
          - branch to entry point (LC of CAB) **
    - no  - move CAB of EP to tail of RAL list
          - set paging switch to on
          - call PAGER to initiate page transfer (CAB)
          - call EXEC **
```

EP points to the process to be entered. This routine attempts to give the CPU to the EP process.

Figure 8

Process Execution Routine

EXEC

```
- set EP to point to CAB of highest priority in RAL list
  which does not have its paging switch on
- initialize QUANTUM of EP
- ENTER EP **
```

This routine selects the process with the highest priority which does not have the paging switch on and gives the processor to it.

Figure 9

Process Execution Selector

```
READY (CAB)

        - lock q timer

        - save virtual processor

        - is CAB on RAL or RAU list?

             - yes - set wake-up switch to on

                   - unlock q timer

                   - return **

             - no  - place CAB on tail of RAU list

                   - set wake-up switch to off

                   - delete CAB from B-list

                   - restore virtual processor

                   - unlock q timer

                   - return **
```

Figure 10

Readying Routine

```
PRIORITY (CAB, level)

        - find out from directory if called from within the limits of
          the JCL routine

            - no  - return **

            - yes - set priority field of this CAB to level

                  - return **
```

Figure 11

Priority Setting Routine

```
LOCK (CAB)

        - is there a process CAB in the RAU list?

            - no  - return **

            - yes - is RAL list full? (number depends on
                    installation)

                  - yes - return **

                  - no  - search RAU list for highest priority CAB

                        - set locked switch to on

                        - move CAB to RAL list

                        - set paging switch to on

                        - call PAGER for this CAB for at least LC
                          (possibly for complete working set)

                        - return **
```

Figure 12

Locking Selector Routine

```
UNBLOCK (PROC ID)

        - get CAB address from directory

        - READY (CAB)

        - return **
```

Figure 13

Ready Invoking Routine

```
BLOCK (EP)

        - lock q timer and save CPU conditions

        - is wake-up switch on?

             - yes - set wake-up switch to off

                   - unlock q timer

                   - return **

             - no  - put CAB on B-list

                   - unlock CAB and delete from RAL list

                   - call EXEC **
```

Figure 14

Blocking Routine

```
RALKIKOUT (CAB, sw)

        - are there any RAL processes except CAB?

            - no  - is any page of CAB not involved in I/O?

                  - no  - re-enter FINDPAGE without disturbing linkage
                          and parameters (thus wait for I/O to
                          complete and produce a free page)

                  - yes - set sw equal to this page

                        - return **

            - yes - from among the processes at the lowest priority
                    level in RAL list select the one that has been in
                    the system the longest and put it at the head of
                    RAU list and set it unlocked

                  - set sw equal to zero

                  - return **
```

Figure 15

Unlocking Selector Routine

```
PAGER (CAB ptr)

        - is page containing LC of CAB in memory?

            - yes - is page containing the word pointed to by the
                    address field of the instruction referenced by
                    the LC field in memory?

                - yes - turn paging switch off

                      - return **

                - no  - look up secondary storage address in the
                        directory for this page

                   P1 - is there a request for this page in the
                        INPUT queue?

                      - yes - return **

                      - no  - place this page number and CAB in
                              INPUT queue ahead of any non-paging
                              requests

                            - call FINDPAGE (CAB, sw)

                            - is sw greater than zero?

                            - yes - put sw as actual page for this
                                    new entry into INPUT queue

                                  - call INPUTER

                                  - return **

                            - no  - is sw equal to zero?

                                  - yes - return **

                                  - no  - turn paging switch off

                                        - return **

            - no  - look up secondary storage address in directory for
                    this process for page containing LC

                  - go to P1
```

Figure 16

Paging Handler

PAGEFAULT

```
        - save virtual processor
        - lock q timer
        - turn paging switch on
        - call PAGER (EP)
        - is paging switch on?
            - no  - unlock q timer
                  - restore processor **
            - yes - delete CAB (EP) and place on tail of RAL list
                  - call EXEC **
```

When an instruction tries to reference a page that is not in memory, this routine is activated.

Figure 17

Page Fault Interrupt Handler

```
FINDPAGE (CAB, sw)

        - are there any free pages in CMM?

            - yes - set sw to one

                  - return **

            - no  - are there any B-list processes?

                  - yes - set used switches to off for all B-list CABs

                   FP1 - set pointer to CAB of lowest priority and
                         rear-most of processes with used switches
                         off

                       - are there any pages in this process not
                         being used for I/O?

                       - yes - select the least used page in this
                               process

                         FP4 - has this page been changed since
                               loaded into memory?

                             - no  - set directory to secondary
                                     storage address

                                   - set sw to actual page in memory

                                   - return **

                             - yes - put this page and appropriate
                                     information in OUTPUT queue

                                   - call OUTPUTER

                                   - set sw equal to zero

                                   - return **

                       - no  - set used switch to on in this CAB

                             - is there another CAB in this list?

                             - yes - go to FP1

                             - no  - go to FP2
```

Figure 18

Page Freeing Routine

```
FP2 - no  - are there any RAU processes?

      FP5 - no  - is this CAB the EP?

                - no  - set sw equal to minus one

                      - return **

                - yes - call RALKIKOUT (CAB, sw)

                      - is sw equal to zero?

                      - yes - go to FP2

                      - no  - set ptr to page pointed
                              to by sw

                            - go to FP4

          - yes - set all used switches to off for CABs
                  in RAU list

            FP3 - set ptr to CAB of lowest priority and
                  rear-most of processes with used
                  switches off

                - are there any pages in this process
                  not involved in I/O?

                - yes - select the least used page in
                        this process

                      - go to FP4

                - no  - set used switch to on

                      - are there any other CABs in this
                        list?

                      - yes - go to FP3

                      - no  - go to FP5
```

If a page was found, its number is returned in sw. If no page was found, but a pageout was initiated, sw is set equal to zero. If no paging was initiated sw is set equal to minus one.

Figure 18 (continued)

```
FOUND (ptr)

        - pick up actual page from entry in OUTPUT queue pointed to
          by ptr

        - delete this entry

        - put actual page in destination page address of first paging
          entry in INPUT queue

        - call INPUTER

        - return **
```

Figure 19

Paging Input Initiator

```
OUTPUTER

        - find first entry in OUTPUT queue that is not in progress

          I2 - is the device busy?

                   - yes - move to next item

                         - is this the end of the queue?

                         - yes - return **

                         - no  - go to I2

                   - no  - set in progress switch to on

                         - issue WRITE command

                         - return **
```

Figure 20

Output Controller

```
INPUTER

        - find first entry in INPUT queue that is not in progress

         I1 - is the device busy?

                  - yes - move to next item

                        - is this the end of the queue?

                        - yes - return **

                        - no  - go to I1

                  - no  - set in progress switch to on

                        - issue READ command

                        - return **
```

Figure 21

Input Controller

```
CREATE (file, size, PROC ID)

        - is a file already created with this name for this process?

             - yes - return **

             - no  - search directory for space of size

                   - if not found - output error message

                                  - call KILL (EP) **

                   - if found     - put secondary storage address into
                                    the directory for this PROC ID

                                  - delete this space from available
                                    secondary storage list

                                  - put file name into directory

                                  - return **
```

Figure 22

File Creation Routine

```
DELETE (file name)

        - is there a file by this name?

             - no  - return **

             - yes - put this space on available list

                   - return **
```

Figure 23

File Deletion Routine

```
ASSIGN (object, code, direction, record, account number)

    - is CAB system device an interactive device?

        - yes - return **

        - no  - is direction IN?

            - no  - is the object a file?

                - no  - set CAB output device to point to
                        object of type, code

                      - return **

                - yes - is account number present?

                  AS1 - no  - get secondary storage address of
                              the file from directory and put
                              in CAB output device

                            - return **

                      - yes - find permanent file of name
                              given in object under account
                              given in account number

                            - go to AS1

            - yes - is the object a file?

                - no  - set CAB input device to point to
                        object of type, code

                      - return **

                - yes - is account number present?

                  AS2 - no  - get secondary storage address
                              of the file from directory and
                              put in CAB input device

                            - return **

                      - yes - find permanent file of name
                              given in object under account
                              given in account number

                            - go to AS2
```

Figure 24

Device Assignment Routine

```
FREE (device, sw)

        - remove from CAB

        - remainder of action depends on device

              i.e., if tape - write end of file and rewind

                    if printer - skip to a new page and print system
                                 header

                    etc.
```

Figure 25

Device Releasing Routine

```
READ (address, size)

        - look in the CAB of EP to see if either a device or a file
          has been assigned

             - no  - output "no device or file assigned for read"
                     message to the system output device

                   - call KILL (EP) **

             - yes - is it a device?

                   - yes - is it capable of input?

                         - no  - output error message

                               - call KILL (EP) **

                         - yes - set page containing address to show
                                 I/O is in progress

                               - put address, device, 0, size, CAB
                                 pointer on INPUT queue

                               - call INPUTER

                               - BLOCK (EP)

                               - return **

                   - no  - has a file been created?

                         - yes - look up secondary storage address for
                                 file, record number

                               - put address, device, secondary storage
                                 address, size, CAB pointer, record
                                 number on INPUT queue

                               - call INPUTER

                               - BLOCK (EP)

                               - return **

                         - no  - output error message

                               - call KILL (EP) **
```

Figure 26

Input Initiation Routine

```
WRITE (address, size)

     - look in the CAB of EP to see if either a device or a file has
       been assigned

          - no  - output "no device or file assigned for write"
                  message to the system output device

                - call KILL (EP) **

          - yes - is it a device?

                - yes - is it capable of output?

                      - no  - output error message

                            - call KILL (EP) **

                      - yes - set page containing address to show
                              I/O is in progress

                            - put address, device, 0, size, CAB
                              pointer on OUTPUT queue

                            - call OUTPUTER

                            - BLOCK (EP)

                            - return **

                - no  - has a file been created?

                   W1 - yes - is record too big, i.e. past EOF?

                            - yes - output error message

                                  - call KILL (EP) **

                            - no  - look up secondary storage
                                    address for file, record number

                                  - put address, file, file address,
                                    size, CAB pointer, record number
                                    on OUTPUT queue

                                  - set I/O in progress switch on

                                  - call OUTPUTER
```

Figure 27

Output Initiation Routine

```
                    - BLOCK (EP)
                    - return **
          - no  - output error message
                - call KILL (EP) **
```

Figure 27 (continued)

I/O INTERRUPT

```
- save processor

- lock q timer

- discover interrupting device

- was transfer in or out?

    - in  - was transfer requested by an entry in the INPUT
            queue?

          - yes - was this a page transfer?

                - yes - set page address in directory to
                        actual page address

                      - delete entry from INPUT queue

                      - turn paging switch off in CAB

                      - turn I/O in progress switch off for
                        appropriate page in CMM

                      - call INPUTER

                      - unlock q timer

                      - restore processor **

                - no  - turn I/O in progress switch off for
                        appropriate page in CMM

                      - call UNBLOCK for this PROC ID

                      - delete appropriate entry in INPUT
                        queue

                INT 1 - call INPUTER

                      - call OUTPUTER

                      - unlock q timer

                      - restore processor **

          - no  - call SWITCHER (device) **
```

Figure 28

Input Output Interrupt Handler

```
- out - was this a page transfer?

      - yes - call FOUND (ptr) with ptr setting to entry
              in queue that caused interrupt

            - call OUTPUTER

            - unlock q timer

            - restore processor **

      - no  - turn I/O in progress switch off for
              appropriate page in CMM

            - call UNBLOCK for this PROC ID

            - delete entry in OUTPUT queue

            - go to INT 1
```

Figure 28 (continued)

CREATOR

```
C1 - is there an entry in the CREATION queue?

         - no  - BLOCK SELF

               - go to C1

         - yes - is there sufficient secondary storage to create
                 another process?

            C3 - no  - output appropriate message

                     - delete entry from CREATION queue

                     - go to C1

               - yes - are there M users in the system?

                     - yes - go to C3

                     - no  - output log-on request to all devices
                             for which there are entries in queue
                             for which this has not been done

                           - set try counter to 1

                        C2 - input log-on information

                           - is log-on valid?

                           - no  - increment try counter

                                 - is try counter less than or
                                   equal to maximum number?

                                 - yes - go to C2

                                 - no  - delete entry from CREATION
                                         queue

                                       - BLOCK SELF

                                       - go to C1

                           - yes - copy a CAB onto new place in
                                   secondary storage for the first
                                   process on the CREATION list
                                   that does not have one
```

Figure 29

Process Creation Routine

```
- insert necessary information
  into that CAB

- copy JCL routine into this file

- place CAB location on secondary
  storage into system directory as
  a BLOCKED process

- assign a PROC ID and transmit it
  to the user

- set priority to level 2

- call UNBLOCK (PROC ID)

- go to C1
```

Figure 29 (continued)

SWITCHER (device)

```
    - is there a process assigned to this device?

        - no  - put entry in OUTPUT queue to output system
                response indicator

              - execute a READY for CREATOR's CAB

              - put device address in CREATION queue

              - call OUTPUTER

              - unlock q timer

              - restore processor **

        - yes - place entry point of JCL copy into LC of CAB
                assigned to this device

              - is CAB ready?

              - no   - READY (CAB)

                     - are any I/O in progress switches set for
                       this CAB?

                     - yes - turn off all that are not paging
                             switches

                           - delete same from INPUT queue and
                             OUTPUT queue

                           - go to S1

                     - no  - go to S1

           S1 - yes - unlock q timer

                    - restore processor **
```

Figure 30

Activation Character Decoder

```
KILL (ptr)

        - return all non-permanent files to the available secondary
          storage list and delete from directory

        - return all pages in secondary storage to available status

        - terminate all I/O for pages belonging to this process

        - set CMM to indicate all pages of this process in memory are
          free

        - disconnect system device from the CAB

        - delete the CAB

        - is this CAB the EP?

            - yes - call EXEC **

            - no  - ENTER EP **
```

Figure 31

Process Deletion Routine

An entry in the INPUT or OUTPUT queue contains at least the following:

address of buffer

size of buffer

CAB address

paging switch

actual page address

secondary storage address

CAB's page number

in progress switch

device address or file address

Notice that the address of the buffer could contain a buffer address if the paging switch is off, or an actual page address if the paging switch is on. Similarly, some of the other fields could do double duty depending upon whether the entry is for page transfer or for regular I/O.

Figure 32

Input and Output Queue Description

Chapter 2

MICROPROGRAMMING

Section 1 - Introduction

M. V. Wilkes, in an article for Computing Surveys, summarizes the reasons microprogramming appeals to designers by the following statements:

> 1. It provides economical means whereby the smaller machines of a series can have large instruction sets compatible with those on the larger ones.
> 2. Maintenance aids can be provided; for example, the read-only memory can have a parity bit, and special diagnostic microroutines can be provided for the use of the maintenance engineers.
> 3. Emulation is possible.
> 4. Flexibility exists to provide new features in the future.[3]

Among the hardware manufacturers who currently use microprogramming in the construction of their machines, the two attitudes mentioned in Chapter 1 are prevalent. There are, of course, many variations between these two extremes. An interesting observation is that IBM, when announcing the System/360, chose to allow the user access to the microprograms but by the time the system was readily available had decided microprogramming was exclusively a designer's tool. Another manufacturer has adopted the permissive attitude. INTERDATA has provided the user the ability to microprogram the Model 3 and the Model 4.

This chapter presents evidence to be used in an evaluation of the capabilities of microprogramming in the area of operating systems and the emulation of special properties for that system.

Section 2 - Overview of Implementation Features

A number of existing microprogrammable systems could be studied to investigate the fourth reason, the flexibility inherent in microprogramming, noted by Wilkes in the previous section. An alternative to the use of an existing system for this study would be the postulation of a hypothetical machine which would exhibit the characteristics common to microprogrammable processors. Some such postulations have been done [14], but due to the availability of an actual machine, the INTERDATA Model 4 was chosen as the vehicle to be used for the investigation.

The discussion of microprogramming and the microprograms that are included are not intended to implement the entire operating system modeled in the first chapter. Instead, the uses of microprogramming will be investigated with an interest in the underlying principle of the adaptability of the microprogrammable machine. Some postulation will occur in regard to specific features of the operating system, but the results of the investigation will be used as evidence in attempting to answer the questions posed in the first section of Chapter 1. The example microroutines presented later in this chapter to illustrate the coding necessary for implementation of the normal user-machine are not necessarily those provided by INTERDATA. Rather, the reference manual was studied, and the routines written as an example of what the code might be. In order to verify that the micro-instructions worked in the manner described herein (see Appendix A), a microprogram to emulate a simple instruction set was prepared. This program was then assembled and a binary tape produced. The printed output from this assembly appears in Appendix D. This binary tape was then loaded into the microsimulator "PANDORA" provided by INTERDATA to test microroutines. A small user program was then executed by the simulated micromachine. The response of the simulated micromachine was as expected. If the simulator was working properly the micro-instructions and phase relationships described in Appendix A were properly understood.

Section 3 - Hardware Description

For the purpose of discussion and distinction, the machine which interprets and executes microprograms is called the micromachine while the "emulated" machine is called the user-machine. The reason that emulation is straightforward is that all user-machines are emulated, and one need only choose the characteristics that he wishes to emulate.

The INTERDATA Model 4 recognizes ten instructions. These are:

ASSEMBLER SYMBOL	DEFINITION
A	Add
S	Subtract
X	Exclusive OR
N	AND
O	Inclusive OR
L	Load
C	Command
T	Test
B	Branch on Condition
D	Decode

When writing the microprogram, the designer has many of the standard assembler capabilities if he uses the micro-assembler provided by INTERDATA. He may use symbolic names and mnemonics for the commands, as well as make use of special addressing functions and predefined register symbols.

Model 4 processor operation is directed by the read-only memory (ROM). Instructions loaded into the processor are placed in the RD register from which the micro-instruction is decoded.

> ROM with its pre-wired micro-program, directs the Processor through the control unit. Processor Control can, depending on the micro-operation code, set-up the Arithmetic Logic Unit (ALU) to the desired mode of operation, test for specified hardware conditions, issue functional commands to establish hardware conditions, initiate memory cycles, set-up micro-instruction loops, or load and unload selected registers in the hardware register stacks.[15]

The format of a micro-instruction is generally:

```
OP CODE    DESTINATION REG,SOURCE REG,MODIFIERS
```

For example, the micro-instruction

```
L    MAR,MDR,NC
```

says to Load the Memory Address Register with the contents of the Memory Data Register and Not to OR in a bit from the Carry position of the FLG register. There are also some immediate type micro-instructions. For example, the micro-instruction

```
L    MR4,X'01'
```

says to Load into MicroRegister 4 the immediate operand "01". The "X" indicates that the number in quotes is hexidecimal.

In all arithmetic and logical operations, the arithmetic register (AR) is assumed to be one operand. Thus, to add a one to the contents of MicroRegister 4, either of these two sets of micro-instructions would accomplish this task:

```
L    AR,X'01'
A    MR4,MR4,NC
```

or

```
L    AR,MR4,NC
A    MR4,X'01'
```

A list of the registers accessible by the micromachine appears in Appendix B and the execution times of the micro-instructions appear in Appendix C. For a more complete specification of the details of microprogramming the Model 4, the reader is referred to the Model 4 reference manual [15].

The Model 4 has a special Decode Read-Only-Memory (DROM) which is used to address the microroutines that perform user's instructions. While the INTERDATA Model 3 has no particular orientation beyond byte addressing of memory, the Model 4 is oriented toward the standard INTERDATA user instruction set.

> In the Model 4 there are four hardware conditions known as "phases". Each phase has corresponding sets of micro-instructions. In general, Phase Zero is dedicated to user's instruction fetch and class decoding: Phase One to indexing for the second operand; Phase Two to user's instruction execution; and Phase Three for interrupt service and display support. These phases effect and in-turn are effected only by the Decode micro-instruction. When the Decode micro-instruction is used to bring about a phase change, the subsequent state of the phase pointer is dependent on user's instruction format; whether or not the instruction is indexed and the current state of the phase pointer.[15]

The diagram in Figure 33 shows the flow through the phases, and Appendix A presents a summary of the instructions necessary to move through the phases. In addition, Appendix A shows pictorially the actions of the location pointer and the contents of the registers for the necessary parts of the phases of the normal machine.

The Model 4 has a configuration available which has high speed secondary storage via a "selector channel" connected to the high speed memory bus. In addition, there is a capability for a "direct memory access channel". The multiplexor bus of the Model 4 controls the devices not connected to the high speed selector channel. (See Figure 34.) The actual details of input/output are avoided to accentuate the de-emphasis of attention to the IOCS and, again, allow concentration on the properties of hardware and memory multiplexing. The response to

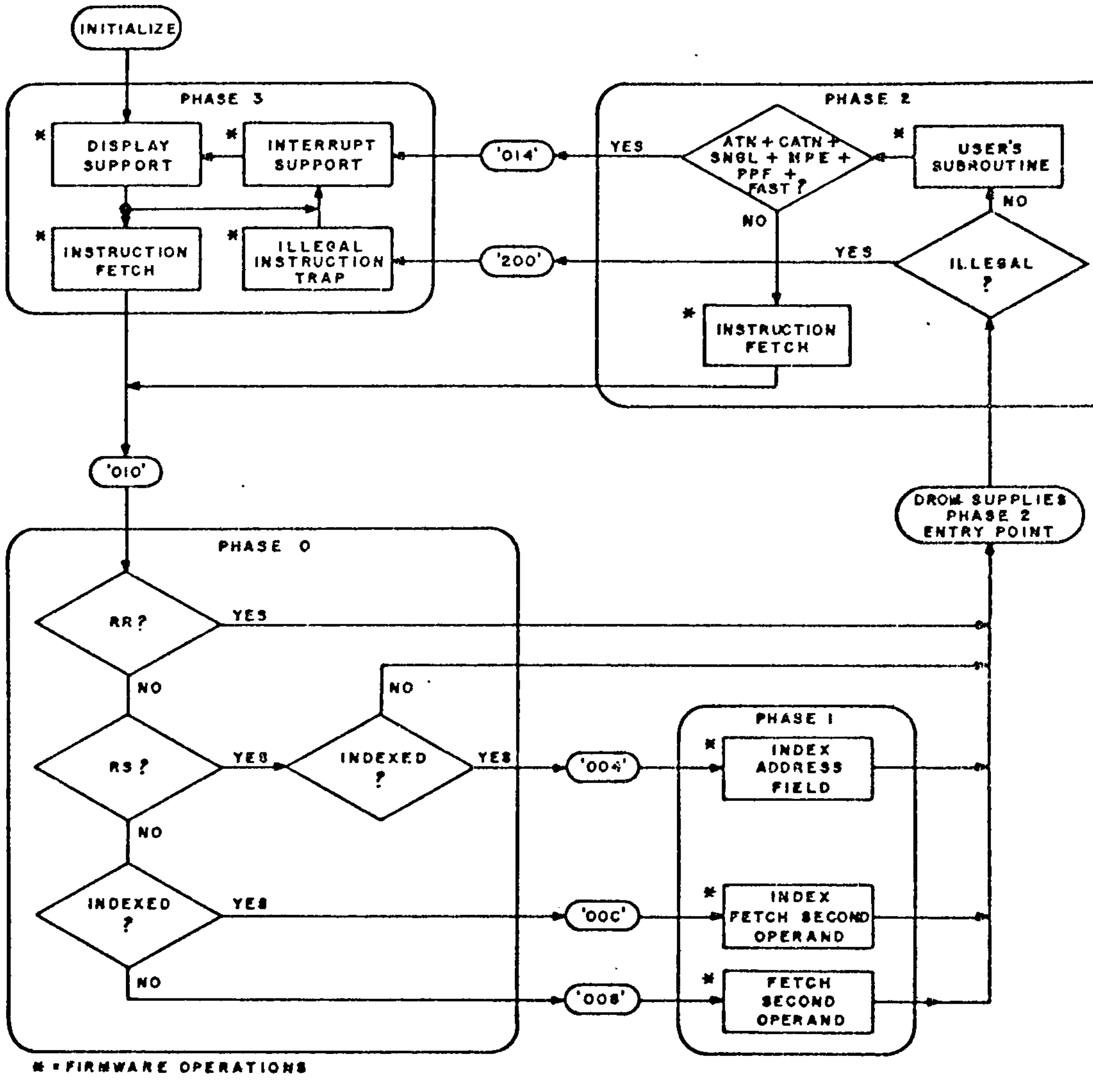


Figure 33

General Flow of User Instruction [15]

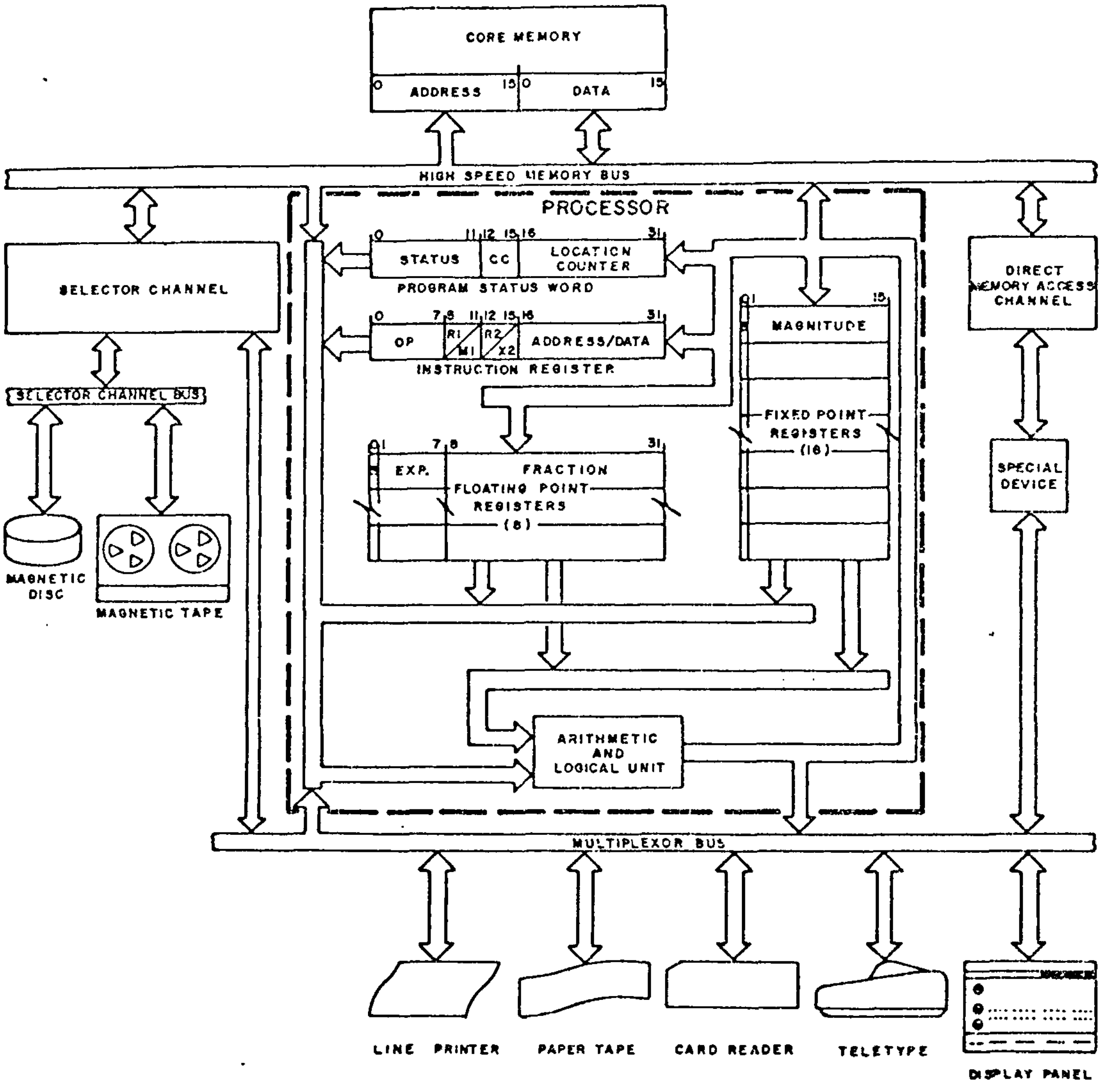


Figure 34

Model 4 System Block Diagram [16]

I/O that must occur in the micromachine occurs in phase three, and no discussion of phase three beyond the code necessary to enter phase zero will occur.

Section 4 - Firmware Implementation

The normal Model 4 user instruction set is similar to that found on the IBM System/360 [17]. The major difference between the two is the lack of the base register in the Model 4. Since the micro-machine is oriented toward the implementation of the normal user-machine, a brief discussion of the format of the normal Model 4 user instructions will clarify the microcode examples given later.

In the normal Model 4, there are three basic types of user instructions: RR, RX, and RS. Those user instructions which move information from user register to user register are RR format instructions. These user instructions are one halfword in length and have the general format:

OP CODE   REG ONE REG TWO

The eight bit operation codes which are recognized by the hardware as RR format user instructions are: X'0n', X'2n', and X'9n'.

The other two types of user instructions each take one full word of memory. Their general format is:

OP CODE   REG ONE REG TWO ADDRESS FIELD

The RS user instructions are the immediate operand types with the address field containing the operand. The register two field is an index register reference if it is non-zero. The register referenced in the register one field is the destination register. The operation codes which are recognized by the hardware as RS format user instructions are: X'Cn' and X'En'. The RX user instructions contain the address of the desired operand in the address field and thus require another memory reference to access the desired operand. As in the RS format user instructions, the register specified by the register two field is used as an index register if the field is non-zero. The operation codes which are recognized by the hardware as RX format user instructions are: X'4n', X'6n', X'Dn', and X'C2'.

In order for these user instructions to be emulated, the type (class) must be decoded by the micromachine. Phase zero of the micromachine is dedicated to decoding the class of a user instruction in memory pointed to by the LOC register. Phase zero micro-instructions begin at X'010' in the ROM. This address, as well as those associated with all phases except the ones generated by the DROM to enter phase two, are predetermined by the hardware according to user instruction type being emulated and current phase of the micromachine. When phase one is entered, if it is at all, the address chosen is X'004' for RS-indexed user instructions, X'008' for non-indexed RX user instructions and X'00C' for RX-indexed user instructions. Phase three begins at location X'014'.

Those user instructions which are RS without an index cause the micromachine to go directly from phase zero to phase two and the address in ROM is chosen by the DROM as a function of the operation code. This is also true for RR user instructions. The DROM selects an address in ROM for execution of phase two to begin for the emulation of user instructions for which phase one has been completed. These address constraints must be met by a microprogram whether it is for the emulation of the normal user-machine or a specially prepared one. This preselected addressing speeds up execution on the normal machine. For example, phase zero in the normal machine contains the following micro-instructions:

ROM ADDRESS

```
010        L     IR,MDR,NC
011        L     MAR,LOC,NC
012        D     AR,YD,PO
```

This code sequence is not self explanatory because the execution of phase zero is embedded in the decode (D) micro-instruction which is executed to leave phase zero. It automatically selects either phase one or phase two to be entered and automatically selects the address to which control is to be transferred. The LOC register, which points to the user instruction in memory, is also automatically incremented and a memory read is performed if the destination phase is phase one.

A micro-implementation of a paging feature uses the first 80 hexidecimal locations of core as a page table. This micro-implementation of a property desired by the operating system model will be compared with a micro-implementation of the normal machine. This paging machine is not completely specified, but the components that perform the paging are given in figures along with the conventions needed to finish specifying the user-machine. The phase zero micro-instructions for this paging machine are as follows:

ROM ADDRESS

```
010        C     CUT
011        L     IR,MDR,NC
012        L     MAR,LOC,NC
013        D     AR,YD,PO
```

The only difference in the two phase zero routines is the presence of the command to reset (clear) the utility flip-flop.

The phase one micro-instructions for the normal machine for all three types of user instructions are:

ROM ADDRESS

```
004        L     AR,YS,NC         PHASE 1-RS WITH INDEX
005        A     MDR,MDR,NC+NF
006        D     LOC,LOC,PC

008        L     MAR,MDR,NC       PHASE 1-RX WITHOUT INDEX
009        D     LOC,LOC,P1

00C        L     AR,YS,NC         PHASE 1-RX WITH INDEX
00D        A     MAR,MDR,NC+NF
00E        D     LOC,LOC,P1
```

In the case of indexed user instructions this code adds the contents of the index register into either the address in the RX user instruction or the "effective" data in the RS user instruction and then executes a decode micro-instruction. In the non-indexed RS case, the micro-instructions set up the registers for a memory reference and then a decode micro-instruction is executed. The decode micro-instruction that terminates phase one causes a memory reference to fetch the addressed operands if the IR register contains an RX user instruction. The address to which control is then transferred is selected by an index down the contents of the DROM by the operation code of the user instruction. The contents of that DROM location are then used to specify to what ROM address control is transferred. This same indexive technique occurs upon leaving phase zero if the user operation code is RR or non-indexed RS. Phase two is the new hardware condition.

The phase one microcode that is needed to support automatic paging appears in Figure 35. This represents quite an expansion in the operations that must occur. Some of this expansion was made more difficult by the special properties of the decode instruction that are so helpful in the normal implementation. This phase one code checks both the location counter for paging and the user address in the

```
HEX ADDRESS   OP  OPERAND             COMMENTS

    004       C   CB                  CLEAR BANK FLIP-FLOP
    005       L   CNTR,X'1'           COUNTER REG SET TO 1
    006       L   RAH,H(ADLOC)        BRANCH TO
    007       L   RAL,L(ADLOC)        ADLOC
    008       C   CB                  CLEAR BANK FLIP-FLOP
    009       L   CNTR,X'0'           COUNTER REG SET TO 0
    00A       L   RAH,H(ADLOC)        BRANCH TO
    00B       L   RAL,L(ADLOC)        ADLOC
    00C       C   CB                  CLEAR BANK FLIP-FLOP
    00D       L   CNTR,X'2'           COUNTER REG SET TO 2
    00E       L   RAH,H(ADLOC)        BRANCH TO
    00F       L   RAL,L(ADLOC)        ADLOC

ADLOC         C   SUT                 SET UTILITY FLIP-FLOP
              L   AR,LOC,NC+CS
              N   MR4,X'FC'           GET LOC VALUE
              S   MR4,MR4,NC+NF       IN MR4
              L   AR,MR4,CS+NC
              S   MR4,X'2'
              L   AR,X'0'             TEST IF ZERO
              O   MR4,MR4,NC          BRANCH IF
              B   3,NOCHNG            NO PAGE CHANGE
              L   AR,LOC,NC+CS
              N   MR4,X'FC'
              C   SB
              L   MR0,H(RET2)
              L   AR,MR0,CS+NC
              A   MR0,L(RET2)
              C   CB                  BRANCH TO MAPPING
              L   RAH,H(MAPPER)       ROUTINE   MAPPER
              L   RAL,L(MAPPER)
RET2          B   3,FAULT
              L   MAR,MR4,CS+NC
              C   MR
NOCHNG        L   AR,X'1'
              L   MR0,X'0'
              C   RPT
              A   MR0,MR0,NC+NF
              C   SB
              L   AR,MR0,NC
              L   MR4,X'1'
              S   MR0,MR4,NC
              B   1,ADDRS
              L   AR,YS,NC
              A   MDR,MDR,NC
              L   AR,X'0'
```

Figure 35

Phase One Paging Microcode

```
                    A  MRO,MRO,NC
                    B  2,ADDRS
                    D  LOC,LOC,PC       RS -- EXIT
ADDRS               L  MR4,MDR,CS+NC
                    L  MRO,H(RET)
                    L  AR,MRO,CS+NC
                    A  MRO,L(RET)
                    C  CB
                    L  RAH,H(MAPPER)
                    L  RAL,L(MAPPER)
RET                 B  3,FAULT
                    L  MAR,MR4,CS+NC
                    D  LOC,LOC,P1
FAULT               L  MR4,X'C3'        SPECIAL LPSW
                    L  IR,MR4,CS+NC     OPCODE -
                    L  MAR,X'80'        DROM GIVES
                    D  LOC,LOC,P1       SPSW AS ADDRESS
```

This must all appear in one ROM page. Otherwise, the branch instructions must all be coded as:

```
C  CB
L  RAH,H(branch address)
C  SB
B  __,branch address
```

Phase zero must reset utility flip-flop.

Figure 35 (continued)

instruction for paging. When the user reference is made to memory the routine MAPPER, shown in Figure 36, is called. MAPPER is also called from phase two and is, therefore, considered a multi-phase routine.

The microcode in phase two performs the operations needed to execute user instructions. To exit phase two another decode instruction is used. This time either phase zero is re-entered or phase three is entered. Phase three is for the support of the console display and controls and for interrupt support. In the normal machine phase two code simply does its job, then exits by the following sequence of micro-instructions:

```
L    MAR,LOC,NC
D    LOC,LOC,P2N
```

In the paging machine, when phase two is entered RR and RX instructions are ready to be emulated. All RS instructions must, however, cause the execution of the code shown in Figure 37 if the utility flip-flop is reset. All of the routines in phase two must execute the code in Figure 38 to exit to phase zero or phase three. This exiting routine might appear only once in the ROM and be branched to as a common routine.

These routines may detect a page fault. A page fault occurs when the actual page specified in the page table at the location indicated by the user page specification is zero and the master mode bit of the current program status word (PSW) is off. When a page fault occurs, the current PSW is saved at location X'84-87', the user address reference causing the fault is saved at X'88-89', and a new PSW is taken from X'8A-8D'. The microcode which executes this response to a page fault is embedded in the previous figures. Specifically, the

This routine must all reside in one page of ROM.

```
MAPPER          C  SB                 INDEX PAGE
                L  AR,MR4,NC          MAP
                N  MAR,X'FC'          USER PAGE IN 4
                S  MR4,MAR,NC+NF      ACTUAL PAGE IN 3
                L  MAR,MAR,SR+NC
                C  MR
                L  AR,MDR,CS+NC       SEE IF IN
                N  MR3,X'FC'          MASTER MODE
                L  AR,PSW,CS+NC       YES TO SKIP
                N  MR2,X'1'           NO - PAGEFAULT
                L  AR,X'0'
                A  MR2,MR2,NC
                B  2,SKIP
                A  MR3,MR3,NC         ACTUAL PAGE ZERO
                B  3,SKIP             NO TO SKIP
                C  CB                 YES
                L  RAH,H(PGFLT)
                L  RAL,L(PGFLT)
SKIP            L  AR,MDR,NC
                N  AR,X'FF'           COUNT ACCESS
                A  AR,X'1'
                L  AR,MDR,NC
                N  MR2,X'FF'
                S  AR,MR2,NC+NF
                A  MDR,MR1,NC+NF
                C  PW
                L  AR,MR4,NC
                A  MR4,MR3,NC
                L  FLG,X'0'
RETURN          C  CB
                L  AR,MR0,CS+NC       RETURN TO
                N  RAH,X'FF'          CALLER VIA
                L  AR,MR0,NC          MR0
                N  RAL,X'FF'
PGFLT           L  MAR,X'80'
                C  MR
                L  MAR,X'84'          SET UP FOR PAGE-
                C  PW                 FAULT
                L  MAR,X'82'
                C  MR
                L  MAR,X'86'
                C  PW
                L  AR,MR4,NC
                N  MR4,X'FC'
                L  MR4,MR4,SR+NC
                L  MDR,MR4,SR+NC
```

Figure 36

Multiple-phase Mapping Routine

```
L  MAR,X'88'
C  PW
L  MAR,X'8A'
C  MR
L  MAR,X'80'
C  PW
L  MAR,X'8C'
C  MR
L  MAR,X'82'
C  PW
L  LOC,MDR,NC
L  FLG,X'3'
B  3,RETURN
```

Figure 36 (continued)

All RS instructions must check the UT flip-flop and branch to this routine if it is reset, with MRO containing the return address.

```
PSLOC          L  AR,LOC,NC+CS
               N  MR4,X'FC'
               S  MR4,MR4,NC
               L  AR,MR4,CS+NC
               S  MR4,X'2'
               L  AR,X'0'
               O  MR4,MR4,NC
               B  3,NOPROB
               L  AR,LOC,NC+CS
               N  MR4,X'FC'
               C  CB
               L  RAH,H(MAPPER)
               L  RAL,L(MAPPER)
NOPROB         L  FLG,X'0'
               L  AR,MRO,CS+NC
               C  CB
               N  RAH,X'FF'
               L  AR,MRO,NC
               N  RAL,X'FF'
```

The phase two code must do the following upon return to the caller:

```
               B  3,OKADD
               C  CB
               L  RAH,H(SPSW)
               L  RAL,L(SPSW)
OKADD          ---------------
                    .
                    .  normal sequence of phase two code
                    .

SPSW           L  MAR,X'82'        SPSW IS REFERENCED
               C  MR               IN PHASE ONE WHEN
               L  PSW,MDR,NC       FAULTS OCCUR IN
               L  MAR,X'84'        THAT PHASE.  DROM
               C  MR               PROVIDES ADDRESS
               L  LOC,MDR,NC       IN THAT CASE.
               L  MAR,LOC,NC
               D  LOC,LOC,P2N
```

Figure 37

Phase Two Coding for Paging and Faults

User (phase two) instructions that set the condition code, must do so in PSW in memory at Page 0, X'80'. All instructions must execute the following sequence of code. This code must be all in one ROM page.

```
EXIT          L  AR,LOC,CS+NC
              N  MR4,X'FC'
              S  MR4,MR4,NC
              B  2,OK
              L  MAR,X'82'
              C  MR
              L  AR,MDR,CS+NC
              N  AR,X'FC'
              A  MR4,X'4'
              L  MDR,MR4,CS+NC
              C  PW
              L  MR0,H(BACK)
              L  AR,MR0,CS+NC
              A  MR0,L(BACK)
              C  CB
              L  RAH,H(MAPPER)
              L  RAL,L(MAPPER)
BACK          B  3,SPSW
              L  LOC,MR4,CS+NC
              L  MAR,LOC,NC
              D  LOC,LOC,P2N
OK            L  MAR,X'82'
              C  MR
              L  AR,MDR,CS+NC
              N  MR4,X'FC'
              L  AR,LOC,CS+NC
              N  MR3,X'FC'
              S  AR,MR3,NC+NF
              A  MR4,MR4,NC+NF
              L  MDR,MR4,CS+NC
              C  PW
              L  MAR,LOC,NC
              D  LOC,LOC,P2N
```

Figure 38

Phase Two Exiting Code for Paged Processor sections labelled SPSW, FAULT, and PGFLT are responsible for implementing the fault trapping mechanism. The amount of expansion required for the ROM routines to implement automatic paging is not small; however, the power of the automatic paging hardware is a significant capability. Execution of each user instruction is slowed considerably; exactly how much depends on two factors. These factors are: (1) where the user instruction is located; that is, whether a page boundary is in the middle of an instruction, immediately after it, or nowhere near it, and (2) whether a page fault occurs during a memory reference. Furthermore, being in master mode can speed up the execution time somewhat since page fault tests are skipped. The results of the timing considerations are illustrated by Figure 39. To summarize this figure, the increase in normal execution time from the normal user instruction set to an automatic paging set is in the range of 8 microseconds to 58.4 microseconds depending on the previously stated conditions. The apparent number of ROM locations added by this paging code is 170 locations, (X'AA'). This is less than one ROM page (a ROM page is X'100' locations). When the total ROM size capability of 1536 locations is considered this amount is relatively insignificant. If another model had the same characteristics as the Model 4 but more read-only memory (4096 locations are addressable by the 12 bit contents of DROM), the 170 locations would positively represent an insignificant cost in space for the increase in user memory that was automatically addressable with the paging property.

RR instruction - no address reference

- page boundary not following

  increased normal time by 8.0 microseconds

- page boundary following (assume no page fault)

  if not in master mode

    increased normal time by 24.4 microseconds

  if in master mode

    increased normal time by 23.3 microseconds

- if address reference occurs (assume no page fault) also add the following times

  if not in master mode

    increased normal time by 40.0 microseconds

  if in master mode

    increased normal time by 37.6 microseconds

The maximum increase in RR instruction execution time assuming no page fault occurs is 40.0 microseconds and the minimum increase in execution time is 8.0 microseconds.

RX instruction

- if no page boundary near and no page fault occurs

  if in master mode

indexed	- increased normal time by 36.6 microseconds
non-indexed	- increased normal time by 33.6 microseconds

  if not in master mode

indexed	- increased normal time by 38.4 microseconds
non-indexed	- increased normal time by 34.0 microseconds

Figure 39

Execution Times

- if page boundary near   in   after

    if in master mode

        indexed   - increased by 52.6 or 55.4 microseconds

        non-indexed - increased by 49.6 or 52.4 microseconds

    if not in master mode

        indexed   - increased by 55.6 or 58.4 microseconds

        non-indexed - increased by 51.2 or 54.0 microseconds

A page boundary may appear either in the middle of an instruction or at the end of the instruction. The maximum increase in RX instruction execution time if no page fault occurs is 58.4 microseconds and the minimum increase in execution time if no page fault occurs is 33.6 microseconds.

<u>RS instruction</u>

- if no page boundary near

    indexed   - increased normal time by 29.8 microseconds

    non-indexed - increased normal time by 32.0 microseconds

- if page boundary near   in   after

    if in master mode

        indexed   - increased by 38.4 or 36.4 microseconds

        non-indexed - increased by 27.6 or 22.0 microseconds

    if not in master mode

        indexed   - increased by 39.6 or 37.6 microseconds

        non-indexed - increased by 28.8 or 23.2 microseconds

A page boundary may appear either in the middle of an instruction or at the end of the instruction. The maximum increase in

Figure 39 (continued)

RS instruction execution time if no page fault occurs is 39.6 microseconds and the minimum increase in execution time if no page fault occurs is 23.2 microseconds.

Figure 39 (continued)

Section 5 - Conclusions

The operating system model in the first chapter assumed that the machine upon which it would be implemented, the target machine, would have certain properties. Some of these properties might not have existed. (Paging hardware, in fact, did not.) If the hardware of the target machine had not been microprogrammable, then some other method of providing the needed properties would have to be chosen. Among these are: (1) changing the system model to no longer require these properties, (2) buying a new computer upon which the desired features already exist, (3) buying special hardware to modify the existing computer, thus creating the desired properties, (4) having a special computer with the required properties designed and built, and (5) simulating the target machine with software.

The paging implementation described in the previous section provided evidence which may be used to evaluate microprogramming as an alternative to the above choices. The two main problems encountered by the paging microcode were concerned with the use of core memory for the CMM and the degradation of execution time for user instructions. The core memory problem could be solved by making available some WCM for use as a scratchpad memory. The speed of execution problem is simply a price that must be paid for utilizing a method of implementation other than the special hardware modification. Assume the sum of the execution times of the micro-instructions necessary to emulate normal user instructions on the firmware system is equal to the execution time of the equivalent user instructions on the hardware system. Then, the firmware implementation could not help but be faster than a software emulation provided that the feature being added was one which effected every user instruction as did the paging feature. As the speed of execution of hardware implemented user instructions increases relative to the firmware implemented ones, the value of microprogramming over software simulation decreases. However, increasing the hardware implemented user instruction execution rates usually implies a rapid increase in CPU cost.

The already established ability of microprogramming to extend and modify instruction sets more easily than in hard-wired systems opens a door to further use of ROM and WCM for the implementation of operating systems. Usually, the major portions of an operating system can be pure procedures. Those parts that are "pure" would lend themselves to implementation in ROM. The routines, or parts of them, that comprise the operating system would then become system primitives. They would be recognized by the emulated system as "user instructions" although the master-slave distinction remains. The effect of this implementation would be similar to having a computer with a hard-wired operating system. Those routines which needed parameters passed to them could be invoked utilizing the address field normally present in a user instruction to point to the data area containing the necessary information. If enough WCM were present, then some system data bases could be maintained in it, with a resulting speed increase.

This compression of system routines into ROM would not have an adverse effect on normal instruction execution time. This is true since new instructions would not need to cause a general modification affecting all instructions. In the INTERDATA Model 4, for example, all such routines would only require the addition of appropriate phase two code to perform the routines. In fact, this firmware system would be faster than the equivalent software on the Model 4, because for each instruction previously executed as software in a routine and now only a part of a primitive, there will be a saving of from 8.0 to 58.4 microseconds. This means that the system would run much faster if implemented in firmware on the Model 4. In general, the utilization of firmware to build primitives can reduce the amount of "decoding" of user instructions that must occur; only one decoding function need be performed. Also, only one memory fetch is required to fetch the primitives instead of many fetches for the many instructions of the routines of a software system. Only one increment of the location counter need occur per primitive executed. These features each contribute to the saving of time over the equivalent software system. All of the hardware multiplexor routines of the model in Chapter 1, described on pages 23 through 28, are good examples of routines that could become system primitives in a firmware implementation.

In addition, all of the memory multiplexor routines and some special data structure manipulation primitives could also be implementable in firmware. If the latter course were chosen exclusively, the user might be allowed to execute some of the primitives to ease his own data manipulation problems.

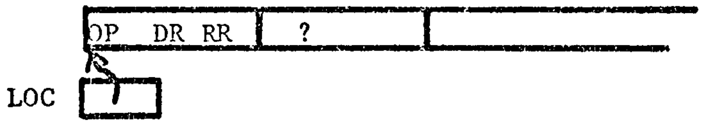
The major problem that occurs in a firmware implementation of an operating system is the loss of ability to provide the user with powerful instruction sets. If the ROM space is at all limited, then only a certain amount of extra space was provided which can be used by the operating system. Implementation beyond that would require the sacrifice of user instruction routines and thus limit the user instruction set even more. There exists some point at which an even trade-off between speed and capability of the system and speed and capability of the user instruction set occurs. Microprogramming does, however, allow this trade-off point to be determined dynamically by the system designer.

These conclusions indicate that the ability to modify a machine after it has left the drawing board is a powerful research and design tool. Although the normal user of a computer need not be concerned with microprogramming, such a powerful capability should not be denied that group of researchers or system designers who are willing to spend the extra time necessary to understand and benefit from its use.

APPENDIXES

Appendix A. Micromachine Phase Structure

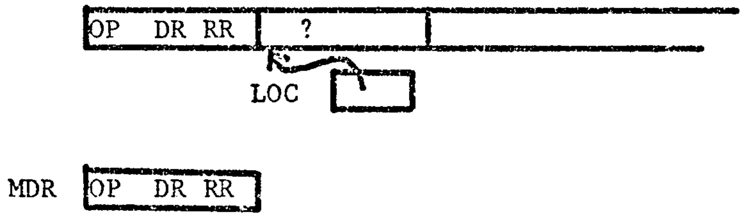
During phase three of a previous instruction



a decode instruction is executed to exit phase three.

```
L      MAR,LOC,NC
D      LOC,LOC,P3
```

After the execution of this instruction:

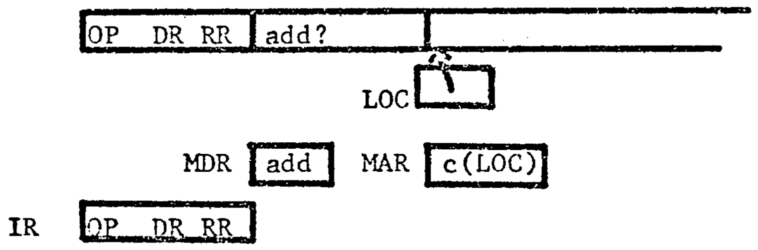


and the phase pointer is zero. The phase zero microsequence appears to be:

```
010  L      IR,MDR,NC
011  L      MAR,LOC,NC
012  D      AR,YD,P0
```

Now, either phase one or phase two is entered, depending on whether the OP in IR is RR, RX, or RS indexed or unindexed.

If OP is not RR or unindexed RS, the status of the hardware is:



If OP is indexed RS, X'004' is jammed into the microlocation counter.

```
004  L     AR,YS,NC
005  A     MDR,MDR,NC+NF
006  D     LOC,LOC,PC
```

and phase two is entered.

If OP is non-indexed RX, X'008' is jammed into the micro-location counter.

```
008  L     MAR,MDR,NC
009  D     LOC,LOC,P1
```

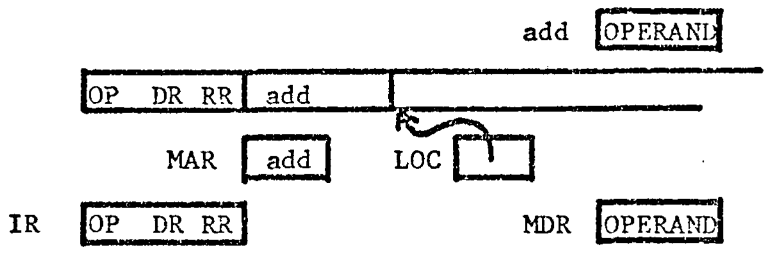
and phase two is entered.

If OP is indexed RX, X'00C' is jammed into the microlocation counter.

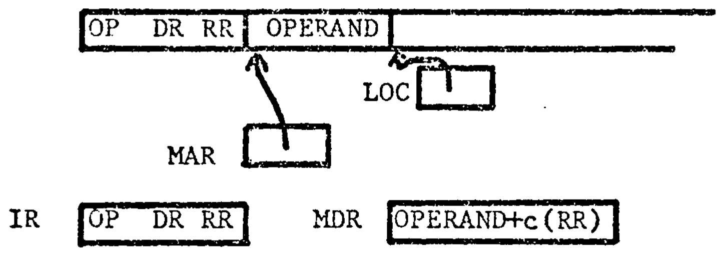
```
00C  L     AR,YS,NC
00D  A     MAR,MDR,NC+NF
00E  D     LOC,LOC,P1
```

and phase two is entered.

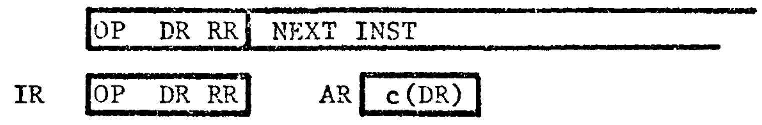
After any of these except the RS case the situation is as follows:



If RS was the case, whether indexed or not, the following conditions apply upon entering phase two.



If RR was the case, then the following conditions apply upon entering phase two.



Appendix B. Model 4 Register Addresses [15]

CODE	DESTINATION	SOURCE
0000	RAH*MR0**	MR0
0001	RAL*MR1**	MR1
0010	YS* MR2**	MR2
0011	MR3	MR3
0100	MR4	MR4
0101	MAR	MAR
0110	LOC	LOC
0111	PSW	PSW
1000	AR	NULL
1001	IR	IR
1010	MDR	MDR
1011	FLR	IR4
1100	CNTR	NULL
1101	IO***	IO***
1110	YD	YD
1111	YDP1	YDP1

* Bank must be reset
** Bank must be set
*** Not a register

Appendix C. Model 4 Instruction Execution Times [15]

Add		800 nsec
Add Immediate		800 nsec
Subtract		800 nsec
Subtract Immediate		800 nsec
Exclusive OR		400 nsec *
Exclusive OR Immediate		400 nsec *
AND		400 nsec *
AND Immediate		400 nsec *
Inclusive OR		400 nsec *
Inclusive OR Immediate		400 nsec *
Load		400 nsec *
Load Immediate		400 nsec *
Load I/O		1200 nsec
Command		400 nsec
Test		400 nsec
Branch on Condition	True:	800 nsec
	False:	400 nsec
Decode		800 nsec

* Exceptions: The instruction takes 800 nsec: if RAL is specified as a Destination Register.

Appendix D. Reproduction of Assembler Output

MODEL 4 MICRO CODE

```
                  * THIS IS A TEST OF THE ROM PROGRAMS FOR
                  * A SUBSET OF THE NORMAL INSTRUCTION SET
                  * CONSISTING ONLY OF THE FOLLOWING:
                  *
                  * LH REG,ADDRESS(INDEX) RX
                  * LHR REG1,REG2 RR
                  * AH REG,ADDRESS(INDEX) RX
                  * AHR REG1,REG2 RR
                  * STH REG,ADDRESS(INDEX) RX
                  * B ADDRESS RX
                  *
                  * AND THE SPECIAL WRITE WITH CONVERSION
                  * INSTRUCTION WHICH HAS THE FORMAT
                  * WWC REG1,REG2 RR
                  *
                  * THE FIRST REG IS NOT USED.
                  *
                  *
                          ORG X'4'
0004  48E0                L   AR,YS,NC          PHASE 1 - RS W/INDX
0005  CAA0                A   MDR,MDR,NC+NF
0006  0661                D   LOC,LOC,PC
0007  4660                L   LOC,LOC,NC        NO-OP
0008  45A0                L   MAR,MDR,NC        PHASE 1 - RX WO/INDX
0009  066B                D   LOC,LOC,P1
000A  4660                L   LOC,LOC,NC        NO-OP
000B  4660                L   LOC,LOC,NC        NO-OP
000C  48E0                L   AR,YS,NC          PHASE 1 - RX W/INDEX
000D  C5A0                A   MAR,MDR,NC+NF
000E  066B                D   LOC,LOC,P1
000F  4660                L   LOC,LOC,NC        NO-OP
0010  49A0                L   IR,MDR,NC         PHASE ZERO
0011  4560                L   MAR,LOC,NC
0012  08EB                D   AR,YD,P0          EXIT TO PHASE 1 OR 2
0013  4660                L   LOC,LOC,NC        NO-OP
                  *
                  * PHASE THREE COMES NEXT
                  *
0014  4560                L   MAR,LOC,NC
0015  066B                D   LOC,LOC,P3        EXIT TO PHASE 0
                          ORG X'01D'
001D  4AE0        LHRR    L   MDR,YS,NC
```

```
001E  4EA0      LHRX    L    YD,MDR,NC
001F  48A0              L    AR,MDR,NC
0020  6AA4              O    MDR,MDR,NC
0021  4560              L    MAR,LOC,NC
0022  066F              D    LOC,LOC,P2J
0023  4AE0      AHRR    L    MDR,YS,NC
0024  CEA5      AHRX    A    YD,MDR,CO
0025  4560              L    MAR,LOC,NC
0026  066B              D    LOC,LOC,P2N
0027  4AE0      STH     L    MDR,YD,NC
0028  3200              C    MW
0029  4560              L    MAR,LOC,NC
002A  066B              D    LOC,LOC,P2N
002B  5302      WWCRR   L    MR3,X'02'
002C  4D31              L    IO,MR3,ADRS
002D  54A8              L    MR4,X'A8'
002E  4D43              L    IO,MR4,CMD
002F  48E0              L    AR,YS,NC
0030  D430              A    MR4,X'30'
0031  4D31              L    IO,MR3,ADRS
0032  4D42              L    IO,MR4,DA
0033  4560              L    MAR,LOC,NC
0034  066B              D    LOC,LOC,P2N
0035  4650      BRX     L    LOC,MAR,NC
0036  066B              D    LOC,LOC,P2N
                             END


                             SYMBOL TABLE
AHRR    0023        AHRX    0024        BRX     0035
LHRR    001D        LHRX    001E        STH     0027
WWCRR   002B
```

REFERENCES

REFERENCES

Abbreviations used in the references:

AFIPS	American Federation of Information Processing Societies
SJCC	Spring Joint Computer Conference
ACM	Association for Computing Machinery

References, in order cited:

[1] Wilkes, M.V., "The Best Way to Design an Automatic Calculating Machine," Manchester U. Computer Inaugural Conf., 1951, p. 16.

[2] Opler, A., "Fourth Generation Software," Datamation, 1, 13 (1967), p. 22.

[3] Wilkes, M.V., "The Growth of Interest in Microprogramming: A Literature Survey," Computing Surveys, 1, 3 (1969), p. 139.

[4] Denning, Peter J., "The Working Set Model for Program Behavior," Communications of the ACM, 5, 11 (1968), p. 323.

[5] Randell, B., and Kuehner, C.J., "Dynamic Storage Allocation Systems," Communications of the ACM, 5, 11 (1968), p. 297.

[6] Lampson, Butler W., "A Scheduling Philosophy for Multiprocessing Systems," Communications of the ACM, 5, 11 (1968), p. 347.

[7] Graham, Robert M., "Protection in an Information Processing Utility," Communications of the ACM, 5, 11 (1968), p. 365.

[8] Saltzer, J.H., Traffic Control in a Multiplexed Computer System, MAC-TR-30 (thesis), M.I.T., Cambridge, Massachusetts, July, 1966.

[9] Bensoussan, A., Clingen, C.T., and Daley, R.C., "The MULTICS Virtual Memory," The Second Symposium on Operating Systems Principles, (October, 1969), Princeton University Press, p. 30.

[10] Motobayashi, S., Masuda, T., and Takahashi, N., "The HITAC5020 Time Sharing System," Proceedings of the 24th National Conference of the ACM, (1969), ACM Publications, p. 69.

[11] Bruffey, B., Bryan, E., Doeppel, B., and Smith, J., "Universal Time-sharing System Functional Specifications." Unpublished design specification, September 20, 1968.

[12] Daley, Robert C., and Dennis, Jack B., "Virtual Memory, Processes, and Sharing in MULTICS," Communications of the ACM, 5, 11 (1968), p. 306.

[13] Corbató, F.J., Merwin-Daggett, Marjorie, and Daley, R.C., "An Experimental Time-Sharing System," AFIPS Conf. Proc. 21 (1962 SJCC), National Press, Palo Alto, 1962, pp. 335-344.

[14] Rosin, Robert F., "Contemporary Concepts of Microprogramming and Emulation," Computing Surveys, 1, 4 (1969), p. 135.

[15] Model 4 Micro-instruction Reference Manual, publication number 29-032R01, INTERDATA, INC., 1968, p. 1.

[16] Reference Manual, publication number 29-004R02, INTERDATA, INC., 1969.

[17] IBM System/360 Principles of Operation, form A22-6821-6, International Business Machines Corp., 1967.