

HIMICS: A VIRTUAL MEMORY ENVIRONMENT FOR MINI-COMPUTERS AND A
DESCRIPTION OF ITS LEVEL 2 PROCESSOR

by

ARLAN E. BENTZ

B.S., Kansas State University, 1968

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

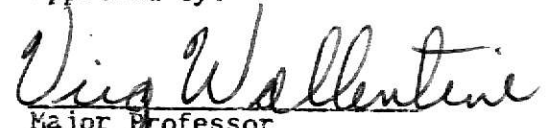
Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1975

Approved by:


Major Professor

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH THE ORIGINAL
PRINTING BEING
SKEWED
DIFFERANTLY FROM
THE TOP OF THE
PAGE TO THE
BOTTOM.**

**THIS IS AS RECEIVED
FROM THE
CUSTOMER.**

LD
2668
R4
1975
B45
C.2
Document

TABLE OF CONTENTS

SECTION NAME	CHAPTER ONE	PAGE
1.1	INTRODUCTION	1
1.2	TECHNIQUES FOR RECURRENT USE OF MEMORY	1
1.2.1	OVERLAY STRUCTURES	1
1.2.2	VIRTUAL MEMORY	2
1.2.2.1	VIRTUAL MEMORY TECHNIQUES	4
1.2.2.2	ADVANTAGES OF VIRTUAL MEMORY	7
1.3	SUPPORT FOR EMULATORS	7
1.4	INTER-EMULATOR COMMUNICATION	8
1.5	MANAGEABLE SOFTWARE	8
1.6	INSTRUMENTATION	9
1.6.1	RECORDED COUNTS	9
1.6.2	WORKING SET OPTIONS	10
1.7	OVERVIEW OF THE SYSTEM	11
1.7.1	ADVANTAGES OF THE SYSTEM	13
1.7.2	EXPLANATION OF LEVELS	15
1.8	IMPLEMENTATION	16
1.8.1	HARDWARE ALLOCATION	17
1.8.2	MEMORY LEVELS	17
1.8.3	LOCATION OF SOFTWARE	22
1.9	SUMMARY	23
1.10	INTRODUCTORY DESCRIPTION OF REMAINING CHAPTERS	24

CHAPTER TWO

2.1	INTRODUCTION	26
2.2	PAGE MANAGEMENT	26
2.2.1	THRASHING IN PAGED MEMORY SYSTEMS	26
2.2.2	PAGING OPTIONS	27
2.2.3	WORKING SET SIZE	30
2.2.4	PAGING TRANSFER FLOW PATTERNS	30
2.2.4.1	TRANSFER FLOW PATTERN 1	34
2.2.4.2	TRANSFER FLOW PATTERN 2	34
2.2.4.3	TRANSFER FLOW PATTERN 3	35
2.2.4.4	TRANSFER FLOW PATTERN 4	36

SECTION NAME	PAGE
2.2.5 PAGING ALGORITHMS USED IN SYSTEM	37
2.2.5.1 PAGING ALGORITHM DISCUSSION FOR OPTION 1	39
2.2.5.2 PAGING ALGORITHM DISCUSSION FOR OPTION 2	42
2.2.5.3 PAGING ALGORITHM DISCUSSION FOR OPTION 3	44
2.2.6 THRASHING AS RELATED TO PAGING OPTIONS	45
2.3 I/O FOR SYSTEM	46
2.3.1 SPOOLING OF I/O	46
2.3.2 PROGRAM REQUESTED I/O	46
2.4 FILE MANAGEMENT	47
2.4.1 DATA BASES FOR FILE MANAGEMENT	48

CHAPTER THREE

3.1 INTRODUCTION	55
3.2 ALGORITHM 1: MAIN DRIVER FOR NOVA ROUTINES	55
3.3 ALGORITHM 2: GENERATES THE SYSTEM	61
3.4 ALGORITHM 3: MAIN INPUT/OUTPUT DRIVER	61
3.5 ALGORITHM 4: INPUT DRIVER	68
3.6 ALGORITHM 5: OUTPUT DRIVER	70
3.7 ALGORITHM 6: TRANSFERS DATA TO INPUT ADDRESS	71
3.8 ALGORITHM 7: TRANSFERS DATA TO OUTPUT FILE	74
3.9 ALGORITHM 8: TRANSLATES NOVA I/O ERROR CODE	74
3.10 ALGORITHM 9: PAGE OPTION 1 MAIN ROUTINE	76
3.11 ALGORITHM 10: PAGE OPTION 1 INITIAL ROUTINE	84
3.12 ALGORITHM 11: PAGE OPTION 2 MAIN ROUTINE	85
3.13 ALGORITHM 12: PAGE OPTION 2 INITIAL ROUTINE	88
3.14 ALGORITHM 13: PAGE OPTION 3 MAIN ROUTINE	90
3.15 ALGORITHM 14: PAGE OPTION 3 INITIAL ROUTINE	90

SECTION NAME	PAGE
3.16 ALGORITHM 15: RETRIEVE PAGE	91
3.17 ALGORITHM 16: END OF JOB	92
3.18 ALGORITHM 17: LEVEL 3 MEMORY TO LEVEL 2 MEMORY	92
3.19 ALGORITHM 18: LEVEL 3 MEMORY TO LEVEL 1 MEMORY	93
3.20 ALGORITHM 19: LEVEL 2 MEMORY TO LEVEL 3 MEMORY	93
3.21 ALGORITHM 20: LEVEL 1 MEMORY TO LEVEL 2 MEMORY UNDER OPTIONS 1 AND 2	93
3.22 ALGORITHM 21: LEVEL 1 MEMORY TO LEVEL 2 MEMORY UNDER OPTION 3	94
3.23 ALGORITHM 22: LEVEL 1 MEMORY TO LEVEL 3 MEMORY	94
3.24 ALGORITHM 23: LEVEL 2 MEMORY TO LEVEL 1 MEMORY	95
3.25 ALGORITHM 24: INPUT SPOOLING	95
3.26 ALGORITHM 25: JOB QUEUE SEARCH	98
3.27 ALGORITHM 26: OBJECT DECK FILE NAME	98
3.28 ALGORITHM 27: RETRIEVE OBJECT DECK FILE NAME	98
3.29 ALGORITHM 28: CREATE OUTPUT FILE NAME	99
3.30 ALGORITHM 29: LIST STACK DEPTH COUNTS FOR PAGE OPTIONS 1 OR 3	99
3.31 ALGORITHM 30: LIST STACK DEPTH COUNTS FOR PAGE OPTION 2	100

CHAPTER FOUR

4.1 INTRODUCTION	101
4.2 SYSTEM MODIFICATION	101
4.2.1 I/O FILES	101
4.2.2 ROLLIN AND ROLLOUT	103
4.2.3 MULTI-TASKING	104
4.2.4 PRIORITY	104
4.2.5 PARAMETER PASSING FOR SUBROUTINE INDEPENDENCY	104

SECTION NAME	PAGE
4.3 TESTING OF ALGORITHMS	105
4.4 CONCLUSION	110
4.4.1 THEORETIC TIME ADVANTAGE	110
4.4.2 SUGGESTED SYSTEM LOADS FOR SYSTEM PERFORMANCE EVALUATION	115
4.4.3 COST OF MEMORY	116

APPENDICES

APPENDIX A: ALGORITHMS	117
APPENDIX B: SYSTEM CONFIGURATION FOR PAGE OPTION 1	160
APPENDIX C: SYSTEM CONFIGURATION FOR PAGE OPTION 2	161
APPENDIX D: SYSTEM CONFIGURATION FOR PAGE OPTION 3	162
APPENDIX E: SUBROUTINE CALLING ORDER	163

ILLUSTRATIONS

	PAGE
FIGURE 1-1 OVERLAY STRUCTURE	3
FIGURE 1-2 VIRTUAL MEMORY STRUCTURE	5
FIGURE 1-3 FIXED WORKING SET OPTIONS	12
FIGURE 1-4 HIERARCHICAL STRUCTURE	14
FIGURE 1-5 HARDWARE VIEW OF HIERARCHICAL STRUCTURE	18
FIGURE 1-6 ALTERNATE VIEW OF SYSTEM LEVELS	19
FIGURE 1-7 LEVELS OF MEMORY IN THE SYSTEM	21
FIGURE 2-1 LEVEL 2 MEMORY LAYOUT FOR VARIOUS PAGE OPTIONS	29
FIGURE 2-2 WORKING SET SIZE	31
FIGURE 2-3 TRANSFER FLOW PATTERN 1	32
FIGURE 2-4 TRANSFER FLOW PATTERN 2	32
FIGURE 2-5 TRANSFER FLOW PATTERN 3	33
FIGURE 2-6 TRANSFER FLOW PATTERN 4	33
FIGURE 2-7 MEMORY DOMINATION UNDER THE COMPETITIVE VARIABLE WORKING SET SIZE OPTION	43
FIGURE 2-8 SPOOL TABLE	49
FIGURE 2-9 DISK USAGE SECTOR TABLE (DUST)	49
FIGURE 2-10 FILE MANAGEMENT TABLES	51
FIGURE 2-11 LOCATING REQUESTED PAGES IN THE LEVEL 2 PROCESSOR	53
FIGURE 3-1 JOB PAGE CONTROL BLOCK	57
FIGURE 3-2A USER PARAMETER BLOCK	63
FIGURE 3-2B SYSTEM GENERATED PARAMETER BLOCK	63
FIGURE 3-3 INTERDATA FUNCTION CODES FOR I/O	64
FIGURE 3-4 INTERDATA I/O ERROR CODES	65
FIGURE 3-5 EXAMPLE OF I/O SPLIT ACROSS PAGE BOUNDARIES	69
FIGURE 3-6 EXAMPLE OF PAGE MAPPING (512 BYTES VS 256 WORDS)	73
FIGURE 3-7 CONTIGUOUSLY ORGANIZED FILES	75
FIGURE 3-8 NOVA I/O ERROR CODES	77
FIGURE 3-9 STACK DEPTH COUNTS	81
FIGURE 3-10 EXTENDED PAGE FAULT TABLE	89
FIGURE 3-11 SEQUENTIALLY ORGANIZED FILES	97
FIGURE 4-1 EXAMPLE OF ALGORITHM TESTING	107
	108
	109
	111
FIGURE 4-2 MINIMUM PAGE FAULT TIME FOR SINGLE CPU	114

The HIMICS system is a hierarchical virtual memory system for a hierarchy of interconnected mini-computers. This paper describes the design of the software system. The software system design in this paper is a hierarchical design with two major processor levels. An overall description of both processors is given and then a detailed description of its level 2 processor is presented. The detailed description includes the algorithms, written in a dialect of PL/1, along with a written description of them. The HIMICS system will provide a virtual memory system for a network of mini-computers and also allow the emulation of high level languages. The implementation of this system should result in an increase of processor efficiency and system throughput for the mini-computers involved in the network. The paper is concluded with a dialectic comparison of a single processor system versus a multi-processor system.

CHAPTER ONE

1.1 INTRODUCTION

In this paper we propose a design for a hierarchical mini-computer system called HIMICS (Hierarchical multi-tasking Mini-computer Computer System). The system is designed with five major objectives in mind. These objectives are:

- (1) To provide virtual memory capability.
- (2) To provide support for emulators.
- (3) To provide inter-emulator communication.
- (4) To provide manageable software.
- (5) To provide sufficient instrumentation and monitor capabilities in order to encourage meaningful system evaluations and comparisons.

A general discussion of each of these objectives will be given before we present the actual design of the system.

1.2 TECHNIQUES FOR RECURRENT USE OF MEMORY

There are several techniques that are commonly used in modern day computers to execute programs which have a larger address space than the primary memory available to them. Two of the more commonly used techniques are overlay structures (1,6,12) and virtual memory (2,12).

1.2.1 OVERLAY STRUCTURES

With overlay structures, segments of the program are kept on secondary storage and brought into main memory in an hierarchical sequence as they are needed. Pre-specified segments

may be overwritten by incoming segments. This is illustrated in Figure 1-1. Here the user has a 520K program to be run in a 320K address space. Segments A, B and C are first loaded and execution begins. As soon as segment B is no longer needed, segments D and E can be overwritten in B's address space. The same process happens when C and E are no longer needed. They can be overwritten by F.

From this illustration it is apparent that the user must have a knowledge as to what segments are to be overwritten. It is the user's responsibility to issue orders for the overlay to occur. This is the major disadvantage of the overlay technique. The second technique, virtual memory, does not require the user to have this additional knowledge.

1.2.2 VIRTUAL MEMORY

The key to virtual memory relies on the fact that, for an instruction in a program to be executed, only the instruction and the data that it operates on need be in primary memory. From the instruction's point of view the rest of the program may be located on any level of memory. This removes the requirement that the job's entire address space be in physical memory at once. Because this physical restraint is removed, the operating program has the illusion that it has an extremely large memory, thus the term "virtual memory". Since a job's entire address space need not all be in primary memory at once,

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

Overlay Structure

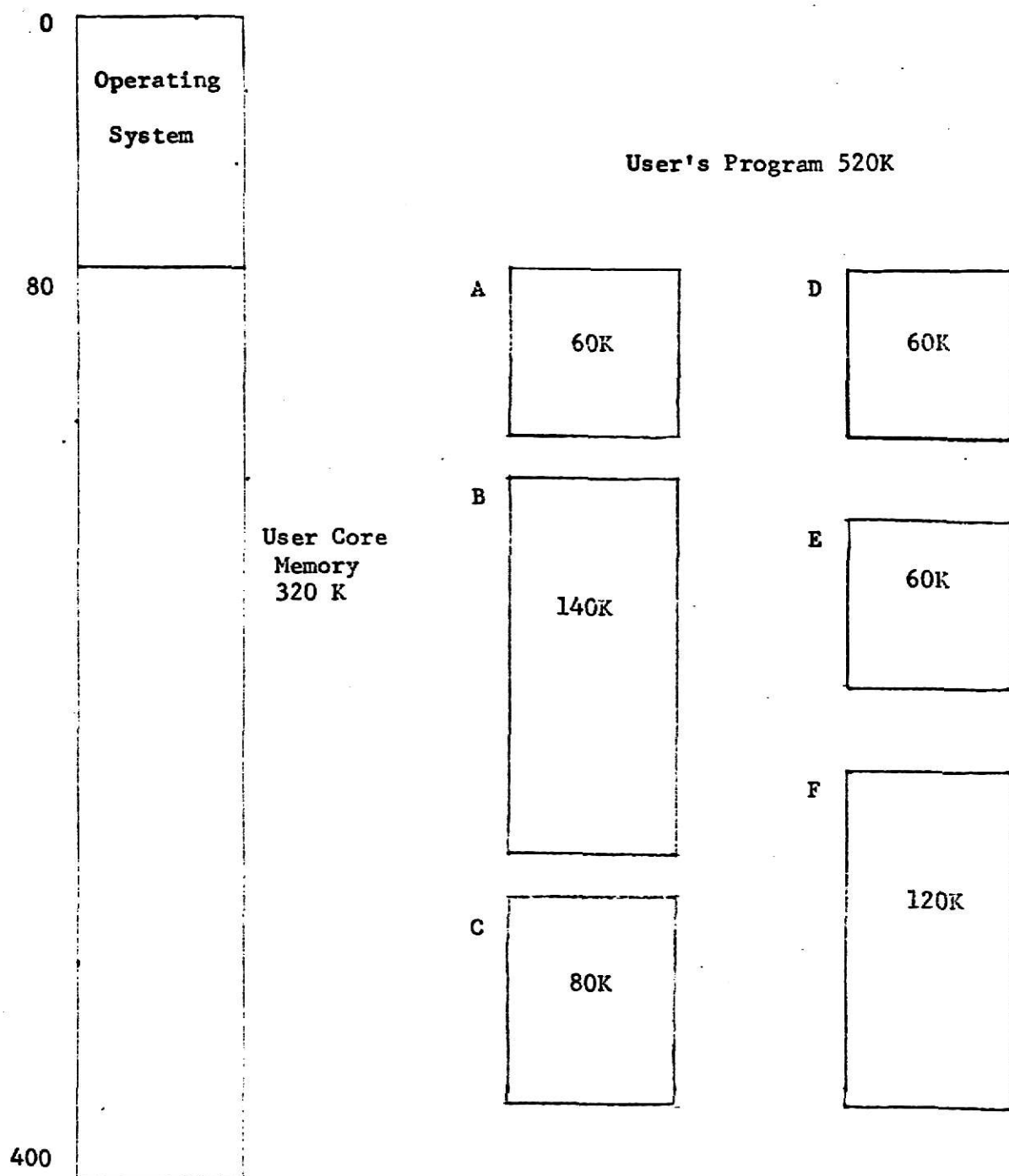


Figure 1-1

the sum of the address spaces of the jobs being multiprogrammed is permitted to exceed the physical size of main memory. This is illustrated in Figure 1-2. All three jobs are being executed in a physical memory space of 280K. The total sum of all three jobs is 460K.

The major constraint as to the size of the virtual space is limited by the hardware configuration. The hardware limits the number of addressable cells. The limiting factor is the number of bits used for an address. For example, if the hardware allows 8 bits for an address, then there are 256 addressable cells. The addresses would range from 0 to 255. This limits the virtual memory to this same size of space. The virtual space is usually considerably larger than the available primary memory of the machine.

1.2.2.1 VIRTUAL MEMORY TECHNIQUES

Two major virtual memory techniques are, demand-paged memory management (1,7) and segmented memory management (1). As was stated previously, virtual memory requires that the instruction and data to be operated on be located in primary memory. If this were to be done one instruction at a time it would be too time consuming. Instead they are retrieved in sections upon demand. If the sections are all equally divided into the same length, they are individually referred to as a page. This is the origin of the term "demand-paged" memory.

Virtual Memory Structure

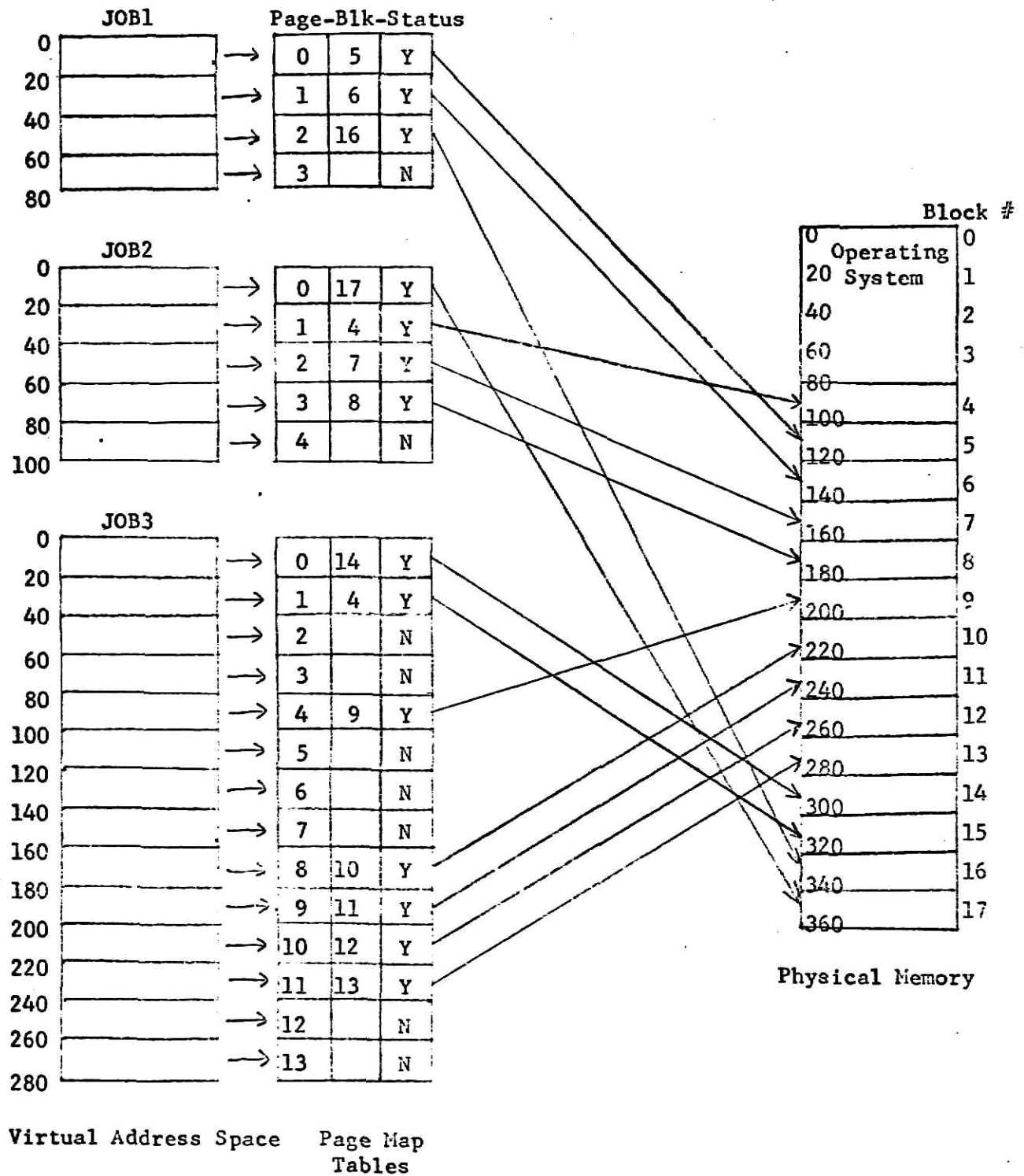


Figure 1-2

If the sections are unequal in length they are called segments, hence "segmented" memory. This paper will only be concerned with the former, demand-paged memory.

At the start of execution of a user's program, the first page is brought into primary memory. This is done by the virtual memory module, which will be explained in section 1.7.2. As each instruction is executed, the virtual memory module checks to make sure that all the address space referenced is in primary memory. If the address space is not in primary memory an interrupt, called a page fault (2), is generated. The operating system then processes this interrupt. This is done by loading the required page into primary memory. The process is then restarted from the point of the interrupt. Each additional required page is brought into primary memory upon request.

This can obviously lead to the point where primary memory is full when a new page is being requested by the virtual memory module to be brought into primary memory. To alleviate this situation page replacement (1) is necessary. This consists of removing from primary memory a page that does not have a high probability of being referenced in the near future. This page is then placed in secondary memory while the newly requested page is moved into primary memory.

1.2.2.2 ADVANTAGE OF VIRTUAL MEMORY

Virtual memory is commonly used today by many large computers.

There are many advantages to be gained by using virtual memory.

Some of these advantages include:

- (1) Increase in the number of programs that can be multiprogrammed in a system.
- (2) Capability of running a program whose address space exceeds the primary memory space currently available, if less than the maximum addressable memory.
- (3) Makes programs more portable from large machines to small machines.
- (4) Helps eliminate fragmentation of dynamic storage allocation.

These are especially appealing to mini-computers since mini-computers by nature have a smaller primary memory space.

1.3 SUPPORT FOR EMULATORS

For use in this paper, we will define an emulator to be a firmware interpreter. This interpreter will convert a user program from the original language, instruction by instruction, into the desired computer actions. An emulator written for this system must be aware of the manner in which to access the virtual addressing system of the host processor. An emulator will not create a machine language program. Instead, it will in effect execute a small microprogram for each instruction of the original language. The result of this activity will be the execution of the original instruction.

The host machine's user assembler language itself will be emulated by HIMICS to allow virtual memory capabilities. HIMICS will

also allow "high-level" Languages to be emulated. Languages such as PL/1, APL, and COBOL are likely candidates for emulation. These languages will require a large amount of space for their emulators, but due to the virtual storage capabilities of the system, this space requirement does not present a problem.

1.4 INTER-EMULATOR COMMUNICATION

In many programming situations, it is desirable to use different languages for different modules of the program. For example, one might want the processing portion of his algorithm to be coded in an assembler language, and the input/output sections written in a high-level language. Such process linkage will be allowed in this system. Interprocess communication will also be allowed in this system due to the capabilities of the host machine's operating system. A task or emulated process can start another task executing. After starting another task, the calling task may wait until the named task terminates. The calling task on the other hand may also continue processing and test for the called task's completion when necessary. A task can also cancel another task which is executing. This kind of communication is solely dependent upon the functions of the host operating system.

1.5 MANAGEABLE SOFTWARE

The key to manageable software is to keep it simple. This can be accomplished by using a structured design (3). In this

approach a complex system is divided into small independent modules. This allows one to comprehend each module without keeping the details of the entire system in mind. Furthermore, modifications to the system are simplified since a module can be changed or added without affecting other modules.

A part of this structured design is provided by the operating systems of the host processors. Even if the operating systems have to be modified to run in a virtual memory paging environment, it is worth the trade-off. These modules are not only structured, they will be almost error free from the start, and thus more manageable. Obviously it would require a great deal of time to produce the equivalent modules from scratch.

1.6 INSTRUMENTATION

1.6.1 RECORDED COUNTS

In order to evaluate a system's efficiency it is necessary to employ techniques to record the specific actions taken by the system. An obvious method of instrumentation is to keep a count of how many times a pre-specified event occurs. By knowing the system input, and the actions caused, it is possible to evaluate the system.

There are two counts which must be taken for every instruction. The first is a count for each unique operation code. When evaluating the system, the frequency of execution of each kind of instruction is essential. The second count records the number of times each

page is referenced. This will be used to evaluate the performance of the system under different paging options.

1.6.2 WORKING SET OPTIONS

Built into the system at system generation time is the ability to perform paging under one of three options. These three options have been chosen to yield different working set (1,8,9) sizes in order to evaluate the system under various work loads for maximum efficiency. In this paper, working set size refers to the number of pages contained in Level 1 and Level 2 memory. This composes a collection of the program's most recently used pages (11). The terms Level 1 memory and Level 2 memory will be explained in detail in section 1.8.2 of this chapter. The first option operates under a non-competitive fixed partitioned working set size. The second option will allow the system to operate with a competitive variable working set size based on a local paging rate. The third option, which operates on a competitive variable working set size also, is based on a global scale and allocates secondary memory using the LRU stack (1,10) principle.

Under option one, the total amount of secondary memory will be divided by the total number of jobs allowed to be multi-programmed. This will be set at system generation time. Each job will then have a fixed working set the same size as any

other job. For example, if there are 180 page frames and 3 jobs, each job will have 60 page frames for its use. (See Figure 1-3A). Under option two the size of the working set for a job will be adjusted to approach maximum efficiency as best as can be determined. This will be based on a competitive paging ratio computed on a per-job paging rate over a given time period. Jobs having high paging rates tend to increase their working set size while low paging jobs decrease their working set size.

Option three will take the entire working set available and let all jobs have the space needed on a first come first use basis. This treats the Level 2 memory on a global basis, whereas under option one it was treated on a per-job basis or local level. This will allow, for example, two jobs, job 1 needing 20 page frames and job 2 needing 120 page frames, to be completely contained in Level 2 memory at once. (See Figure 1-3B). Under option one above, the fixed size per job, job 2 could only have 60 of its 120 pages in Level 2 memory at once.

Obviously in order to tell which of the above three methods is the best, monitoring of the different options is necessary. This will include a count of the number of page faults occurring out of each memory level for each job under the given option. There will also be an option to turn monitoring on or off.

1.7 OVERVIEW OF THE SYSTEM

The HIMICS system may be viewed as a hierarchical structure (4,11). A structure of this nature consists of modules located

Fixed Working Set Options .

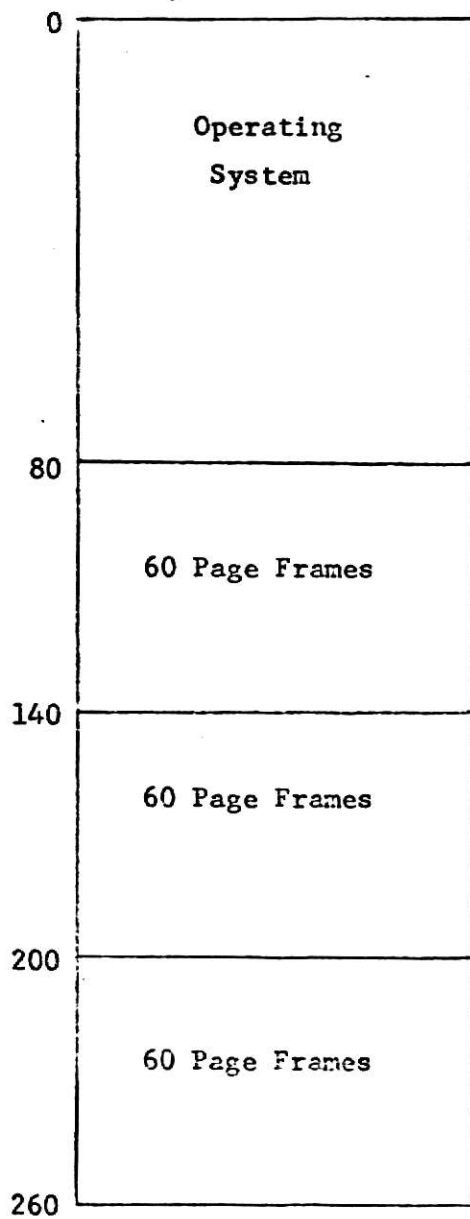


Figure 1-3A

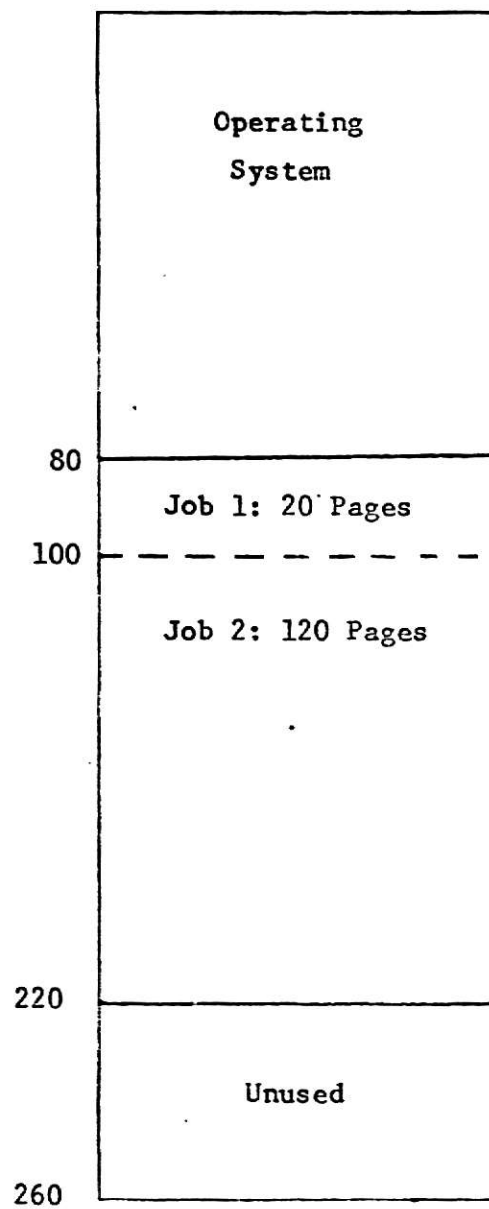


Figure 1-3B

Figure 1-3

on different hierarchical levels (See Figure 1-4). This type of structure is called "layered insensitivity" (4,11). The levels are insensitive because each level is allowed to call upon the services of levels immediately above or below it in the structure, but not those levels farther than one level away. This means that each level is not concerned about how or where things are done in the levels above it or subordinate to it, and treats them all as one level. Each level may be referenced by the level above or below it in the hierarchy, but no level may be dependent on a level which is not a logically sequential level in the hierarchy structure. For example, level 4 of the HIMICS system will interface with the file management system level 5, and virtual storage management level 3, but level 4 may not call upon levels 1, 2, or 6.

1.7.1 ADVANTAGES OF SYSTEM DESIGN

There are several advantages to this type of design.

- (1) The system is easier to understand.
- (2) Each module is easier to implement.
- (3) The verification of the entire system is accomplished by verifying each individual level in a bottom-up fashion.
- (4) Modification of the system is simplified.
- (5) The software system is relatively portable (i.e. interfacing with different hardware requires only the lowest level of the system to be compatible, and the upper levels do not require modification).

Hierarchical Structure

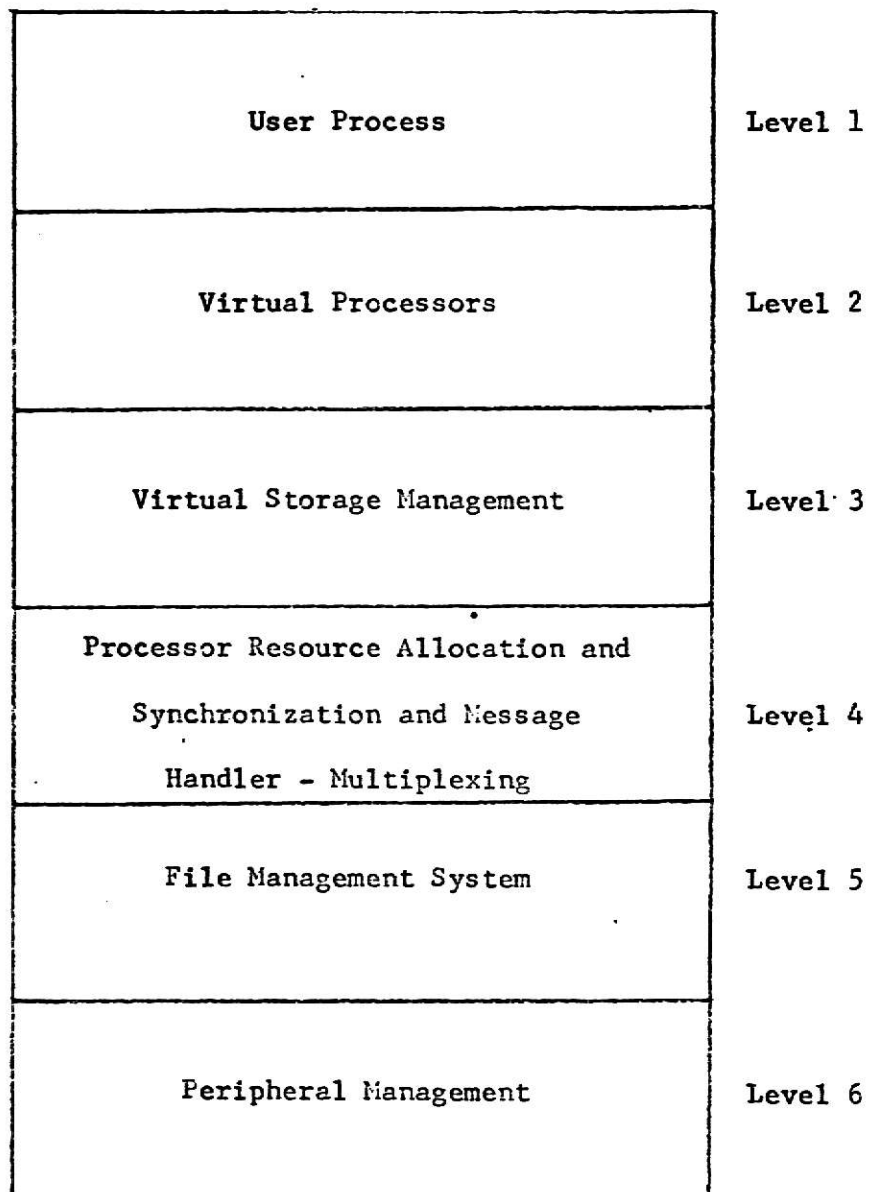


Figure 1-4

1.7.2 EXPLANATION OF LEVELS

A short explanation of what is contained in each module follows.

Level 1 contains the user processes which are interpreted and executed in the primary memory of the host machine. These user programs may be written in any language supported by an emulator on the HIMICS system.

Contained in Level 2 are the virtual processors. This is the system of emulators which execute one instruction of the user's program at a time. Before each instruction is executed the current real address of the virtually addressed operands must be retrieved. Therefore all memory references must be detected and sent to level 3 to be converted before the emulation of the instruction may occur.

Level 3 contains the virtual storage management system. This system must detect any page faults which are generated from the virtual addresses of the instruction's operands. The virtual addresses of the operands must be converted to real machine addresses. This of course requires the page to be located in primary memory. This management system must interact with the file management system through the message handler in level 4 to retrieve pages to primary memory.

The message handler and resource allocation systems in level 4 are intermeshed deeply with the operating system of its host machine. The message handler is responsible for the generation

and control of all information transfer from the file management system. This communication will consist of I/O messages to and from level 6, page requests from secondary memory in level 5, and the current state of the system (i.e. "request page", "page being transferred"). This message exchange coordinates the activities of levels 5 and 6 with the upper 4 levels of the system.

The file management system will reside at level 5. It will handle all page requests between secondary and primary storage. All input and output messages will be processed by the file management system upon request. Included in the file management module will be tables which contain the current location of each job's pages. These tables are initialized when the job is started, and are updated as pages are moved from one level of memory to another. Level 4 will send page requests and I/O messages through the message handler requesting pages to satisfy page faults and I/O requests when needed.

Level 6 is the peripheral management module. This module will handle the interface with all peripheral devices connected to the system. This may include printers, card readers, teletypes, display terminals, or whatever hardware is available to interface with the system.

1.8 IMPLEMENTATION

We have just presented a machine independent description of the virtual addressing system for a mini-computer system. Now a

more detailed description of the implementation of the system at Kansas State University will be presented.

1.8.1 HARDWARE ALLOCATION

The layered insensitivity graph in Figure 1-5 makes the allocation of duties to actual hardware transparent. As can be seen, the upper four modules will be located in an Interdata 85 computer. The lower two levels will be located in a Nova computer.

The upper four levels are located in the Interdata machine because of its greater processing speeds. The extensive software required by the HIMICS system for each user instruction requires a fast processor. The Interdata cycle time of 270 nanoseconds meets these general speed requirements. The Nova machine is used as a peripheral processor. This processor will have more time to perform its duties, and yet removes a great processing overhead from the Interdata. This setup lets each machine do what it does best, and allows a more efficient and faster system. Using two CPU's in effect allows parallel processing. Real I/O may be supervised by the Nova while the Interdata is processing a user's program. Another view of the system design is given in Figure 1-6. The system shown is based on a multiprogramming environment of three users.

1.8.2 MEMORY LEVELS

It is apparent from Figure 1-6 that the use of two separate CPU's primary memory, and disk memory creates three levels of

Hardware View of Hierarchical Structure

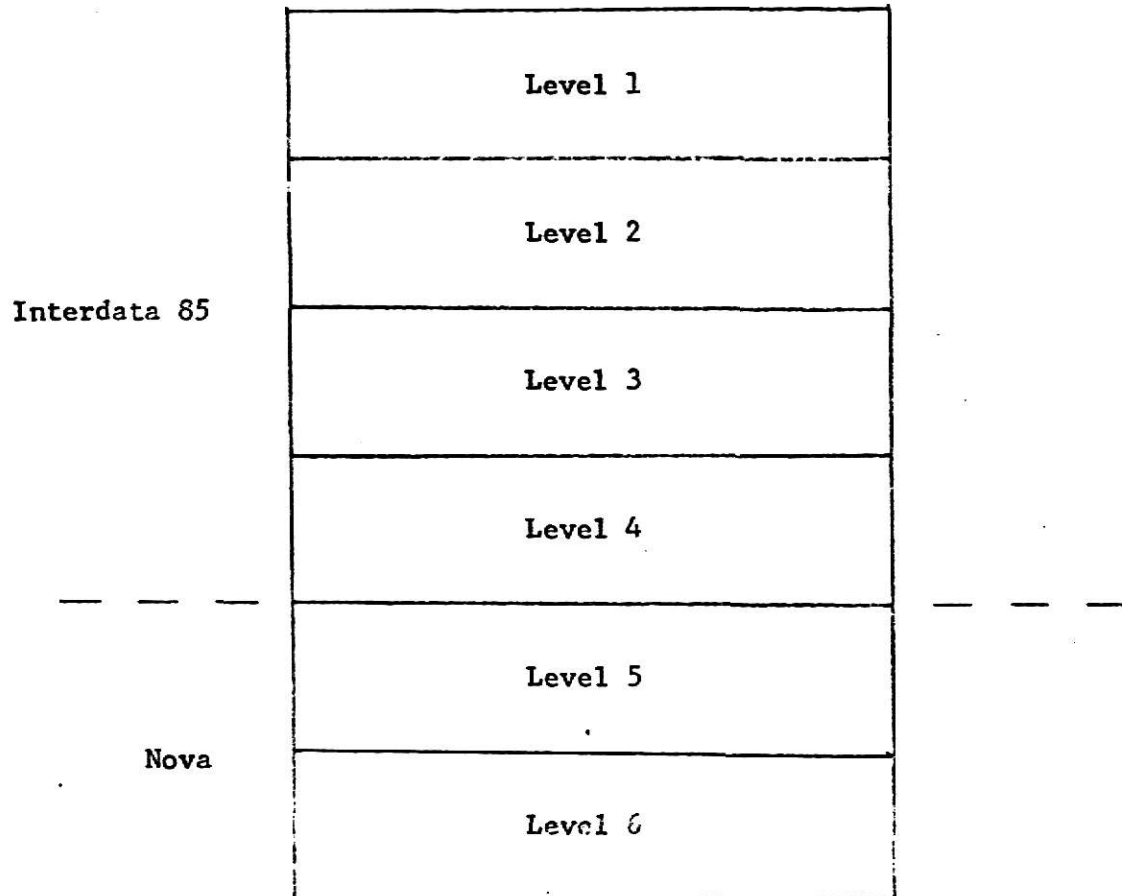


Figure 1-5

Alternate View of System Levels

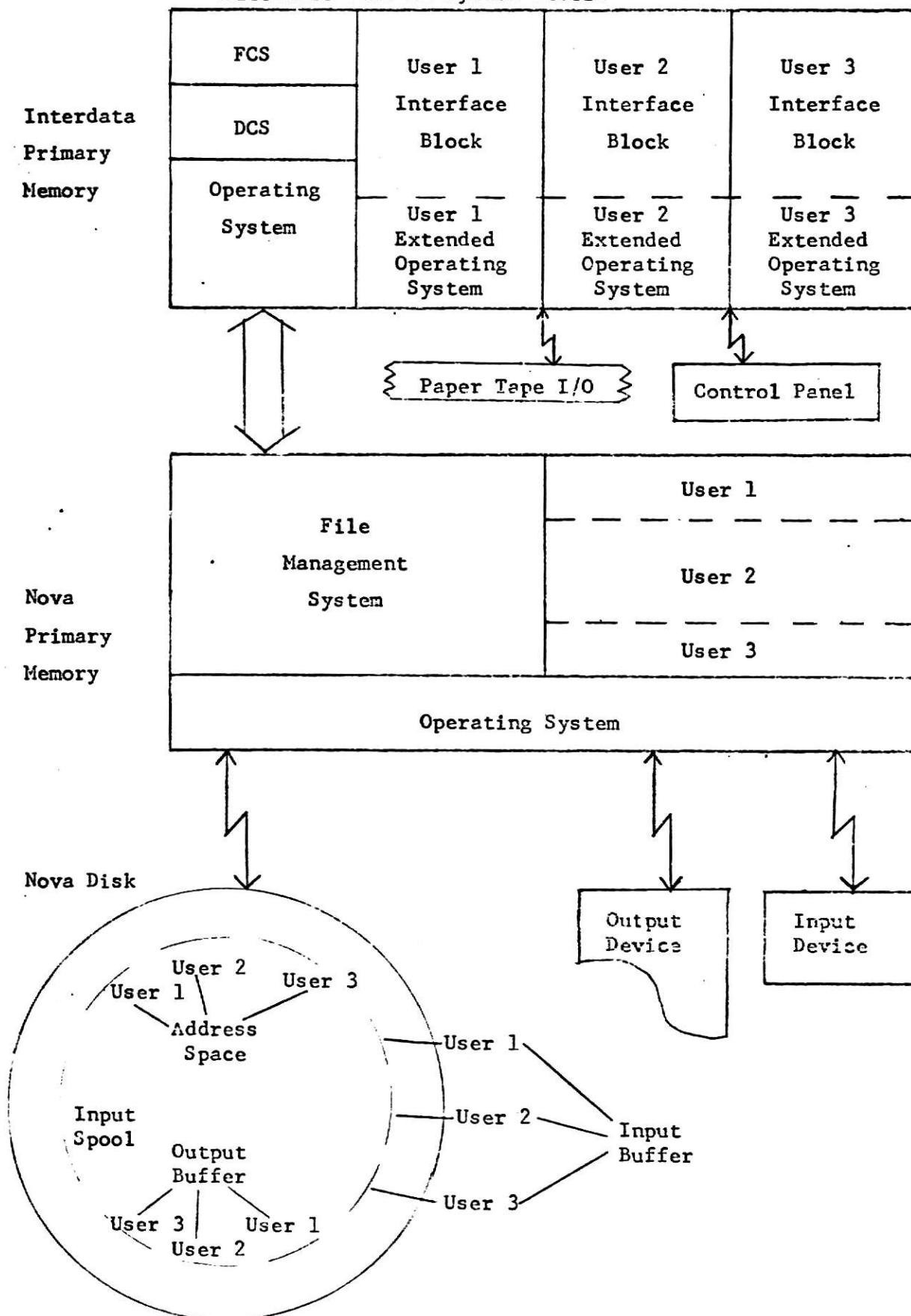


Figure 1-6

memory. These three levels will be referred to as Level 1, Level 2 and Level 3 memory in the remainder of this paper (see Figure 1-7). The Nova disk is the lowest level of memory or Level 3 memory. Each user's program upon entry to the system is spooled to an input file in this third level of memory by the Nova. When the user's job is set to running, the program is put into his address space, also located on Nova disk, Level 3 memory. The data input is stored in his input file. Any output generated by his program will be spooled onto his output file for printing when his job terminates.

Nova primary is the second level of memory for this system. The management system which controls all page traffic in the Nova is located here. This file system receives page traffic from the Interdata, and using its own paging algorithm, rearranges the user's pages in the extended page space in the Nova's Level 2 memory. If a page is being transferred to Level 3 memory (i.e. paged out of Level 2 memory), it is copied back to the address space only if it is an original page. If a page in Level 2 memory is requested by the Interdata to satisfy a page fault, a bit is kept to record whether or not this page is original to the address space in Level 3 memory. If it is, then the page must be copied back to the user's address space before being overwritten when paged out of the Nova's Level 2 memory.

Levels of Memory In The System

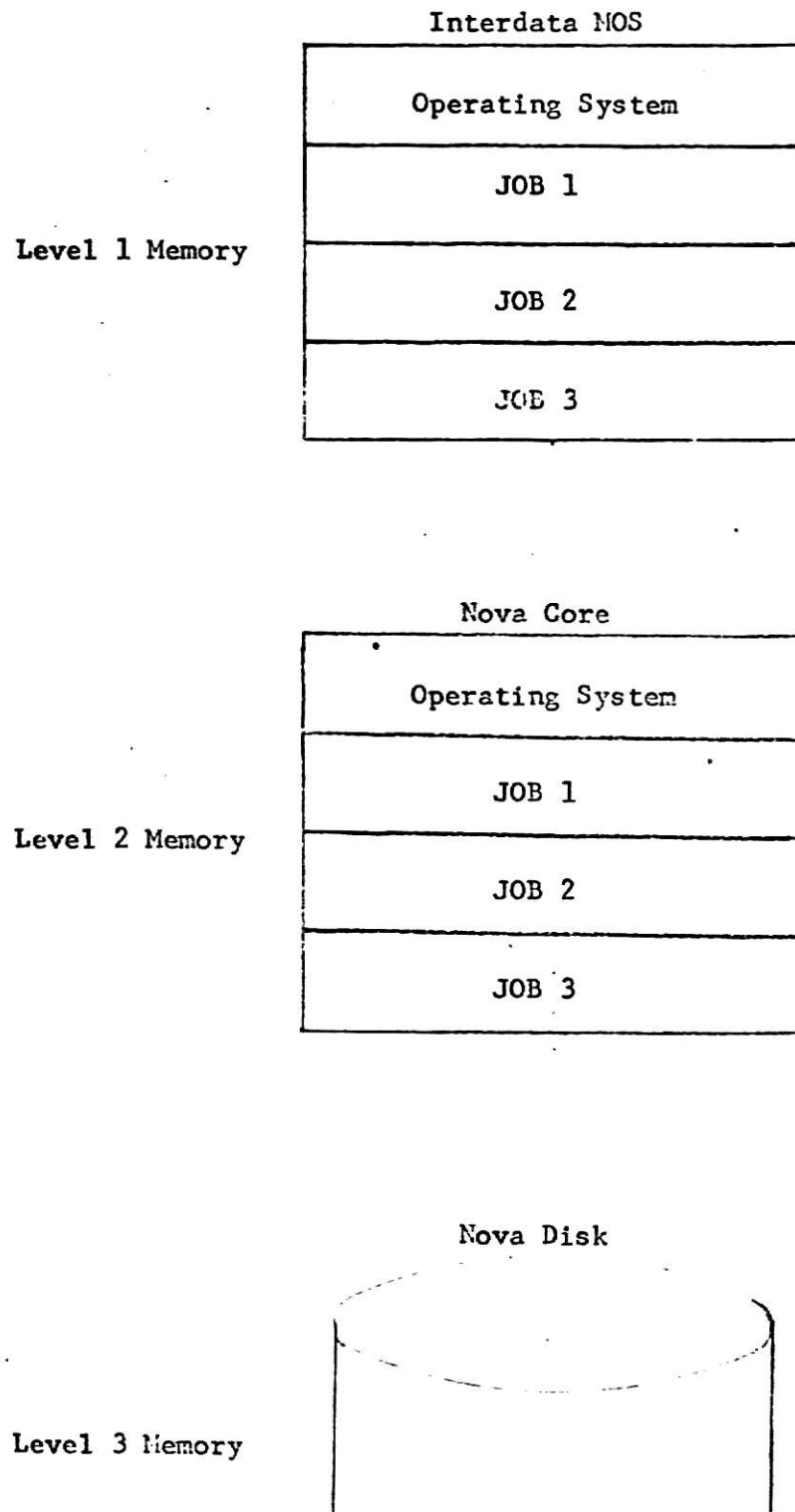


Figure 1-7

This file management system must have tables which keep track of the pages located in its Level 2 memory and Interdata Level 1 memory and all of the user's files located in Level 3 memory. This management system must also interact with the Interdata. All I/O communication and messages must be received and handled by this system.

The first level of memory is located in the Interdata. Located in Level 1 memory will be the user's PCB (Process Control Block). Each user will have space for a fixed number of pages of his program, along with a page table containing information necessary to handle page faults and address mapping.

1.8.3 LOCATION OF SOFTWARE

I/O SVC parameter blocks are created in the user's extended operating system in Level 1 memory. These parameter blocks are needed to inform the peripheral processor of the type of I/O which is to be done. Also the task identification, virtual page number, interval timer and starting and ending location must be included in the parameter block. These parameter blocks are built and then passed to the Nova system.

The FCS (fixed control store) is read only memory which contains the Interdata machine language instruction interpreter. The I/O instructions will not be interpreted by an emulator as are other instructions, but will be interpreted by the program located

in the FCS. The dynamic control store (DCS) will contain the language emulator. The address translator, which recognizes and handles page faults and virtual to real address translations, will also be stored in the DCS portion of the Interdata Level 1 memory. The proposed design may be likened to that of a single system. Level 1 Interdata memory corresponds to primary memory. Level 2 Nova core memory, and Level 3 Nova disk memory corresponds to secondary memory.

1.9 SUMMARY

In order to bring the overall picture of the HIMICS system into focus a short summary of the system will be given. This system will be in a multiprogramming environment. Each user will initially have 64K of virtual address space at his disposal. All real I/O will occur in the Nova. A user's source program will be spooled into an input buffer on the Nova's disk. The source program will then be copied onto the user's virtual address space also located on the Nova disk. The user's data will be put in his input file on the Nova disk. The user's source program will be divided into fixed length pages. Each user's program will be given a starting virtual address of zero. As soon as the user's job is set to running by the Interdata operating system, a PCB is created in the Interdata memory. The page table will be located in this PCB.

This table will be empty when the job becomes running. A page table is also set up in the Nova Level 2 memory.

Immediately after a job is started, a page fault will be generated in the Interdata. Page zero will be requested, and paged in from Level 3 memory to Level 1 memory. The system is now ready to begin execution of the program.

The emulator is given the current instruction to process. All operands must be converted to real addresses. When the page needed is not located in the Interdata, a page fault is generated and processed. When all operands are mapped, the instruction may be interpreted. The instruction counter is incremented, and the next instruction is executed (unless the previous instruction was a jump of some kind).

A special case is encountered when the end of a job is reached. An I/O SVC must be generated to the Nova to empty the output buffer to the output device. Then a job termination message will cause the address space, buffers, and Nova core page space to be released. The Interdata extended operating system then terminates the job in the Interdata, and a new user's job is initiated.

1.10 INTRODUCTORY DESCRIPTION OF REMAINING CHAPTERS

The remainder of this paper will be concerned with levels five and six, which are the levels contained in the Nova. (See Figure 1-5.) Levels one through four are included in a report by Smith (5).

Chapter two will include a general discussion of the paging algorithms, and page migration patterns. Chapter three will dwell on the implementation of the system. This will include algorithms and data structures for implementation. Chapter four will be a short summary of the HIMICS system and a concluding survey of future work which could be done on the system.

CHAPTER TWO

2.1 INTRODUCTION

The three main functions the Nova performs in the system are, page management, file management and Input/Output for the system. In this chapter page management will be analyzed in detail. Different page flow patterns will be traced through the system. Three different paging options will be discussed and high level algorithms will be given for their implementation. File management and I/O for a general system will be discussed, but not in detail, since the operating system being used in this implementation will handle most of this for the system. I/O for this system will be discussed in detail in Chapter Three.

2.2 PAGE MANAGEMENT

Programs executing in a virtual memory environment, using demand-paging, are brought into main memory in fixed length blocks called pages. Most systems use page sizes ranging from 256 to 1024 words. For the system being implemented in this paper a natural choice for the page size is 256 words. This is because the particular disk drive being used in this implementation reads and writes in blocks of 256 words, with each word being 16 bits long.

2.2.1 THRASHING IN PAGED MEMORY SYSTEMS

Thrashing (8, 13) is a troublesome phenomenon which may seriously interfere with the performance of paged memory systems. It is characterized by too much paging. This causes the processor to be idle a high percentage of the time while performing the actual page transfers. This excessive overhead can cause severe performance

degradation or even collapse of the system. The prime cause of paging's performance degradation is the large time required to access a page stored in auxiliary memory. This is due to the rotation and head positioning times associated with most secondary memory devices.

In the implementation proposed in this paper the secondary memory access time will be minimal or transparent to the overall efficiency of the system. This is due to two main strategies in the design of the system. The first is in the use of three levels of memory, see Figure 1-7, as opposed to the normal two levels of memory. In this proposed system the second level of memory will be directly addressable core located in a Nova mini-computer. Thus the page transfers between Level 1 memory and Level 2 memory will be much faster since they are a core to core type transfer as opposed to a rotational device to core transfer. The second reason is the fact that this proposed system has a dedicated mini-computer to handle the paging. This achieves parallel processing. Thus when one task goes blocked because of a page fault, the host machine can continue processing other tasks while the requested page is being paged in by the second processor. Even if the requested page is in the third level of storage, in effect we have eliminated the wait time in the system since it continues to do parallel processing.

2.2.2 PAGING OPTIONS

The Nova will handle paging for the system under one of three options. The option the system will run under will be set at

system generation time. These options will include the ability to run the system under:

- (1) A competitive variable working set size based on local paging rate.
- (2) A competitive variable working set size based on a global scale.
- (3) A non-competitive fixed partitioned working set size.

The system is being designed with these three options so that statistical data of the system can be recorded and performance evaluation analysis made of the system under the various options. The layout of Level 2 memory for these options are shown in Figures 2-1A thru 2-1C.

Under Option 1, the competitive variable working set size based on local paging rate, each job has a portion of Level 2 memory. See Figure 2-1A. This portion of memory is variable as represented by the wavy lines. The portion each job has depends upon that job's paging rate and the availability of Level 2 memory. A job having a high paging rate tends to increase its working set size while a job with a low frequency paging rate will decrease its working set size.

In Figure 2-1B, Option 2, the competitive variable working set size based on a global scale, a job is given the next available page frame when a page fault is encountered if one is available. If all the page frames are in use, then the page that has been unused the longest is paged out (LRU) (1). The job then uses this vacated page frame. The LRU algorithm is applied globally to determine which page is to be removed if all page frames in Level 2 memory are

Level 2 Memory Layout For Various Page Options

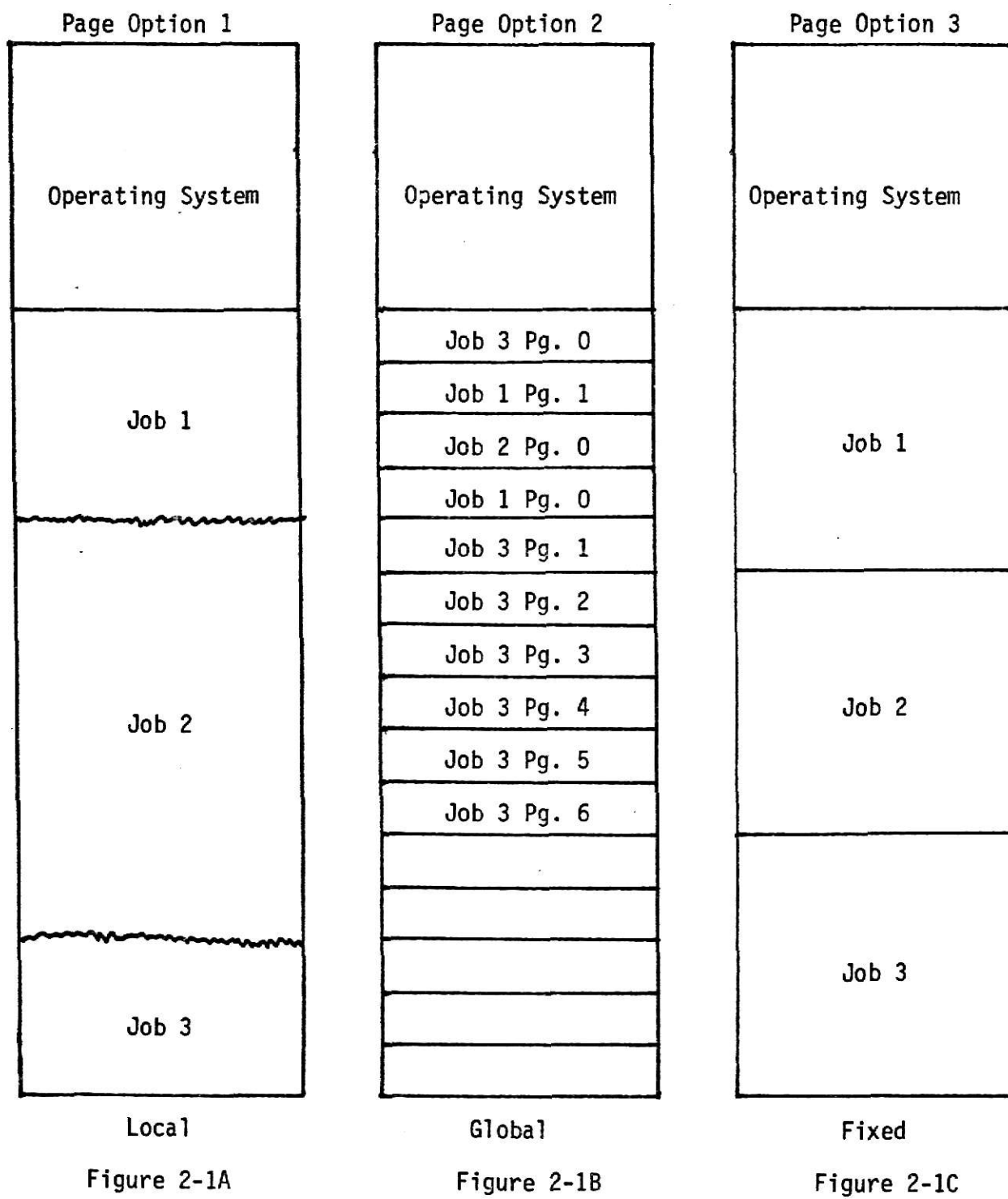


Figure 2-1

in use. Thus under this option a single job may use all of Level 2 memory. This will be explained in detail in section 2.2.5.2.

Under Option 3, the non-competitive fixed partitioned working set size, the amount of Level 2 memory is partitioned into areas of equal size. See Figure 2-1C. The size of each area is determined by dividing the total size of Level 2 memory by the number of partitions. Any remainder of page frames will be distributed by adding to each job one page frame, starting with Job 1, until all remaining page frames have been distributed. For example, if 5 jobs are to be multiprogrammed and there are 19 page frames ($19/5=3$ remainder 4), the five partitions would have 4, 4, 4, 4 and 3 page frames respectively. These calculations and allocations of Level 2 memory are set at system generation time and are static.

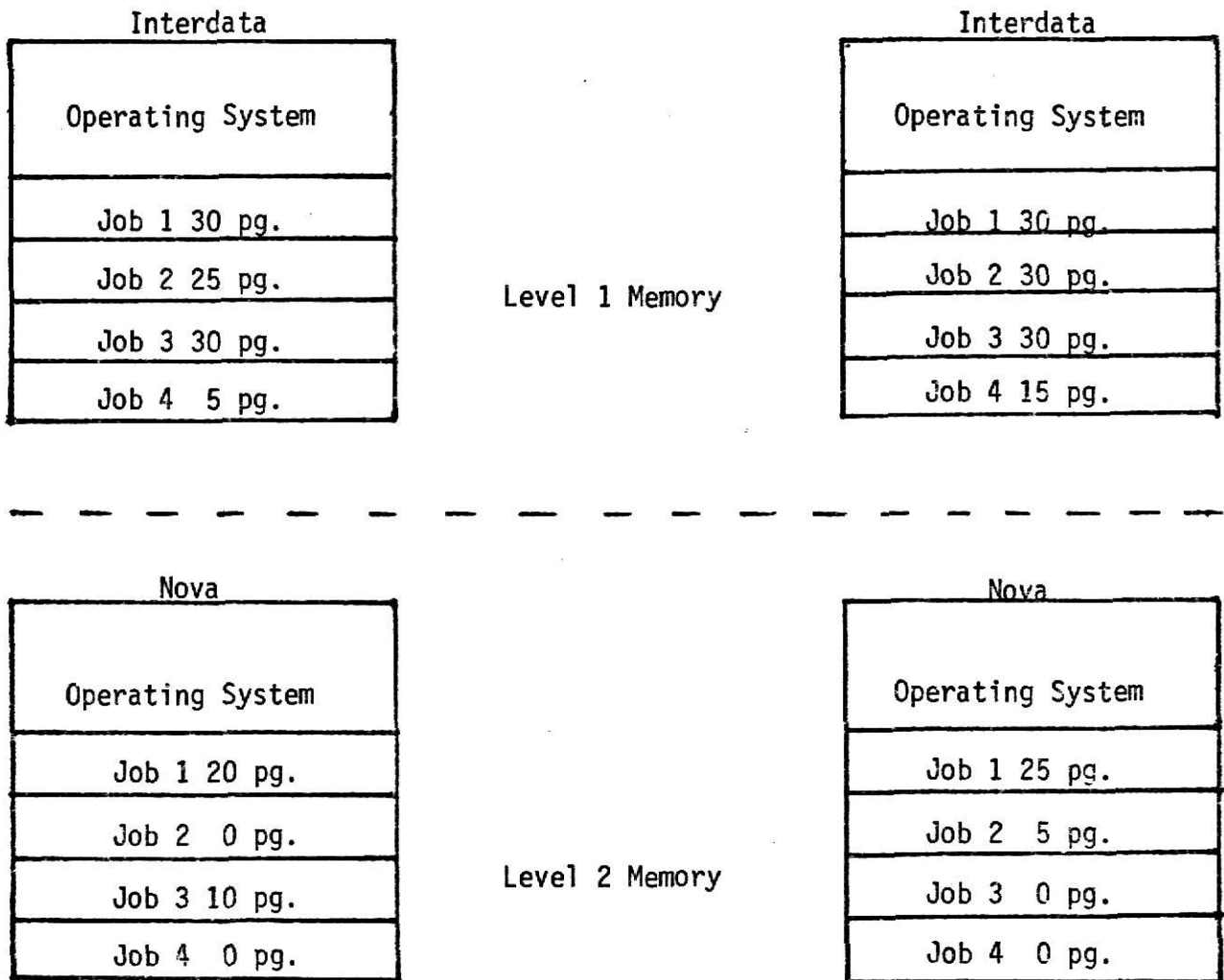
2.2.3 WORKING SET SIZE

Working set size used above refers to the number of pages a program has in Level 1 memory, plus the number of pages it has in Level 2 memory at a given time. The working set size will therefore be a variable with respect to time. For example, Job 1 may have 30 pages in Level 1 memory and 20 in Level 2 memory giving it a working set size of 50 pages at some time (T_i). At T_{i+10} it may have 30 pages in Level 1 memory and 25 in Level 2 memory, giving it a working set of 55 pages. See Figure 2-2.

2.2.4 PAGING TRANSFER FLOW PATTERNS

Paging in the system can follow one of four transfer flow patterns. These patterns, or sequences of page transfers are given in Figures 2-3 thru 2-6.

Working Set Size



Job	Working Set Size (Level 1 + Level 2 Memory)	
	Time T_i	Time T_{i+10}
1	50	55
2	25	35
3	40	30
4	5	15

Figure 2-2

Transfer Flow Patterns

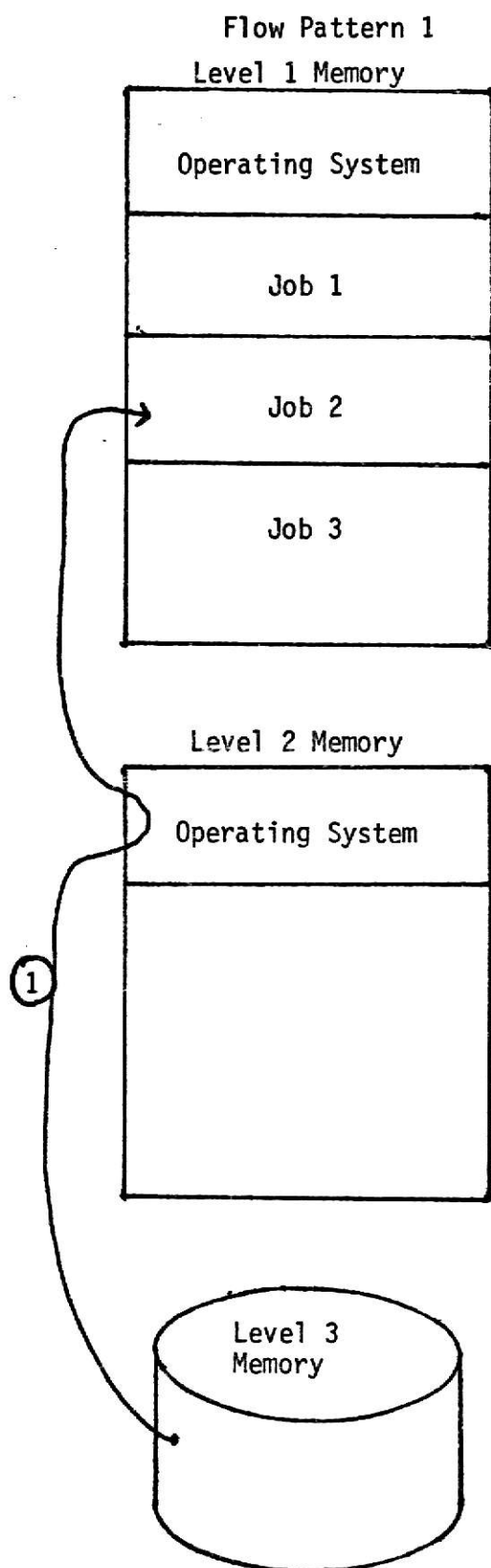


Figure 2-3

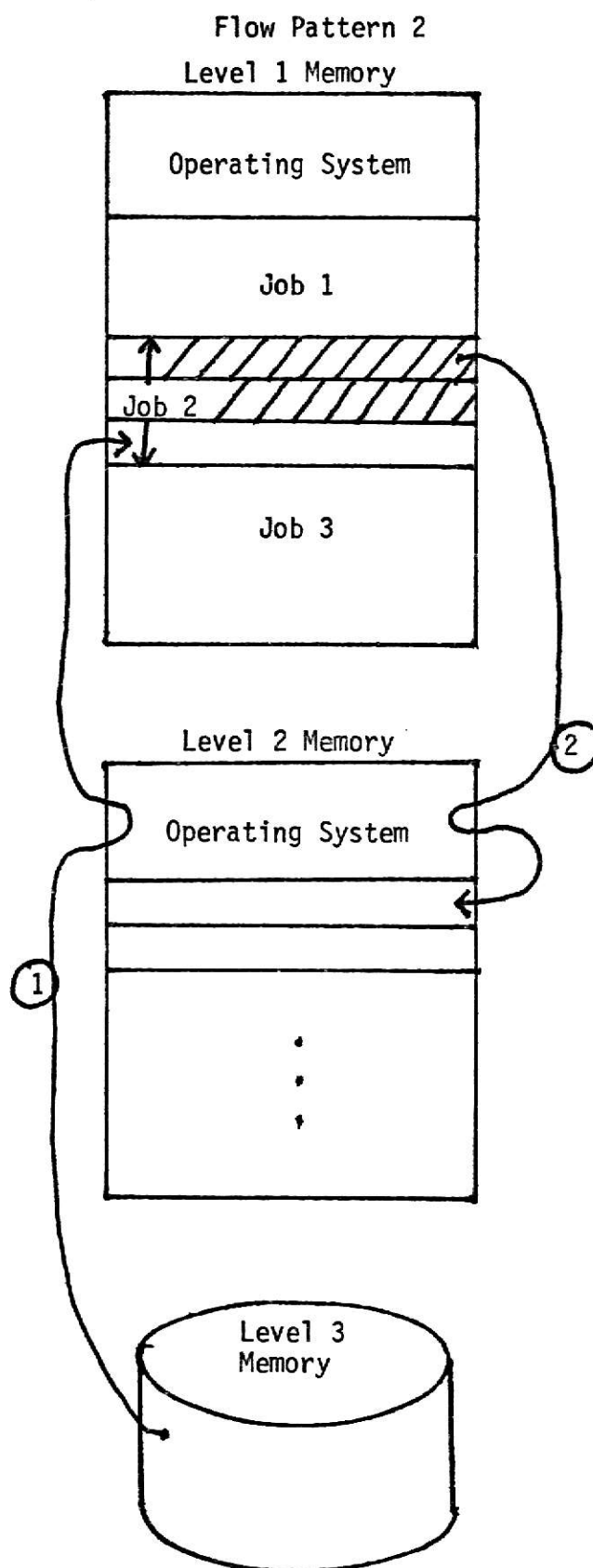


Figure 2-4

Transfer Flow Patterns

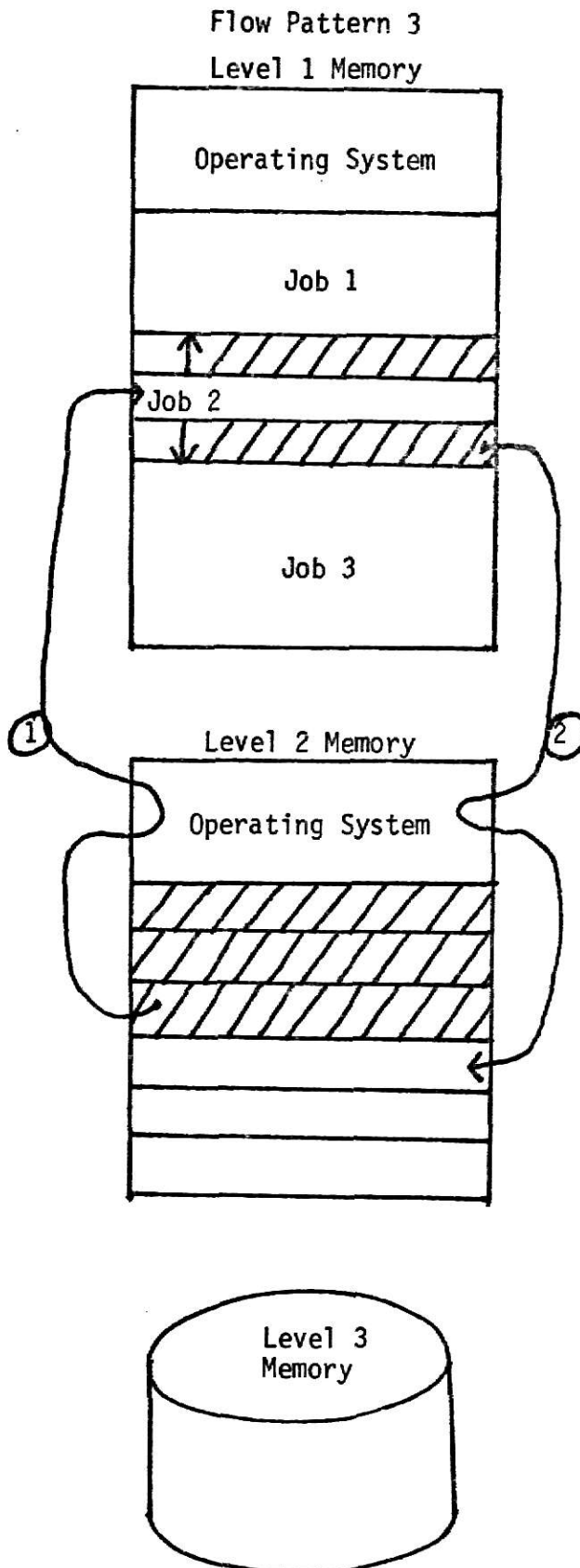


Figure 2-5

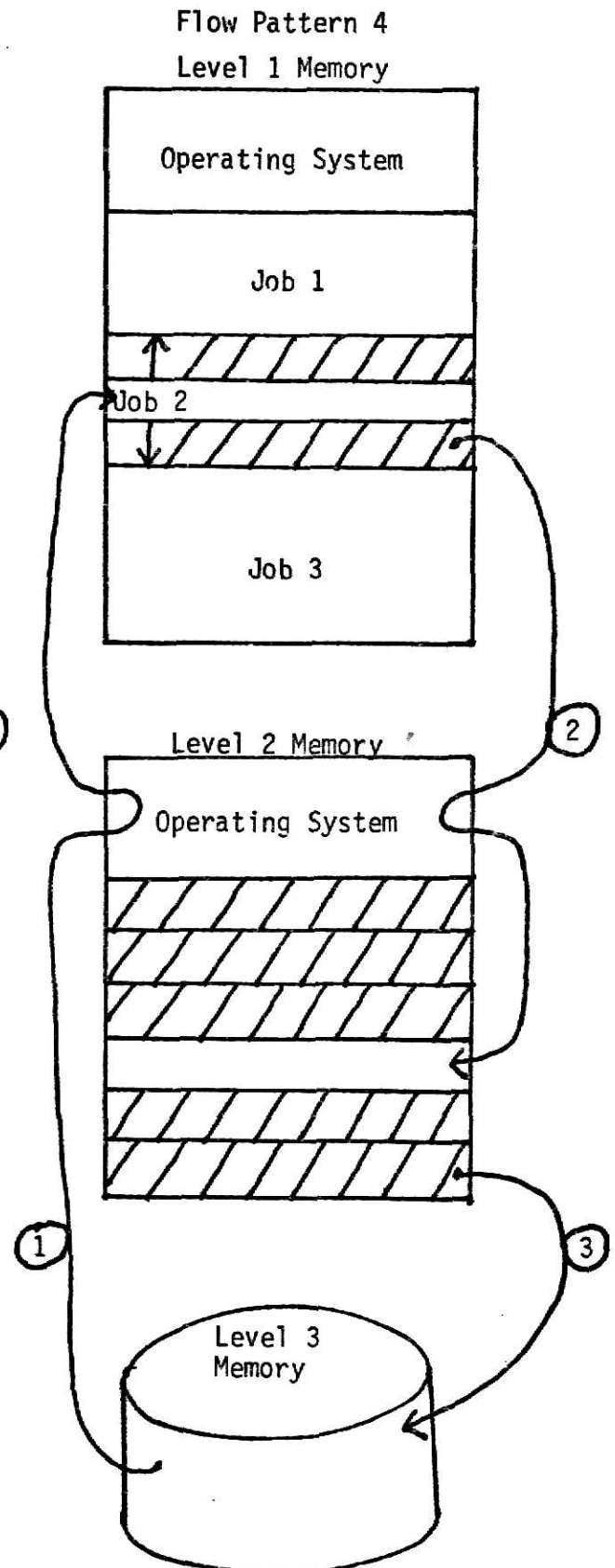


Figure 2-6

2.2.4.1 TRANSFER FLOW PATTERN 1

The simplest transfer is shown in Figure 2-3. This transfer is from Level 3 memory to Level 1 memory. All jobs will initially start with this type of transfer. This is because all jobs are initially spooled to Level 3 memory and before execution can begin the first page has to be transferred into the Interdata's Level 1 memory. The Interdata will issue a request to start this transfer. This type of transfer, Level 3 memory to Interdata Level 1 memory, will continue for each page fault request by a job until that job's Level 1 memory space partition in the Interdata is filled. This partition size will be set at system generation time and will vary depending on how many jobs are set to be multiprogrammed.

2.2.4.2 TRANSFER FLOW PATTERN 2

This type of transfer is shown in Figure 2-4. It occurs when a page fault is requested by a job and there are no page frames available in that job's working set in the Interdata's Level 1 memory. There is always one page frame available for the transfer, but it is not counted in the job's working set. In this transfer a page is transferred from Level 3 memory, to Level 1 memory. Since this takes away the transfer page frame in Level 1 memory, a page has to be paged out of Level 1 memory. This is to allow future page faults an empty page frame into which they can be transferred. This is used for efficiency. If the extra page frame was not available, a page would first have to be transferred out of Level 1 memory before the new page could be transferred in. With the extra page frame, the page to be transferred out can be transferred after

the new page has been transferred in. This allows the process, that caused the page fault, to continue executing in the Interdata while the outgoing page is being transferred to a lower level of memory by the Nova. This flow pattern will continue until the job's Level 2 memory in the Nova is filled, or a page is requested that resides in Level 2 memory. Then flow pattern 3 or 4 will preside.

2.2.4.3 TRANSFER FLOW PATTERN 3

This type of transfer is shown in Figure 2-5. It occurs when a page fault is encountered by the Interdata and the page requested is in the Nova's Level 2 memory. The transfer into the Interdata's Level 1 memory is the same as under transfer flow pattern 2, except the page is copied from the Nova's Level 2 memory instead of from Level 3 memory. This core to core transfer is much faster than the disk to core transfer. This is because the disk to core time has to include a disk seek time for the Read/Write heads on disk to be positioned in order to read a page from disk. This seek time runs on the average of 50-100 milliseconds and is very time consuming compared to the core to core transfer time of approximately 250 microseconds. This is the main reason for the extended page memory system being extended into the Nova's Level 2 memory.

The second part of the transfer, the paging out of Interdata Level 1 memory into Nova Level 2 memory introduces two new possibilities. One is the possibility that the page to be paged out already has an identical copy existing in Level 2 memory. The other is that there is not an identical copy of it. This includes

the case where there was an identical copy of the page to begin with, but the page while in the Interdata was changed, hence two identical copies do not exist. If the page has been changed in the Interdata it is flagged as an original by the Interdata. If the page to be paged out is an original it needs to be recopied. If it is not an original it need not be recopied. If it does not need to be recopied it saves the transfer time. These two possibilities will always exist whenever a page fault is encountered and a previous copy of the page existed in the lower of the two levels of memory prior to the page fault. In this case lower level memory being a higher level number (i.e. Level 1 memory is the highest and Level 3 the lowest).

2.2.4.4 TRANSFER FLOW PATTERN 4

This is the most complex page flow pattern in the system. It is shown in Figure 2-6. Before this type of flow can occur four conditions must exist. First, the job creating the page fault must have all its page frames in Level 1 memory in use. Second, all the page frames available for that job in Level 2 memory must be in use. Third, the page being paged out of Level 1 memory must not have an identical copy residing in Level 2 memory. And fourth, the page causing the page fault must reside in Level 3 memory.

As can be seen in Figure 2-6, the requested page is copied from Level 3 memory into Interdata Level 1 memory. This takes the transfer page from Level 1 memory in the Interdata. A page thus has to be released to recreate the transfer page frame. A page is selected by the Interdata to be paged out to the Nova Level 2 memory.

This frees up a page frame in the Interdata for the next transfer, but in turn takes away the Nova's transfer page frame in Level 2 memory. It therefore has to select a page for removal. This page is then moved to Level 3 memory. This releases a page frame from the Nova's Level 2 memory to be used for the next page fault to Level 1 memory. This completes the cycle.

The goal of this hierarchal memory system is to maximize the number of times pages are in the faster memory levels when being referenced. This implies that flow pattern 3, Figure 2-5, is the desired page flow pattern we wish to achieve in this system. This is because the core to core transfer is much faster than the disk to core transfer as noted previously.

2.2.5 PAGING ALGORITHMS USED IN SYSTEM

In this implementation, using two mini-computers and 3 levels of memory, two paging systems will be used. One to handle the paging between Level 1 and Level 2 memory and the other to handle paging between Level 2 and Level 3 memory. Paging between Level 1 and Level 2 memory will be handled by a Least Recently Used (LRU) (1) approximation algorithm called a Not Used Recently (NUR) (1). The algorithm will be implemented in the host machine. Its implementation is dealt with in detail in a paper by Smith (5). We will look briefly at its logic in order to better understand the paging algorithm necessary between Level 2 and Level 3 memory. Also the reason why the NUR was preferred over the LRU in the host machine is discussed.

The LRU selects for removal the page that has not been referenced for the longest period of time. It is based on the theory that if a page is referenced, it is likely to be referenced again soon. Conversely, if it has not been referenced for a long time, it is unlikely to be needed in the near future. The LRU implies that a time has to be recorded each time a page is referenced. Then when a page is to be removed all these times have to be compared to find which one has been present the longest. This is too time consuming to be done using software and is the reason the NUR is preferred for this implementation.

The NUR approximates an LRU by setting a reference bit associated with the referenced page to 1 everytime the page is referenced. Periodically the reference bits are reset to 0. Thus anytime a page has a 0 reference bit it is known the page has not been referenced since the last time the bit was reset to 0. This means the page has not been used recently and is a candidate for removal. This in turn implies that at any given time, Level 1 will contain the most recently used pages for that period of time, which is the LRU approximation.

The pages that are removed from Level 1 memory are moved into Level 2 memory. Thus Level 2 memory is an extension of Level 1 memory, and will at a given time still contain the most recently used pages.

The paging algorithm being implemented between Level 2 and Level 3 memory will be a true LRU and not an approximation. The reason this can be done at this level and not above is that Level

2 memory is not directly addressable by the user's program. Thus if a page in Level 2 memory is accessed, it is through the operating system. Each time a page is accessed it is moved out of Level 2 memory and this eliminates the need to keep the access times associated with the page, as was described above. Instead a First-in/First-Out (FIFO) stack (1) is kept of all the pages in Level 2 memory. This then is the LRU stack (10). Thus the first page paged into Level 2 memory is the least recently used and will be paged out first when Level 2 memory is full and a new page is to be moved from Level 1 memory to Level 2 memory.

2.2.5.1 PAGING ALGORITHM DISCUSSION FOR OPTION 1

Under Option One, paging in the system will be based on a competitive variable working set size. This means that a job's working set size is allowed to expand or contract. This size change will occur in Level 2 memory. This is because Level 1 memory is a fixed size for each job and is set at system generation time. The determining factor as to whether a job's working set size is changed is based upon its current paging rate compared to its past paging rate. If its current paging rate is significantly slower, then one page is paged out of its working set and the page frame put on a free list for use by other programs. If its current rate is faster, and there is a free page frame available, then its working set size is increased by one page frame. Note, this implies one job may never pre-empt a page frame from another job. It only gets a page if there is one available. In other words the algorithm is applied on a "LOCAL" basis. This means a job's working set stays intact despite the other programs paging activity with which it shares

memory. There is also the case where the past paging rate and the current paging rate are about the same. In this case the working set size is not changed. In this initial design, if the current paging rate is within plus or minus 10% of the previous page rate, the working set size will not change. This will be implemented so that it can be changed as the paging behavior will have to be monitored in the system before an optimum range can be determined.

Before presenting the high level algorithm we will look again at the hierarchy structure of the memory levels and make a few notes. In Figure 1-7 we displayed the three levels of memory in the system. Also it should be remembered that the working set size was defined as the total sum of pages a job has in Level 1 memory plus Level 2 memory, Figure 2-2. As noted above, Level 1 memory is a fixed size. Thus looking back at the 4 transfer flow patterns (Figures 2-3 thru 2-6) it can be seen that increasing a job's working set size will only be effective in reducing the paging rate in flow pattern 4, Figure 2-6. This is because in flow pattern 1 and 2, Level 2 memory for the job is not yet saturated and in flow pattern 3 increasing the size of Level 2 memory for a job will not reduce paging as the paging being done is already within the job's working set. This leaves us to only be concerned with flow pattern 4.

The page fault rate is based on a time variable. This time variable being the length of CPU time that has elapsed for the given task between page faults. When a page fault occurs, this interval time is saved to be compared against when the next page fault occurs.

Initially it will be set to zero. When a page fault occurs, the current fault rate time is compared against 90% of the previous fault rate time to see if the rate of paging has increased by a factor of at least 10%. This indicates a faster rate of paging. If so a check is made to see if there is an available page frame for use. If there is, then this task's working set size is increased by one page. If the above tests fail, then a check is made to see if the rate of paging has decreased by a factor of 10%. This is done by comparing the current fault rate against 110% of the previous fault rate. If the current fault rate is greater than 110% of the previous fault rate, indicating a slower paging rate, then the working set size for the task is decreased by one page. If the fault rate has neither increased or decreased by 10%, then the task's working set size remains the same. The current fault rate is then stored as the previous fault rate to be used when the next page fault occurs.

The algorithm given below will be entered each time a page migration between Level 2 and Level 3 occurs with the exception of a page migration for I/O. I/O will be discussed in section 2.3.2. It has no bearing on working set size but can cause page migrations between memory levels.

High Level Algorithm For Option 1

ENTER PG_OPTION_1(JOB_ACC_TIME)

1. CURRENT_FAULT_RATE=JOB_ACC_TIME-PREVIOUS_FAULT_RATE

2. IF CURRENT_FAULT_RATE<PREVIOUS_FAULT_RATE*90% AND

THERE IS AN AVAILABLE PAGE FRAME THEN

2.1 INCREASE WORKING SET SIZE BY 1

```

3.      ELSE IF CURRENT_FAULT_RATE>PREVIOUS_FAULT_RATE*
          110% THEN

3.1      DECREASE WORKING SET SIZE BY 1

4.      PREVIOUS_FAULT_RATE=CURRENT_FAULT_RATE

5.      RETURN

6.      END

```

2.2.5.2 PAGING ALGORITHM DISCUSSION FOR OPTION 2

Under this option, the competitive variable working set size, Level 2 memory is treated on a "GLOBAL" scale. The algorithm is implemented by using an LRU stack. When a page migration occurs between Level 1 and Level 2 memory the job is given the next available page frame if one exists. If Level 2 memory is saturated, then the LRU stack is checked to see which page in Level 2 memory has been unreferenced the longest. This page is then paged out to Level 3 memory, regardless of which job it belongs to, and its vacated page frame used by the requesting job.

For example, take 3 jobs. Job 1 has 80 pages in Level 2 memory, Job 2 has 39 and Job 3 has 1. See Figure 2-7. Let the number of page frames available be set to 120. Let the order of request be such that Job 3's page is the oldest page. Let the next page fault be issued by Job 2. This will cause Job 3 to lose its only page in Level 2 memory and Job 2's new page to be put on top of the LRU stack.

This is shown to illustrate that 1 or 2 jobs may dominate the Nova's Level 2 memory. This may be advantageous as some small jobs, or jobs which display a high degree of locality (8,9) may

Memory Domination Under The Competitive
Variable Working Set Size Option

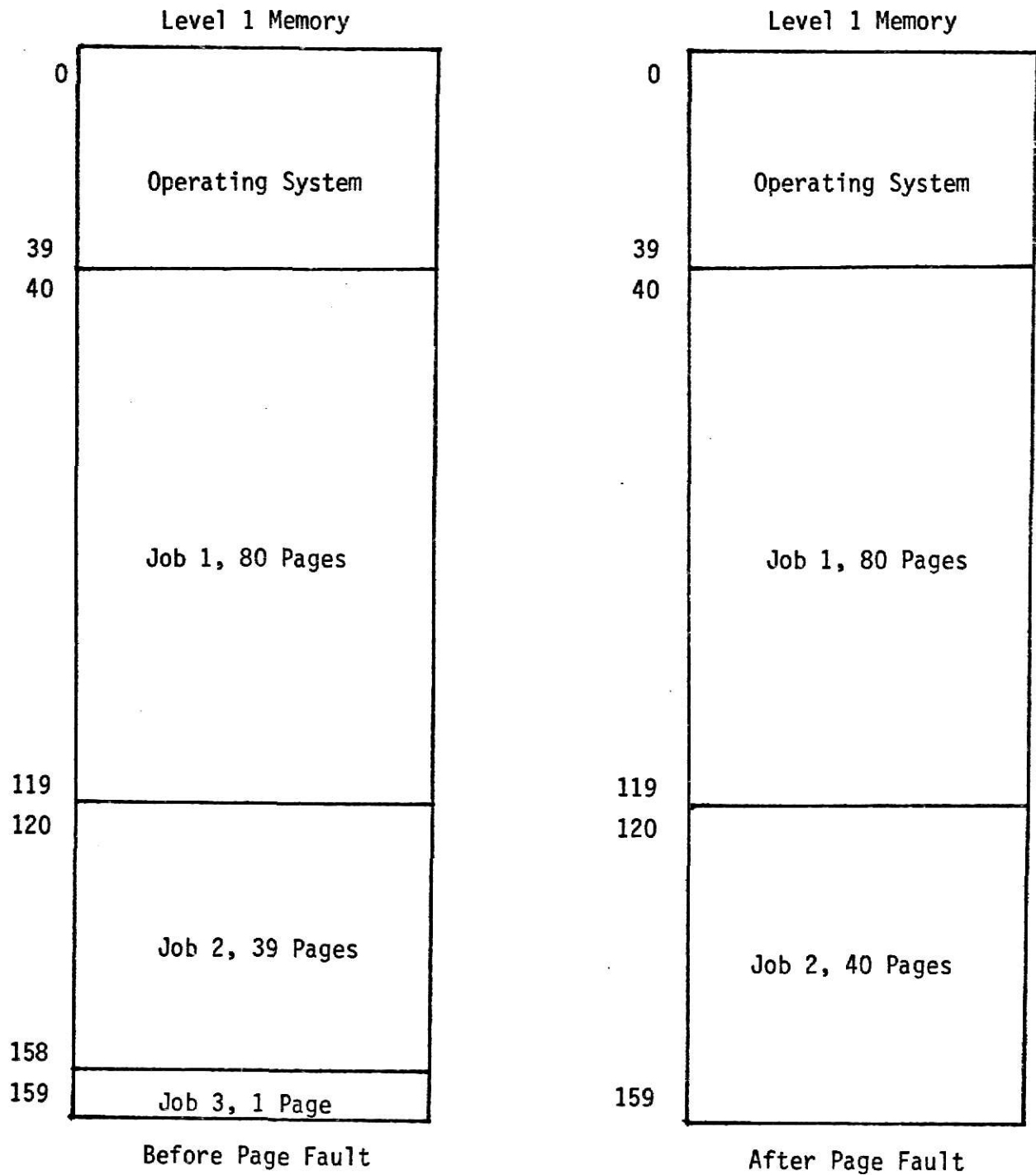


Figure 2-7

never need secondary storage, or at least not a large amount of it.
The high level algorithm for page Option 2 is presented next.

High Level Algorithm For Option 2

ENTER PG_ALG_OPTION2(JOB#)

1. CHECK FOR EMPTY PAGE FRAME
2. IF ONE EXISTS THEN
 - 2.1 USE AVAILABLE PAGE FRAME
3. ELSE CHECK LRU STACK AND TRANSFER TO LEVEL 3 MEMORY
 THE PAGE THAT HAS BEEN RESIDENT THE LONGEST IN
 LEVEL 2 MEMORY AND USE THIS NEWLY RELEASED PAGE
 FRAME
4. RETURN

2.2.5.3 PAGING ALGORITHM DISCUSSION FOR OPTION 3

Under this fixed size working set option, the total number of page frames available will be divided by the number of jobs being multiprogrammed. Each job will then have a fixed maximum number of pages that it may use. For example, if there are 90 page frames of Level 2 memory and 3 jobs are running, each job may have a maximum of 30 pages of Level 2 memory ($90/3=30$). Each job will then use its 30 pages as an LRU stack for paging operations. The logic of the algorithm is the same as in option two. The difference is that option two has a global pool of page frames and option three has a local pool of page frames. The high level algorithm for option 3 is given below.

High Level Algorithm For Option 3

ENTER PG_ALG_OPTION3(JOB#)

1. IF PG_FRAMES_IN_USE(JOB#) \leq TOTAL_AVAILABLE THEN
 - 1.1 USE AVAILABLE PAGE FRAME
2. ELSE OBTAIN THE LEAST RECENTLY USED PAGE FROM LRU
 STACK, MOVE THE LRU PAGE TO LEVEL 3 MEMORY AND
 USE GIVEN PAGE FRAME
3. RETURN

2.2.6 THRASHING AS RELATED TO PAGING OPTIONS

The least amount of thrashing should be exhibited by option one. This is because of two things. First, the algorithm is applied on a local basis. This means excessive paging by jobs in the system will not influence the other job's paging rate. Second, the working set size for a job is allowed to expand or contract if its local paging rate indicates it to be desirable. This will tend to reduce thrashing caused by a fixed partition size as in option three. This is because the critical region causing thrashing may be only 1 or 2 pages. Under the fixed option, the extra 1 or 2 pages can never be included in the job's working set, whereas under option 3 the working set for the job will have the opportunity to increase its size the extra several pages it needs.

The most amount of thrashing should occur under option two, the global competitive working set option. This is due to the global nature of the algorithm, which means jobs will be pre-empting pages from each other. In order for a job to pre-empt a page from

another job, flow pattern 4 (Figure 2-6) must be the pattern followed. This could cause severe CPU wait time in the host machine due to the high speed page transfer between Level 1 and Level 2 memory and the slow speed between Level 2 and Level 3 memory. See Figure 2-6.

2.3 I/O FOR SYSTEM

There will be two basic types of I/O processing handled by the Nova for the system. The first will be the I/O which will take care of the spooling of jobs in the system. The second type will handle program requested data transfers during program execution.

2.3.1 SPOOLING OF I/O

A user's program and unit record input data are initially spooled to disk in the user's virtual address space, starting at virtual address 0. The source program will be divided into fixed length page segments of 256 words each as it is being spooled. Upon completion of a job, all output for the job has been spooled to disk. The Nova then dumps this output to the printer or other output device as requested by the job.

2.3.2 PROGRAM REQUESTED I/O

Program requested input will be handled similar to a page fault. When the user program request input, a message is sent to the Nova by the Interdata. This message will include the starting and ending virtual addresses of the area that the data is to be read into. The Interdata will then lock the page or pages of the I/O data area in, in Level 1 memory if they are currently residing

in Level 1 memory. This is done so that the associated page(s) do not become candidates for removal during I/O. Next the page that contains the beginning virtual address of the I/O area is located. If it is in Level 1 or Level 2 memory, then the I/O is performed to its corresponding real addresses. Next its page map table entries are updated. If the page is located in Level 3 memory, it first has to be copied to an I/O work buffer located in Level 2 memory. Then the I/O is performed to the work buffer and the page transferred back to its Level 3 address. If the virtual addresses span more than one page, then each succeeding page is treated in the same manner as described above until all the I/O has been completed.

2.4 FILE MANAGEMENT

File management in this implementation will only be concerned with that which is necessary to do the paging and the I/O spooling. The main reason for this restriction is hardware. The present system only has one disk and it is being dedicated to I/O spooling and Level 3 memory usage in the system.

The following file management system is described for a system which allows dynamic allocation of core. Since the system being used for the actual implementation of this system does not allow dynamic allocation, the implementation version given in chapter three will vary from the following description. This version is included here as it is a more general version.

2.4.1 DATA BASES FOR FILE MANAGEMENT

This system will use seven types of tables to handle the file management and paging. They are:

- (1) Spool Table (ST).
- (2) Disk Usage Sector Table (DUST).
- (3) Memory Block Table (MBT).
- (4) Job Table (JT).
- (5) Page Map Table (PMT).
- (6) File Map Table (FMT).
- (7) Extended Page Fault Table (EPFT).

The spool table (Figure 2-8), keeps track of the files associated to a unique job. For initial implementation this table will remain in Level 2 memory. In an actual production environment it could become necessary to move it to Level 3 memory due to its potential size, as every job entered in the system will have an entry in the spool table. Entry one in the spool table contains a unique job identification number for each job entering the system. Entry two contains that job's source deck file name. Entry three contains the file name of the job's data input, if any exist. Entry four contains the file name of the job's object deck. Entry five contains the job's output file name. This output file will be dumped to the designated output device at the close of a job.

The Disk Usage Sector Table (DUST), Figure 2-9, is used to store the status of each sector of the disk. Each sector on disk will have a corresponding bit in the DUST table to represent whether the sector is in use or not. A 0 in the table means the sector is not in use. A 1 indicates the sector is in use. A bit map (1)

Spcol Table

	Job ID	Source Deck File ID	Data Deck File ID	Obj. Deck File ID	Output File ID
1	J100	J101	J200	J98	J112
2	J71	J72	J73	J74	J75
3					
4					
5					
⋮					
100					
	1	2	3	4	5

Figure 2-8

Disk Usage Sector Table (DUST)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	1	0	0	1	0	1	1	0	0	0	0	1	1	1
16	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
32	1	1	1	1	1	0	0	1	1	0	0	0	1	0	1	0
48																
64																
80																
96																
⋮																
⋮																

Figure 2-9

is used because it takes up less storage in the system.

The Memory Block Table (MBT), Figure 2-10A, will consist of two parts. A block number, one for each 256 words of storage in Level 2 memory, and the status of the block that is associated with it. The status will tell if a given block is in use, and if so who has it. There will be one MBT located in the system. It is used for dynamic allocation of storage by the operating system for the various tables.

The Job Table (JT), Figure 2-10B, will consist of four parts. The first entry will contain a job number for each job running in the system. The second entry will contain that job's starting address of its page map table. The third entry will contain the job's file map table starting address. The last entry will be the job's length in number of pages. There will be one JT for the system. The JT's main use is to save core. If it were not used each PMT and FMT for each job would have to be given a specific location in core. Furthermore, they would each have to be allocated to hold the maximum number of pages. With the job table, core for the PMT and FMT can be allocated according to the length of the job running.

There will be one Page Map Table (PMT), Figure 2-10C, per job. It will consist of two entries. Entry one will be the address where the page is located in memory. Entry two will be the status of the page. This will indicate if the page is an original. An original page is a page that has been changed in a higher level of memory but not changed in its virtual address space in Level 3

Memory Block Table

Blk. #	Status
0	Op. Sys.
1	Op. Sys.
2	Job 3 Pg. 1
3	Job 2 Pg. 1
4	Job 3 Pg. 0
5	Job 1 Pg. 0
6	Free
⋮	
N	

Figure 2-10A

File Map Table

110
140
141
142
260
261

Figure 2-10D

Job Table

Job #	Loc. PMT	Loc. FMT	Length (pgs.)
1	2000	3103	1
2	1760	3100	3
3	1880	4000	2

Figure 2-10B

Page Map Table

Blk. #	Status
--------	--------

0	0
3	1
0	0
4	1
2	0
5	0

Figure 2-10C

Extended Pg. Fault Table

Job 1 Pg. 0
Job 3 Pg. 0
Job 2 Pg. 1
Job 3 Pg. 1

Figure 2-10E

memory. Hence it is an original since no two identical copies exist. The beginning PMT address, plus a corresponding offset equal to the desired page number, will be used to reference the requested page. Hence the page number does not have to be stored.

The File Map Table (FMT), Figure 2-10D, performs a similar function to the PMT. There will be one FMT per job. There will be one entry in the FMT for each page a job has in Level 3 memory. It will contain the address of where the corresponding page is located in Level 3 memory. The offset principle as described above will be used to reference a particular page.

The Extended Page Fault Table (EPFT), Figure 2-10E, will be used to implement the LRU paging algorithm in the system. It amounts to a FIFO stack of all the pages in Level 2 memory. There will be one EPFT per job.

Figure 2-11 shows how the above tables, used for paging, relate to one another. When a page is requested a check is made to the job table for the location of the page map table. Next the PMT is looked at at the given address plus an offset equal to the requested page number. The block number, first PMT entry, is then checked to see if it is greater than zero. If zero, it implies the page is not in Level 2 memory. If it is greater than zero, the system transfers the given page from the block number, obtained from the PMT, to Level 1 memory. If the block number is zero, the system obtains the starting address of the file map table from the job table, and goes to the FMT to obtain the address of the desired

Locating Requested Pages In The Level 2 Processor

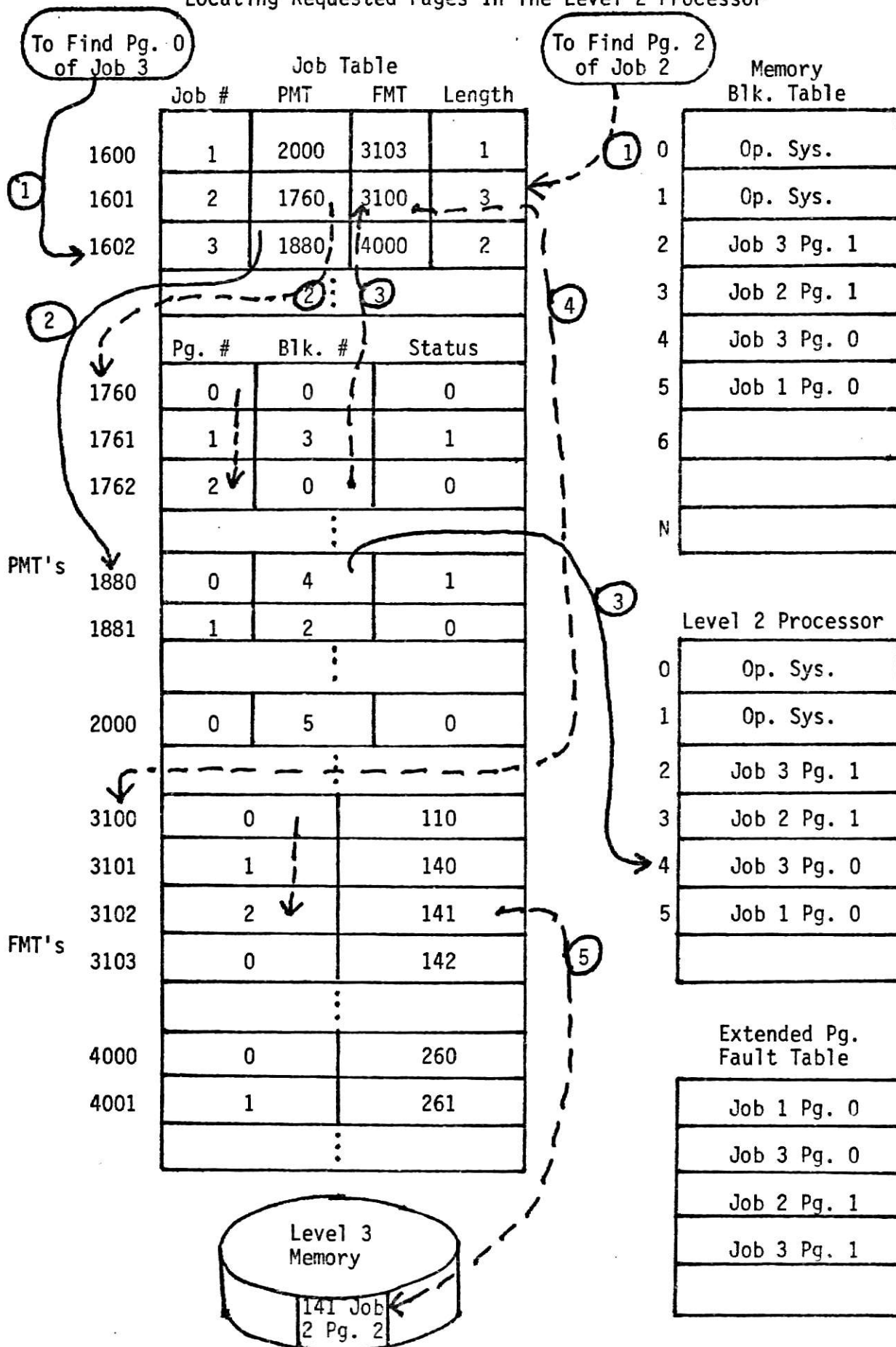


Figure 2-11

page located in Level 3 memory. It is then transferred from Level 3 memory to Level 1 memory.

CHAPTER THREE

3.1 INTRODUCTION

In this chapter we will present the algorithms necessary for the implementation of the HIMICS system in the level two processor. This will mainly include three areas of algorithms. The first area includes the algorithms necessary to do the I/O and to perform the page transfers between levels of memory. The second area includes the algorithms for system generation and initialization. The final area includes the algorithms to run the system under one of the three paging options. Examples of message exchanges will be given in the description of the algorithms. The algorithms are written in a dialect of PL/I. They are intended as descriptive algorithms, rather than actual code. It is recommended that as you read the following algorithm descriptions that you follow along in the algorithms given in appendix A.

3.2 ALGORITHM 1: MAIN DRIVER FOR NOVA ROUTINES

The driver for all of the Nova routines is given in appendix A, as algorithm number one. Its main purpose is to call the appropriate routines when passed a message by the Interdata, or upon interrupt from the card reader. The messages that it will receive are:

- (1) A call to generate the system.
- (2) A call for a requested page to be sent to the Interdata.
- (3) A call at the termination of a job.
- (4) A call to do input or output.

- (5) A call to handle page removal from Level 1 memory.
- (6) A call to spool the program.
- (7) A call to get the next job from the job queue.
- (8) A call to create an object deck file name.
- (9) A call to retrieve a job's object deck file name.
- (10) A call to create an output file name.
- (11) A call to list stack depth counts.

The driver's secondary function is to provide for the data bases that are common to all the subroutines. This will include the data base for page traffic, the data base for the spool table, and the data base for instrumentation.

The page traffic data base will consist of a collection of Page Map Tables (PMT'S), see Figure 3-1. This system will be implemented for three users so three PMT'S will be allocated during system generation. These three PMT'S compose a structure known to the Nova as the Job's Page Control Block (JOB_PCB), which is used for PMT qualification purposes. Since the user is limited to a 64K byte (8 bit byte) virtual address space (or 32K words of 16 bits each), the length of the PMT is set at 128 entries. This allows each page an entry in the PMT, with each entry representing 256K words, or one page of the user's program. This system does not allow for dynamic allocation so the maximum size has to be allocated, even though it may not be needed.

The first entry in the PMT will contain information as to the location in memory of each page of a user's job in object deck form. The job's page numbers will run from 0 to a maximum of 127. Thus in order to find a job's particular page in the PMT, all that

Job Page Control Block

JOB_PCB(3)

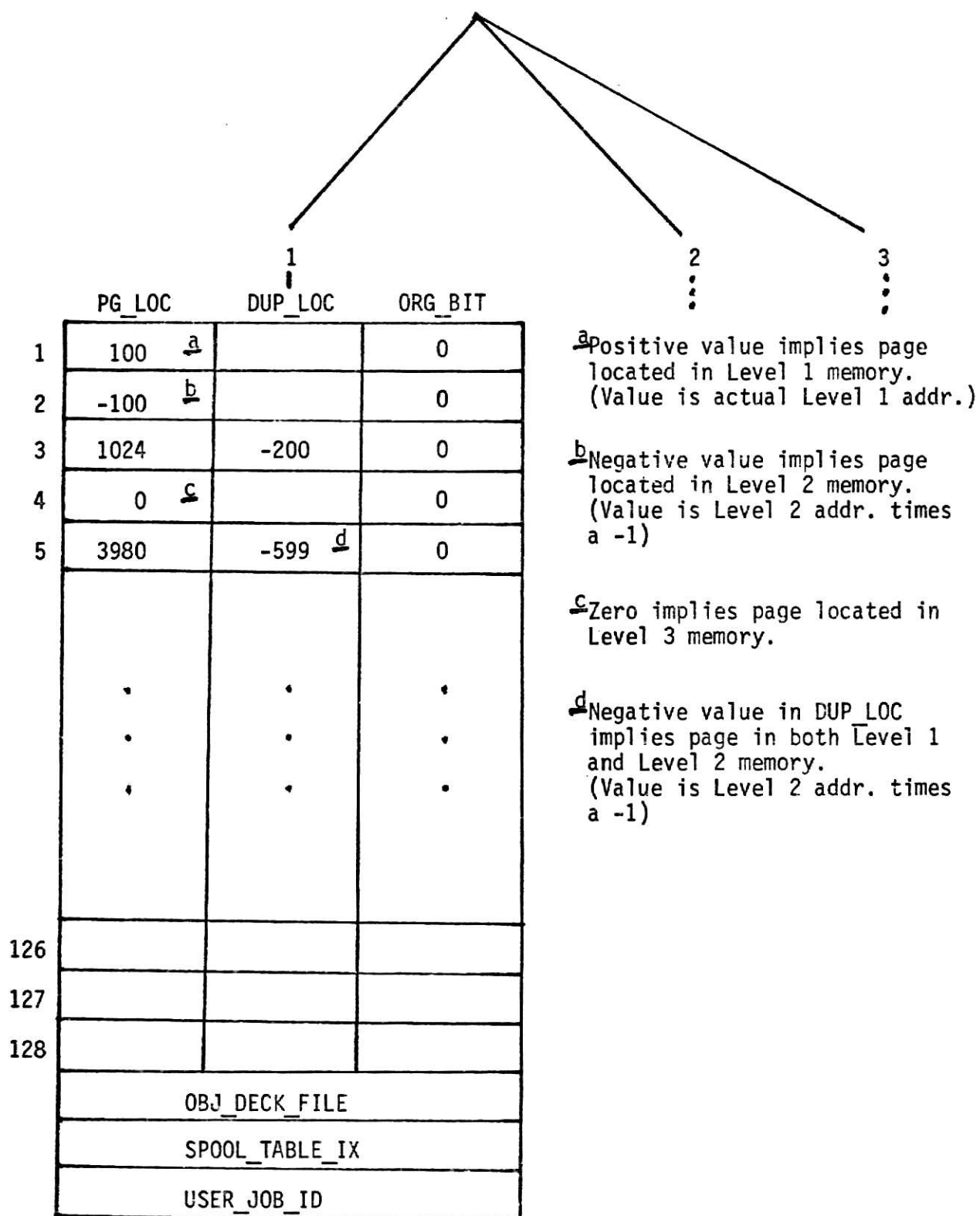


Figure 3-1

is needed is the user's System Job ID (SYS_JOB_ID), which will be either 1, 2 or 3, and the page number. The Nova then uses the system job ID as the index into the job page control block (JOB_PCB), which will produce the correct PMT, (i.e. JOB_PCB(SYS_JOB_ID), refer to data structure in algorithm 1, appendix A. Next a one is added to the requested page number and this value is used as the index to the page's PMT entries. A one is added to the page number to avoid zero as an index. Thus page zero is stored at index one. For example, if the requested page is page number 8 then, PG_NUM will equal 8, and JOB_PCB(SYS_JOB_ID).PG_LOC(PG_NUM+1), will be its corresponding entry location in the PMT.

Each PMT will then have three entries. The first entry, PaGe LOcation (PG_LOC), contains the necessary information needed to find a page located in the system. If the page is located in Level 3 memory, a zero is stored as its page location (PG_LOC). When the Nova is searching for a page and finds a zero as the PG_LOC, it then uses the FILE_ID for that job and passes it and the page number to the appropriate routine to retrieve the page. This page retrieval method is explained in section 3.25, with the discussion of "contiguously organized files" (16).

If the PG_LOC contains a negative number, this indicates the page is stored in Level 2 memory. The negative number is the address in Level 2 memory where the page is stored, that has been multiplied by a negative one. Thus in order to retrieve the page, the Nova multiplies PG_LOC by a negative one and uses this as the page's real address in Level 2 memory.

If the PG_LOC contains a positive number, this indicates the page is in Level 1 memory. The PG_LOC will then represent the address where it is stored in Level 1 memory. It can then be retrieved directly from the address given in PG_LOC.

The second entry, DUPLICATE LOCATION (DUP_LOC), is used by the paging subroutines. It is used to check to see if a copy of a page that is being paged out of Level 1 memory into Level 2 memory already has a copy residing in Level 2 memory. This will be explained in detail in section 3.10. This copy would exist from a prior page migration from Level 1 memory to Level 2 memory.

The third entry in the PMT is the ORIGINAL BIT (ORG_BIT). If it has a value of zero, it indicates the page is not an original. A value of one indicates the given page is an original.

The OBJECT DECK FILE (OBJ_DECK_FILE), is used to store the file ID of the object deck for a job. It is used by RDOS to do page retrieval.

SPOOL TABLE INDEX (SPOOL_TABLE_IX), contains the index into a job's file entries in the spool table.

USER JOB ID (USER_JOB_IDEN), contains the job's system created user job number.

The data base for the spool table is used to store a job's "USER ID" number and all of the files connected with the job. See chapter two, Figure 2-8. It will contain five entries. Entry one is where the user ID number is stored. This number will be created by the Nova at the time the job enters the system.

The second entry is the job's "SOURCE DECK FILE ID." If the job contains a source deck upon input, the file ID will be created

and entered in the table at that time. If no source deck is included, this entry will be ignored.

The third entry is the job's "DATA DECK FILE ID." If the job contains data, a file ID is created and entered at the time of job entry. If no data is present this entry will be ignored.

The fourth entry is the job's "OBJECT DECK FILE ID." If the job contains an object deck at the time of entry a file ID will be created and entered at that time. If there is not an object deck present, this entry will be ignored for the present. It will later be filled in by a subroutine which will create an object deck file ID for a job when requested to do so by the Interdata. This will happen during compilation of the source program.

The fifth entry in the spool table is the job's "OUTPUT FILE ID." This will be created by a subroutine when called by the Interdata. The PaGe TRAFFic TABLE (PG_TRAF_TAB), is used as a data base for part of the instrumentation. It is used to keep count of all page transfers between any two levels of memory, (i.e. Level 2 to Level 1 memory). There are eight counts keep. One for each possible exchange of memory plus two additional counts. The two additional counts, count page exchanges between Level 2 and Level 3 memory which are caused by requested I/O. These along with the total number of page faults will be printed upon termination of a job.

This subroutine also initializes the spoql table at system generation time. This amounts to setting the "USER ID" entry in the spool table equal to 0. When a job is entered into the spool

table, the spool table is searched for a zero user ID number. This then indicates an available entry for that job in the spool table.

The subroutine is interrupt driven by a message handler located in the Interdata. All messages are blocked from entering this subroutine until the previous message request has been fulfilled.

The function of the various calls in this subroutine will be explained in the routines that are called by them.

3.3 ALGORITHM 2: GENERATES THE SYSTEM

Algorithm number 2, appendix A, subroutine SYS_GEN, will be the first subroutine called by the Interdata at system generation time. Its main function is to calculate the free core memory in the Nova, the core not being used by the operating system, and to structure it in blocks of 256 words each. This then is known as Level 2 memory and is used as an extension to Level 1 memory in the Interdata for paging. Its second function is to call the initial program for the page option being used, which will initialize the data bases in the system. This will be explained in detail when the initialization subroutines are discussed. Upon completion of this routine the Nova system is ready for programs to be entered. A message is sent to the Interdata informing it that the Nova system has been initialized.

3.4 ALGORITHM 3: MAIN INPUT/OUTPUT DRIVER

Algorithm number 3, appendix A, subroutine INPUT/OUTPUT, is called on for all I/O operations. It contains the data bases used

for all I/O. Therefore all the other I/O subroutines are internal to this subroutine in order to save passing the data bases and redefining them in each subroutine called by this subroutine.

The data base consists of two parameter blocks, passed by the Interdata when I/O is requested. Also a storage block of 256 words is reserved for a page I/O buffer for pages located in Level 3 memory. This is necessary because in the executable form, the program is stored as a "contiguously organized file." For a detailed discussion of "contiguously organized files," see section 3.25. A contiguously organized file can only be accessed in blocks of 256 words each. In this case each block corresponds to a page. Thus in order to do I/O to a partial page or across page boundaries, the page, if in Level 3 memory, is moved to the buffer area. I/O is done to the corresponding addresses in the buffer and the page transferred back to Level 3 memory. See section 3.7 for a detailed discussion of the transfer of data from input files to the executing program and from the executing program to output files. The two parameter blocks passed by the Interdata are shown in Figure 3-2A and 3-2B. Looking at Figure 3-2A, the function code is an eight bit field which defines the type of I/O operation as defined in Figure 3-3. The Logical Unit (LU) is the device code for the requested I/O device. The status and device address are used to return the ending status upon completion of the I/O operation. If the Nova can complete the transfer as requested, it stores a zero in this 16 bit word. If anything is wrong with the device before the transfer or if anything goes wrong during the transfer, the

User Parameter Block

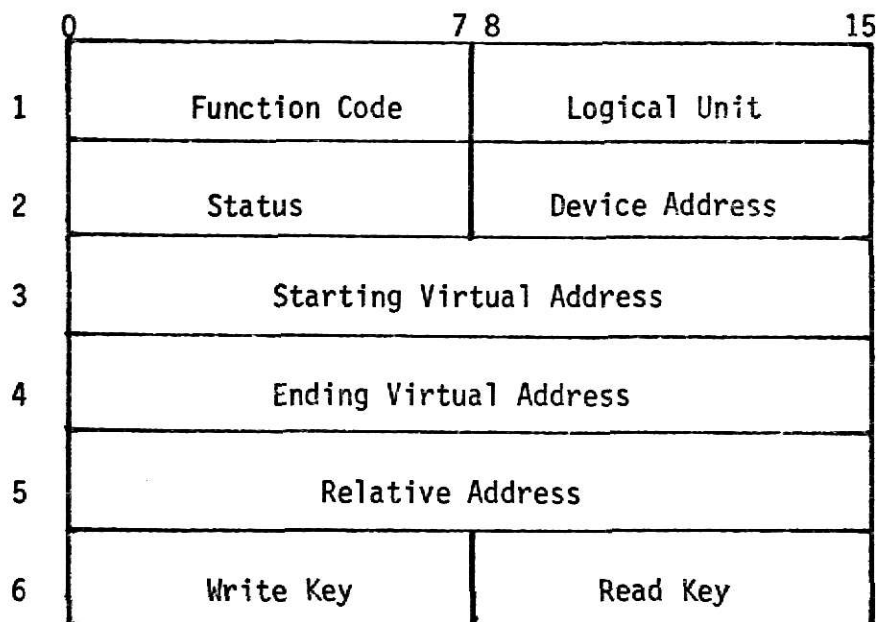


Figure 3-2A

System Generated Parameter Block

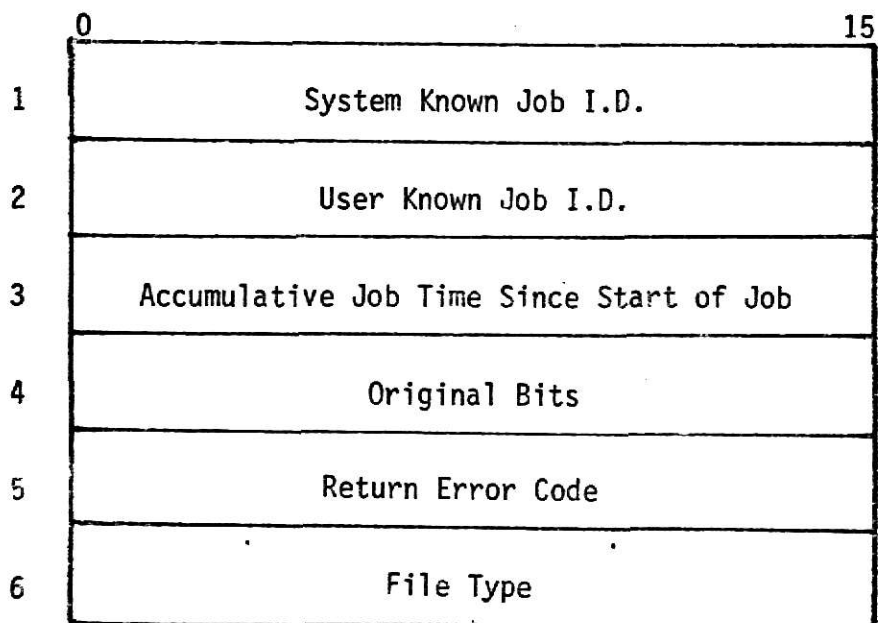


Figure 3-2B

Interdata Function Codes For I/O

OPERATION	BINARY	HEX
Write ASCII and Proceed	0010 0000	20
Write Random and Proceed	0010 0100	24
Write ASCII and Wait	0010 1000	28
Write Random and Wait	0010 1100	2C
Write Binary and Proceed	0011 0000	30
Write Binary and Wait	0011 1000	38
Read ASCII and Proceed	0100 0000	40
Read Random and Proceed	0100 0100	44
Read ASCII and Wait	0100 1000	48
Read Random and Wait	0100 1100	4C
Read Binary and Proceed	0101 0000	50
Read Binary and Wait	0101 1000	58
Test and Set Random	0110 0100	64
Test and Set	0110 0000	60

Figure 3-3

Nova stores in the first 8 bits the Interdata error code (see Figure 3-4), and the device address in the second 8 bits. This actually takes place in another subroutine explained in section 3.5 and section 3.9. The next two parameters are the virtual addresses which define the starting and ending address for the I/O. As noted in the Interdata OS/16-MT Reference Manual (17), the starting address should be on an even byte boundary and the ending address on an odd byte boundary. OS/16-MT uses an eight bit byte. This implies all I/O should transfer a minimum of 16 bits or in multiples of 16 bits. This is important since one word of Nova storage is 16 bits as opposed to 8 bits in the Interdata. The last two parameters are optional and will be ignored by the Nova.

Interdata I/O Error Codes

CONDITION	BINARY	HEX
Illegal Function	1100 0000	X'CO'
Device Unavailable	1010 0000	X'A0'
End of Medium	1001 0000	X'90'
End of File	1000 1000	X'88'
Unrecoverable Error	1000 0100	X'84'

Figure 3-4

The first word of the second parameter block (Figure 3-2B) contains the job name known by OS/16-MT at system generation time. The second word is the job ID created by the Nova when the job was spooled. The next word is the job's CPU time used to this point. This will be in microseconds. The next word contains the original bits in the order in which the pages appear in Level 1 memory. Word five, (RETURN_ERROR_CODE), will be used to convey error messages detected by the Nova. If, upon checking the status in the User Parameter Block (UPB), the Interdata finds some other value than zero, it can output to the job's output file a message, (i.e. "Nova I/O error") and then output the contents of word five of the SGPB, which will contain a code for the error message. Word five will have been set equal to register two of the Nova. RDOS contains a routine that returns to register two, an error code upon encountering an error during I/O. The user can then look this code up in the RDOS Manual to see what caused the error. Word six will contain the FILE TYPE (FILE_TYPE). There will be four file types. See chapter two, Figure 2-8. The (FILE_TYPE) will be passed by the Interdata so that the Nova can use it, along with the SPOOL TABLE Index (SPOOL_TABLE_IX), as an index into the spool table. The spool table index will have been obtained when the job was accessed from the job queue. This will provide the Nova with the correct file name, so that the given file can be accessed for the requested I/O.

Upon an I/O request message from the Interdata the Nova checks the beginning address to see if it is on an even boundary and the ending address to see if it is on an odd boundary. If not, it

assigns the unrecoverable error code to the status, X'84', and the physical address of the device requested to the device address, and returns the parameters to the Interdata. If the boundaries are correctly aligned, the Nova then calculates the number of 16 bit words to be transferred. This is done by subtracting bits 7-15 of the starting virtual address from bits 7-15 of the ending virtual address, adding one to the result and dividing the new result by two. For example if the starting virtual address is 508 and the ending virtual address is 511, the number of words to be transferred is two, $(511-508=3, 3+1=4, 4/2=2)$.

Next the starting and ending page numbers are calculated from the starting and ending virtual addresses. This is done by using the first 7 bits (bits 0-6) of the virtual address as the page number. For a detailed discussion of virtual address to real address translation in this system see the paper by Smith (5).

Next the function code is checked to see if it is an I/O and proceed. If so a message is passed back to the Interdata that the I/O has been started. The function code is again checked to see if the request is for input or output. If for input, the input driver is called. If for output, the output driver is called. These two routines will be explained in the next two sections.

If the function code is neither for input or output, the status is set to X'CO', illegal function, and the device address is set to the physical address of the device requested. The subroutine then returns.

3.5 ALGORITHM 4: INPUT DRIVER

The input driver, algorithm 4 appendix A, subroutine DATA_IN, is called whenever input is requested via the Input/Output driver. The routine first calculates the offset from the beginning of the starting virtual address. This will be used as the offset to be added to the real address once the real address has been calculated. This then is the beginning address where input will be transferred to. Next the requested Interdata function code, passed in the UPB, see Figure 3-2A, and 3-3, is converted to the equivalent Nova RDOS input command.

Next the starting and ending page number are checked to see if they are equal. If so, the subroutine TRANS_IN is called to perform the actual transfer. TRANS_IN will be explained in section 3.7. Passed with the call are the number of words to be transferred, calculated by the previous subroutine, and the offset from the beginning of the page that the data is to be transferred into.

If the input area is split across page boundaries, see Figure 3-5, then the TRANS_IN subroutine has to be called once for each page and or partial page. The offset and number of words has to be calculated prior to each call and the page number incremented by one each time. For example, if the input area is split across three pages, as seen in Figure 3-5, the number of words to be transferred the first time is calculated by subtracting bits 7-15 of the starting virtual address from 512, the number of bytes per page, and dividing the result by two, to give the number of 16 bit words. For the example in Figure 3-5, this would be $(512-510=2, 2/2=1 \text{ word})$.

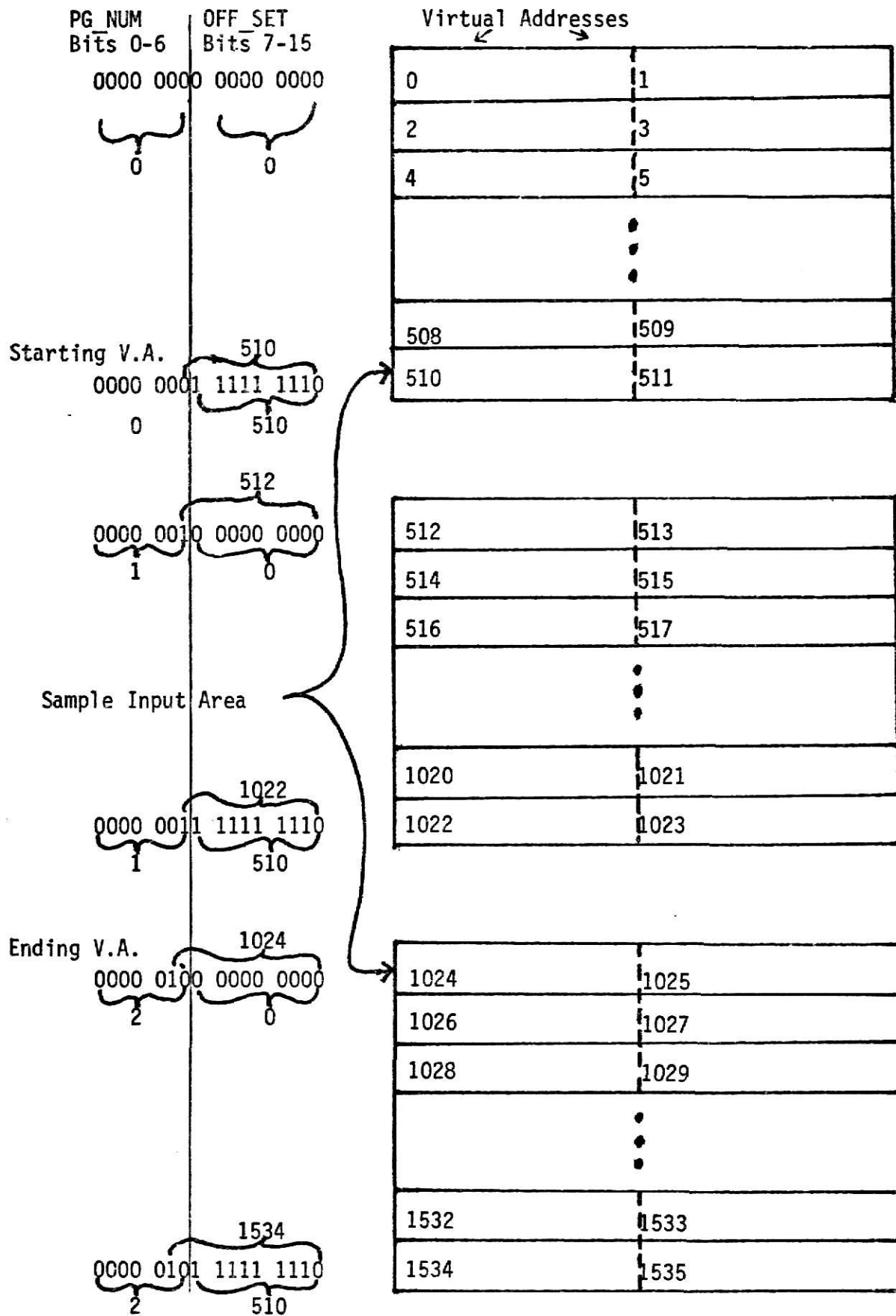


Figure 3-5

This number is then passed in the call to TRANS_IN and the first partial page of the input is transferred.

The number of words just transferred is then subtracted from the total number of words to be transferred. For the example in Figure 3-5, this is $(258-1=257)$. If this new value is greater than or equal to 256 words, then a one is added to the page number and a call is made to TRANS_IN again, with the offset equal to zero and the number of words equal to 256. The offset is zero since a full page is being transferred. This is repeated until the number of words to be transferred is less than 256.

When this happens, if the number of words to be transferred is greater than zero, the page number is again incremented by one and TRANS_IN called, with an offset of zero and the remainder of the words to be transferred in as the other parameter. For the example Figure 3-5, this will be a one.

If no I/O errors occurred during transfers, then the routine returns. If an I/O error has occurred then the error code conversion subroutine is called. This subroutine will be explained in section 3.9.

3.6 ALGORITHM 5: OUTPUT DRIVER

This subroutine, see appendix A algorithm 5, is identical to the subroutine just described in section 3.5 with the exception it calls the subroutine TRANS_OUT instead of TRANS_IN, which handles output instead of input. TRANS_OUT will be explained in section 3.8. The two subroutines could be combined into one, but for modularity purposes they have been kept separate.

3.7 ALGORITHM 6: TRANSFERS DATA TO INPUT ADDRESS

This subroutine, see appendix A algorithm 6, is called by the Subroutine DATA_IN, described in section 3.5, to do the actual transferring of input data. Its main purpose is to locate the page the data is to be transferred into, use the passed offset to map to the first word of the input area and then to transfer the requested data starting at this address.

The first thing the subroutine does is to obtain the file name of the input file. The file is obtained by indexing into the spool table. To reference the correct file it needs to know, the index to this job's spool table entries, the system job ID, and the file type.

The index to the job's spool table is known as it has been stored in the main procedure as (SPOOL_TABLE_IX) when the job was retrieved from the job queue, see algorithm 25, appendix A. The SYSstem JOB ID (SYS_JOB_ID), and the FILE TYPE (FILE_TYPE) have been passed as parameters in the SGPB, see algorithm 3, appendix A. The file type is either type 2 for source deck, or type 3 for data deck. Type 2, source deck, is used during compilation as the source deck is then the input data. The file types correspond to the files location in the spool table--see chapter two, Figure 2-8 (i.e. file type 3 is data deck). The file name is passed to RDOS when the actual read is requested.

Next the job's page table is checked for the location of the page in memory. As explained in section 3.2, a positive value is Level 1 memory, a negative value indicates it is in Level 2 memory

and a zero indicates it is in Level 3 memory.

If the value in the PMT is positive this represents the page's real address location in Level 1 memory. The OFFSET (OFF_SET) is then added to this address and the number of words to be transferred in, are transferred beginning at this address.

If the requested page is located in Level 2 memory, indicated by a negative PaGe LOcation (PG_LOC) value in the PMT, this value must then be multiplied by a negative one and the result used as the real address of the beginning of that page in Level 2 memory. The offset is then divided by two, added to that address (PG_LOC) and the number of words requested transferred in, beginning at this address. The offset has to be divided by two since the offset is calculated using half words, since each half word is addressable in the Interdata. Dividing the offset by two and adding it to the virtual address of a page composed of 256, 16 bit words will produce the equivalent address that would result from adding the original offset to a virtual address composed of 512, 8 bit words. See Figure 3-6. After the transfer is complete, the original bit for this page is set to one. The original bit did not have to be set in the prior example since the Interdata set its original bit for the page.

If the page is in Level 3 memory, the page first has to be transferred to Level 2 memory before input can take place. This is because any input will be to the object deck file, and this file is stored as a "Contiguously Organized File" so that it can be accessed randomly for page transfers. For a detailed description

Example of Page Mapping

Bits 7-15 of Virtual Address

1	2
0 0000 0000	0 0000 0001
3	4
0 0000 0010	0 0000 0011
5	6
0 0000 0100	0 0000 0101
7	8
0 0000 0110	0 0000 0111
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
508	509
1 1111 1100	1 1111 1101
510	511
1 1111 1110	1 1111 1111

Interdata
Level 1 MemoryBits 7-14 of Virtual Address
(Has effect of dividing by 2)Corresponding
Addresses

↔

↔

↔

↔

Same Relative
LocationSame Relative
Location

↔

↔

0	
0 0000 000	
1	
0 0000 001	
2	
0 0000 010	
3	
0 0000 011	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	

Nova
Level 2 Memory

Figure 3-6

of a "Contiguously Organized File," see Figure 3-7 and section 3.25. The object deck file ID is stored in the main procedure when the Interdata first requested use of this file, see algorithm 27, appendix A. This file name is then retrieved and sent with a request to transfer a page from Level 3 memory to Level 2 memory. It is transferred into (PG_BUFF) which is an array of 256 words reserved for this purpose. The algorithm that transfers the page from Level 3 to Level 2 memory will be explained in section 3.18, see algorithm 17, appendix A. After the page has been transferred into PG_BUFF, the offset is divided by two and one added to the result. This is because the page addresses run from 0 to 127 and PG_BUFF, addresses run from 1 to 128. This value is then used as the index at which location the requested data is read into for the given number of words. The page is then transferred back out to its virtual address in Level 3 memory.

3.8 ALGORITHM 7: TRANSFERS DATA TO OUTPUT FILE

This subroutine given as algorithm 7, appendix A, is almost identical to the algorithm just described in section 3.7 with several minor exceptions. First it does output instead of input. Second, as a result of the first, the file type (FILE_TYPE) will always be type 5, output. Since the file is always output the original bits do not have to be reset each time.

3.9 ALGORITHM 8: TRANSLATES NOVA I/O ERROR CODE

This subroutine, see algorithm 8 appendix A, is called upon when an error has been encountered during I/O. It is used to

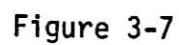


Figure 3-7

translate Nova error code messages to the error codes that the Interdata operating system can recognize. This is done so that at the completion of an I/O instruction the Interdata knows if the I/O was successful or not. RDOS automatically returns an error code in accumulator two if anything goes wrong during I/O. This code will be passed to this subroutine when called, and the subroutine will match it to the equivalent Interdata error code. Any code that can not be matched will be set to the "unrecoverable" Interdata error code. The codes the Nova can pass are given in Figure 3-8. The Interdata codes were presented in Figure 3-4. The illegal function code is not included here, because if it occurs, it will be detected in algorithm 3, prior to this subroutine.

3.10 ALGORITHM 9: PAGE OPTION 1 MAIN ROUTINE

This subroutine, see algorithm 9 appendix A, is used to implement page option one. Page option one is based on a competitive variable working set size applied on a local scale. This option has been described in section 2.2.2 of chapter two.

The data base for the subroutine consists of three identical substructures which make up the Working Set Control Block (WSCB), one for each possible job in the system, and a stack which will contain the AVAILABLE PaGe FRAMES (AVAIL_PG_FRAME). The WSCB will contain three variables and a substructure array. The first variable, PREVIOUS ACCumulative TIME (PREV_ACC_TIME), will record the job's total CPU time at the point of the job's previous page fault. This value is initially set to zero, so upon the first page

AC2	Explanation of Error
0	Illegal Channel Number
3	Illegal Command For Device
6	End of File
7	Attempt To Read A Read Protected File
15	Attempt To Reference File Not Open
26	File Read Error
30	Attempt To Read Into System
33	File Accessible By Direct Block I/O Only
47	Simultaneous Reads On Same QTY Line
74	Address Outside Address Space
101	Ten Second Time-out Occurred
106	QTY/MCA Input Terminated By Channel Close

Figure 3-8

fault it will have a value of zero. The next variable, PREVIOUS TIME SPAN (PREV_TIME_SPAN), is used to record the elapsed time that has occurred by a job between the two previous page faults. For example, if the last three page faults occurred at time T_{i-2} , T_{i-1} , and T_i , then PREV_TIME_SPAN for the current page fault occurring at T_i is $T_{i-1} - T_{i-2}$. PREV_TIME_SPAN will have been recorded in the previous call to this subroutine. Initially it will be set to zero. This will cause the requesting job to be given a page frame in Level 2 memory upon the first page fault if there is one available. The next variable, HEAD, is used as a pointer to point to the top entry in the array used for the Extended Page Fault Table (EPFT). The EPFT structure is used to implement the LRU algorithm in the Nova and for instrumentation page level depth counts in the LRU stack. Instrumentation depth counts will be explained in detail later in this section.

This subroutine is called by the Interdata when it has a page to be paged out of its Level 1 memory in order to free a page frame for its next page fault. Upon entry the subroutine first stores the page number that is to be paged out of Level 1 memory into Level 2 memory in a temporary location. This is to preserve its passed value so that it can be entered in the EPFT at the close of this subroutine. The original value may be changed in the mean time, depending upon which path is taken through this subroutine. Next the CURRENT TIME SPAN (CURRENT_TIME_SPAN) is calculated. This is the job's CPU time that has elapsed since the last page fault occurred to level 2 memory. This will be used later.

Next the subroutine checks to see if the page being paged out of Level 1 memory already has a copy in Level 2 memory. This could have resulted from a previous page fault. This is done by checking the given page number's DUPLICATE LOCATION (DUP_LOC) in the PMT. If this value is less than zero, then it has a copy in Level 2 memory. DUP_LOC will always be zero unless there is a copy in both level 1 and Level 2 memory. The negative value in DUP_LOC, is the given page's Level 2 real address that has been multiplied by a negative one.

If there is a copy of the page already in Level 2 memory, a check is made to see if the page being paged out of Level 1 memory is an original. If it is, then the page is recopied into the available page frame in Level 2 memory, by a call to the Level 1 TO Level 2 subroutine (L1_2_L2_PG). See algorithm 20, appendix A. The page's page map table entries are updated in the L1_2_L2_PG subroutine also. Next the page frame that the old copy is in, is multiplied by a negative one to get its real Level 2 address. This then becomes the next available, or current, page frame to be used for the next page fault. The page's DUP_LOC in the PMT is then set to zero, since there is no longer a copy in Level 1 memory.

If the page being paged out of Level 1 memory does have an identical copy in Level 2 memory then it does not need to be recopied. In either case, the Nova next sends the Interdata the page transfer complete message. Next the index of the page that was transferred from Level 2 to Level 1 memory is located in the LRU stack (EPFT). This index represents the depth at which the page had obtained in the LRU stack prior to being referenced again in

Level 1 memory. A count for each depth that has been reached by a page in the LRU stack prior to the page being requested back to Level 1 memory is kept. It is accumulated at this point in the algorithm. If the job being executed displays good locality, then the depth counts should monotonically decrease as the stack depth increases. See Figure 3-9. The LRU stack is next compressed to fill the extracted page's page map table entries. The updated page's page map table entries will later be placed on the top of the stack. Next the job is checked to see if it is a candidate for a reduction in its working set size. This is done by multiplying its previous time span by 110% and comparing it against its current time span. If its current time span is greater, implying the time interval between the last page fault and the previous page fault is greater by at least 10%, then the working set size will be reduced by one page frame, providing it has at least two pages currently in Level 2 memory. This is counting the page just paged-in. If the job is a candidate for removal and a page can be removed, then a call to the internal subroutine, `Decrease Working Set Size (D_CREASE_WSS)`, is made. This subroutine will be explained later in this section. An increase in working set size does not have to be checked here because the above path implies that an additional page frame is not needed as a result of the page fault. This is due to the fact that there already is a page frame available for the page migration if needed, since the given page existed in Level 2 memory prior to the page fault. If this path has been taken the working set control block (WSCB) time

Stack Depth Counts

LRU Stack (EPFT)

	PG_NUM	DEPTH_COUNT
First Page Entered in Stack →	40	0
	20	5
	10	4
	9	12
	100	24
	14	20
	13	20
	16	30
	17	29
	18	40
Last Page Entered in Stack →	19	55

Increasing Stack Depth ↑

Increasing Depth Counts ↓

Figure 3-9

variables are updated and the page that was paged out of Level 1 memory is placed on top of the LRU stack. The routine now returns.

The following sequence of events is executed as a result of Level 2 memory not containing a copy of the page being paged out of Level 1 memory. First the page is copied from Level 1 memory to an empty page frame in Level 2 memory. This empty frame was made available either in the initial routine, or as a result of a prior call to this subroutine. The Interdata is then sent a message saying that the transfer has been completed. This is so it can continue processing this task even though the Nova has more cleanup work to do on the page fault request.

The variable size working set calculations are next applied. This will determine if the requesting job's working set is entitled to gain a page, lose a page or remain the same. The current time span, calculated at the start of this subroutine, is multiplied by 90% and compared against the elapsed time span that occurred previous to the one just calculated. If the current time span is less than 90% of the previous time span, this implies a faster paging rate. As a result the job is entitled to an additional page frame. The available page frame list is checked, and if one is available, it is used to replace the one taken as a result of the current page fault. Next the pointer to the top of the LRU stack is advanced one so that when the page number of the page that was just transferred in Level 2 memory is entered on the LRU stack, it will point to the top. The LRU stack being the EPFT.

If there was not an available page frame, then the job does not receive another page, even though it was entitled to one. It is not allowed to pre-empt a page from another job.

At this point either the job was entitled to another page frame and there was not one available, or else it was not entitled to one. In either case, the job is checked to see if it is a candidate for a reduction in its working set size. This calculation is the same as the one described in a previous paragraph for page reduction following the path where there was already a copy of the page in Level 2 memory. If the page is a candidate for removal and a page can be removed, then 'N' is set to two. If not, 'N' is set to one. 'N' is a loop variable used to indicate how many times the internal subroutine that removes a page from Level 2 memory is to be executed. Providing the subroutine reached this point of execution, one page will always have to be removed. This is because at this point, a page has been added, the one being paged in, and none have been removed. If a job's working set size is to be reduced by one, then two pages will have to be removed for the same reason as given above.

The following internal subroutine is used to remove a page from the job's working set. First the page number on the top of the LRU stack is retrieved. Its original bit is then checked. If a one, then the subroutine to recopy it back to Level 3 memory is called. If the original bit is a zero then it is not recopied. In either case the page frame index is incremented by one and the vacated page frame added to the list of available page frames.

The page's PMT PG_LOC is then set to zero to represent that its only copy is now in Level 3 memory. After the above is completed, either once or twice, the CURRENT_PG_FRAME is set to the page frame currently on top of the stack of available page frames. Next the pointer to the top of the LRU stack is reduced by zero or one, depending upon if one or two pages were removed. It will then point to the page at the top of the LRU stack, which will be added to the stack just prior to returning. This will be the page number of the page just paged into Level 2 memory. The decrease working set size subroutine now returns to the point of call. The WSCB time variables are updated as they were described for path one, and then the subroutine returns.

3.11 ALGORITHM 10: PAGE OPTION 1 INITIAL ROUTINE

This subroutine, see algorithm 10 appendix A, is called to initialize the data base used in algorithm 9, just described in section 3.10. It is first called by the system generation subroutine, algorithm 2, and thereafter by the end of job routine, algorithm 16, at the close of a job. The variables are therefore initialized for a new job entering the active state.

The subroutine first resets all the page level transfer counts to zero. Next it checks the job's PMT for all pages contained in Level 2 memory. As it finds a page, the page frame index is incremented by one and the page frame put back on the available list. Also the DUP_LOC is set to zero in case the page was located in both Level 1 memory and Level 2 memory. Along with this each PG_LOC in the job's PMT and each LRU depth count is reset to zero. The subroutine then returns.

3.12 ALGORITHM 11: PAGE OPTION 2 MAIN ROUTINE

This subroutine, see algorithm 11 appendix A, is used to implement page option two. Page option two, a competitive variable working set size based on a global scale, has been described in section 2.2.2.

The data base for the subroutine consists of the Working Set Control Block (WSCB), which contains the Extended Page Fault Table (EPFT), the AVAILABLE PaGe FRAMES (AVAIL_PG_FRAME), the index into the available page frames and a pointer (HEAD), pointing to the latest entry in the EPFT. The EPFT is used to implement the LRU paging algorithm in the Nova for Level 2 memory. Since this algorithm is applied on a global scale, it is necessary to have two entries in the EPFT. The first entry is the Job ID (JOB_IDEN), which indicates whose page it is. The second entry (PG_NUM), is the actual page number of the page it represents. The AVAILABLE PaGe FRAME array (AVAIL_PG_FRAME), contains all of the starting addresses of the empty page frames in Level 2 memory. AVAIL_PG_F_INDEX is the current index into the preceding array.

This subroutine is called by the Interdata when it has a page to be paged out of its Level 1 memory. The parameters passed to the routine include the SYStem JOB ID (SYS_JOB_ID), which will be either 1, 2 or 3, the job accumulation time to the point of interrupt (JOB_ACC_TIME), the page number of the page to be paged out of Level 1 memory (PG_NUM), the Level 1 address where this page is located (L1_ADD), and the Interdata original bit set for this page (IORG_BIT). The JOB_ACC_TIME is not used by this subroutine but is included in the call so as to make all three page option subroutines call-able

by the same call routine. This is to make the subroutines compatible so that they are interchangeable.

Upon entry this subroutine first assigns the page number to a temporary variable. This is to preserve its original value to be entered in the EPFT at the end of the subroutine. Depending upon which path is taken through the subroutine, the page number may or may not be changed.

Next the page being transferred from Level 1 to Level 2 memory is checked to see if it has an existing copy already in Level 2 memory. This is done by checking DUP_LOC for a value of less than zero. DUP_LOC will always contain zero unless the page has a copy both in Level 1 and Level 2 memory. If it has a copy in Level 2 memory, the value contained in DUP_LOC will be its real level 2 address that has been multiplied by a negative one. If a copy already exists in Level 2 memory, then the page being transferred out of Level 1 memory is checked to see if it is an original, (i.e. IORG_BIT=1). If it is an original, then it has to be recopied back into Level 2 memory. This is done by calling the Level 1 to Level 2 subroutine. The page map table is also updated in the called subroutine. See algorithm 20, appendix A. The vacated page frame's starting address is then set to the current page frame to be used for the next page fault. Next the job's DUP_LOC in the PMT is set equal to zero since the page no longer has a copy in both Level 1 and Level 2 memory. The Interdata is then sent a message that the page has been transferred so that it can continue processing the task, even though the Nova has not completed its cleanup of the page migration.

If the original bit had been a zero, then the page being transferred out of Level 1 memory would have been identical to the copy already in Level 2 memory. This would save the transfer time and the Interdata would have been sent the transfer completed message right away. Next the Nova searches the EPFT for the job's page number. Since the EPFT is a global pool of pages from all three jobs, both the PAGE_NUM and JOB_IDEN have to be compared. Once the correct entries have been found, "I" will represent the depth in the LRU stack where the page is located. The depth count at this location is then incremented by one. Next the PAGE_NUM and JOB_IDEN are removed from their location in the LRU stack, (EPFT) the stack is compressed and then the entries re-entered at the top of the stack, prior to the subroutine returning.

If after checking DUP_LOC for a copy of the page and finding that a copy does not exist in Level 2 memory, then a call to transfer the page from Level 1 memory to Level 2 memory can be made immediately. Upon completion of the transfer, the Interdata is sent the transfer complete message. The page map table is updated in the L1_2_L2_PG subroutine, algorithm 20, appendix A.

Next, a check is made to see if there are any available page frames left in Level 2 memory. If so, the next available page frame is assigned to the CURRENT_PG_FRAME. It will then receive the next page transfer when the subroutine is called again. The pointer to the top of the LRU stack is incremented by one, so as to continue to point to the top of the stack.

If there was not an available page frame for the next transfer,

then a page has to be transferred out of Level 2 memory in order to make one. The page at the bottom of the stack, since it has been resident the longest, is removed from Level 2 memory. It first is checked to see if it is an original page. If so it is recopied to Level 3 memory. If it is not an original, it is not recopied. Next its PMT PG_LOC is set to zero to indicate that its only copy is in Level 3 memory. The vacated page frame's address is then assigned to the current page frame to be used for the next request. Next, the LRU stack is compressed to fill the removed page entry.

Before returning, the page that was paged out of Level 1 memory is entered, along with its JOB_ID, at the top of the LRU stack.

3.13 ALGORITHM 12: PAGE OPTION 2 INITIAL ROUTINE

This subroutine, see algorithm 12 appendix A, is called to initialize the variables used in algorithm 11 described in section 3.12.

It first resets all of the instrumentation page migration counts to zero. It then searches the extended page fault table for all the pages belonging to the job being deleted. If an entry (page) in the EPFT being checked belongs to the job being deleted, the page frame that the page occupies is put back on the list of available page frames. As these entries are deleted the EPFT is compressed to fill the entries being deleted. See Figure 3-10. After the deletions, the pointer to the top of the LRU stack (HEAD) is reset to the value at the top of the stack so as to point to the

Extended Page Fault Table (EPFT)

Showing before and after effect of
deleting a job from the system.
(i.e. Job 2)

SYS_JOB_ID	PG_NUM	ORG_BIT		SYS_JOB_ID	PG_NUM	ORG_BIT
1	14	0		1	14	0
2	2	1	Deleted	3	3	0
2	20	1	Deleted	1	1	1
3	3	0		3	2	1
1	1	1		1	40	0
2	3	1	Deleted	3	14	1
3	2	1		1	1	0
1	40	0		1	18	0
3	14	1		3	17	1
1	1	0		3	1	0
1	18	0		1	15	1
2	3	1	Deleted	1	9	1
2	14	0	Deleted	3	9	1
3	17	1		1	39	1
3	1	0				
1	15	1				
2	7	0	Deleted			
1	9	1				
3	9	1				
1	39	1				
2	13	0	Deleted			

Before

After

Figure 3-10

least recently used page. The subroutine then returns.

3.14 ALGORITHM 13: PAGE OPTION 3 MAIN ROUTINE

This subroutine, see algorithm 13 appendix A, is used to implement page option three. Page option three, is based on a non-competitive fixed partitioned working set size as described in chapter two, section 2.2.2. Algorithm 13 is basically the same as algorithm 11, described in section 3.12. The only difference between the two are several minor changes in the data structure. Instead of one structure composing the Working Set Control Block (WSCB), there are three identical substructures, one for each job. The substructures are as described in section 3.12, with these exceptions. One, the EPFT only needs a single entry for the page number since each job is allocated a private WSCB. Also, since the algorithm is applied locally, each job must keep a separate variable for assignment to the CURRENT PaGe FRAME (CURRENT_PG_FRAME). The MAXimum PaGe Frame (MAX_PG_F), number is assigned at initialization time to the number of page frames a job will have in Level 2 memory for its use. This is done in algorithm 2, see appendix A. The logic of the algorithm is the same as algorithm 11 in section 3.12. Again the difference between the two algorithms is that in algorithm 11, the available page frames are in a global pool. In this algorithm, algorithm 13, the available page frames are in a local pool contained in the given job's fixed partition.

3.15 ALGORITHM 14: PAGE OPTION 3 INITIAL ROUTINE

This subroutine, see algorithm 14 appendix A, is used to

initialize the variables used for page option 3, algorithm 13. It is first called by algorithm 2 during system generation time. It is thereafter called by the "END OF JOB" routine, algorithm 16, at the close of a job. This then leaves the variables initialized for the next job.

The subroutine first resets all the page counters back to zero. Next it sets all the PaGe LOCations (PG_LOC), in the job's page map table to zero, to represent that the page is in Level 3 memory. This is because all jobs start out located in Level 3 memory. All DUPLICATE LOCations (DUP_LOC) and depth counts are also reset to zero. The subroutine then returns.

3.16 ALGORITHM 15: RETRIEVE PAGE

This subroutine, see algorithm 15 appendix A, is called by the Interdata whenever it encounters a page fault. The subroutine first goes to the page map table and obtains the page's address and or location. Remember a negative number here indicates Level 2 memory and a zero, Level 3 memory. The page has to be in either Level 2 memory or Level 3 memory else the Interdata would not have requested the page. Once the page has been located it is transferred to the requested address in Level 1 memory and this Level 1 address is then entered into the PMT as the current location for the transferred page.

If a copy of the page existed in Level 2 memory prior to the page fault, the negate of its Level 2 address is entered in the page's duplicate location (DUP_LOC). This indicates that a copy of the page now exist in both level 1 and Level 2 memory. It will

be used later when a page migration takes place between Level 1 and Level 2 memory. If there are duplicate copies in Level 1 and Level 2 memories then when a page migration occurs from Level 1 to Level 2 memory the page does not have to be recopied, providing the copy in Level 1 memory is not an original. A message is then sent to inform the Interdata of the transfer and the subroutine returns.

3.17 ALGORITHM 16: END OF JOB

This subroutine, see algorithm 16 appendix A, is called by the Interdata each time a job terminates. It first totals the page counts kept for instrumentation. Next the I/O counts are subtracted from their respective level counts so as not to appear in the counts caused by page faults. This would result because the level counts are taken in the page transfer subroutines, algorithm 17 thru 23, and both page fault routines and I/O request use these algorithms. The counts with appropriate headings are then added to the job's output file. If requested the source program for the job is then spooled to the output device. It is followed by the command to spool its data output file. The job's files are then deleted from Level 3 memory by RDOS. Next the job's entry into the spool table is released. The Interdata is then sent a message that the job has completed and the subroutine returns.

3.18 ALGORITHM 17: LEVEL 3 MEMORY TO LEVEL 2 MEMORY

This subroutine, see algorithm 17 appendix A, is used to transfer a page from Level 3 memory to Level 2 memory. Its main

use is to bring a page into Level 2 memory so that I/O can be performed on the page. The page is transferred into a 256 word array called PG_BUFF.

3.19 ALGORITHM 18: LEVEL 3 MEMORY TO LEVEL 1 MEMORY

This subroutine, see algorithm 18 appendix A, is used to transfer a page from Level 3 memory into Level 1 memory. It is used by the Interdata to get a page from the user's source file into its Level 1 memory via the PAGE_IN subroutine, algorithm 15. After the page has been transferred, its residing address in Level 1 memory is entered in the page map table for future reference. Its original bit is set to zero and the subroutine returns to the point of call.

3.20 ALGORITHM 19: LEVEL 2 MEMORY TO LEVEL 3 MEMORY

This subroutine, see algorithm 19 appendix A, transfers a page from Level 2 memory to Level 3 memory. It is called by the page option subroutines when Level 2 memory becomes saturated and a page frame is needed for future page faults in Level 2 memory.

3.21 ALGORITHM 20: LEVEL 1 MEMORY TO LEVEL 2 MEMORY UNDER OPTIONS 1 AND 2

This subroutine, see algorithm 20 appendix A, is used under paging options one and two to transfer a page from Level 1 memory to Level 2 memory. After the page has been transferred, the original bit associated to its Level 2 memory address is "OR'ED" to its original bit it had in the Interdata. This is because any page that is original in a higher level of memory, is also original to each

lower level of memory. It might be noted here that it is necessary to "OR" the two bits together rather than to just set the level 2 original bit equal to the level 1 original bit. This is because the page may have migrated between Level 1 and Level 2 memory several times. If for example on the third migration the page did not get changed in Level 1 memory, but had been an original in Level 2 memory earlier, the Interdata would now indicate an original bit of "0" for the page. Thus if the original bit in the page map table is set equal to the Interdata's original bit, the fact that the page is still an original to Level 3 memory will be lost. Next the job's page map table (PG_LOC) entry is set equal to the Level 2 address of the page frame that it was recopied into and then multiplied by a negative one. This indicates it is now located in Level 2 memory.

3.22 ALGORITHM 21: LEVEL 1 MEMORY TO LEVEL 2 MEMORY UNDER OPTION 3

This subroutine, see algorithm 20 appendix A, transfers a page from Level 1 memory to Level 2 memory when the paging is performed under option three.

It is identical to algorithm 20 with the exception that the page frame (CURRENT_PG_FRAME), to be used for the next transfer needs to be qualified for the particular task. This is because the available page frames are kept in a task or local pool as opposed to a global pool.

3.23 ALGORITHM 22: LEVEL 1 MEMORY TO LEVEL 3 MEMORY

This subroutine, see algorithm 22 appendix A, is used to

transfer a page from Level 1 memory to Level 3 memory. As the Nova routines stand now it is never called. It is included for future possibilities of rollin and rollout to be discussed in chapter four.

3.24 ALGORITHM 23: LEVEL 2 MEMORY TO LEVEL 1 MEMORY

This subroutine, see algorithm 23 appendix A, is used to transfer a page from Level 2 memory to Level 1 memory. Since the transfer is a core to core transfer, all that is needed is the two core locations. The Interdata passes its core location. The Nova uses the SYS_JOB_ID and PG_NUM to index into the page map table to find its core location. RDOS is then called upon to perform the transfer.

3.25 ALGORITHM 24: INPUT SPOOLING

This subroutine, see algorithm 24 appendix A, is used to spool all user jobs from the card reader to the disk. It first checks the job spool table to make sure there is room to spool another program to Level 3 memory. The spool table is set for 100 entries, but can be changed to fit the environment of the operation. If the spool table is full, a message to that effect is sent back to the Interdata and the subroutine returns. If there is an available location in the spool table, a job ID is created for the incoming job and entered into the spool table. Next the JCL is checked for the presence of a source deck. It might be noted here that as a job is read in, the JCL is stripped off, later to be entered in the job queue with the job's ID number. This is necessary so that the Interdata can later reference the JCL when the program enters the

execution state. If a source deck is present, a unique file name for it is created and entered in the spool table. The source deck is then read into a "sequentially organized file," as defined by RDOS (16), see Figure 3-11. Sequentially organized files reserve the last word of each 256 word block for use as a pointer to the next block of 256 words. With sequentially organized files RDOS handles all of the buffering automatically. This is the reason for using sequential files for I/O. Next the JCL is checked for the presence of an object deck. If an object deck is present a unique file name is created for it and then entered in the spool table. The object deck is then read into a "contiguously organized file", as defined by RDOS. Contiguously organized files are files whose blocks may be accessed randomly and are composed of a fixed number of disk blocks which are located at an unbroken series of disk block addresses (see Figure 3-7). Since the data blocks are at sequential logical block addresses, all that is needed to access a block within a contiguous file is the address of the first block (or the name of the file) and the relative block number within the file. This relative block number within the file will correspond on a one for one basis to the page number. Thus if page four of the object deck is requested, the file name and relative block number (4), are sent as parameters and RDOS will retrieve page four. This is the reason for using contiguously organized files for source decks. Next the JCL is checked for the presence of any data. If the job contains data, a data file name is created and entered in the spool table. The data is then read into this "sequentially organized file". After all the files have been read in, the job's

Sequentially Organized Files

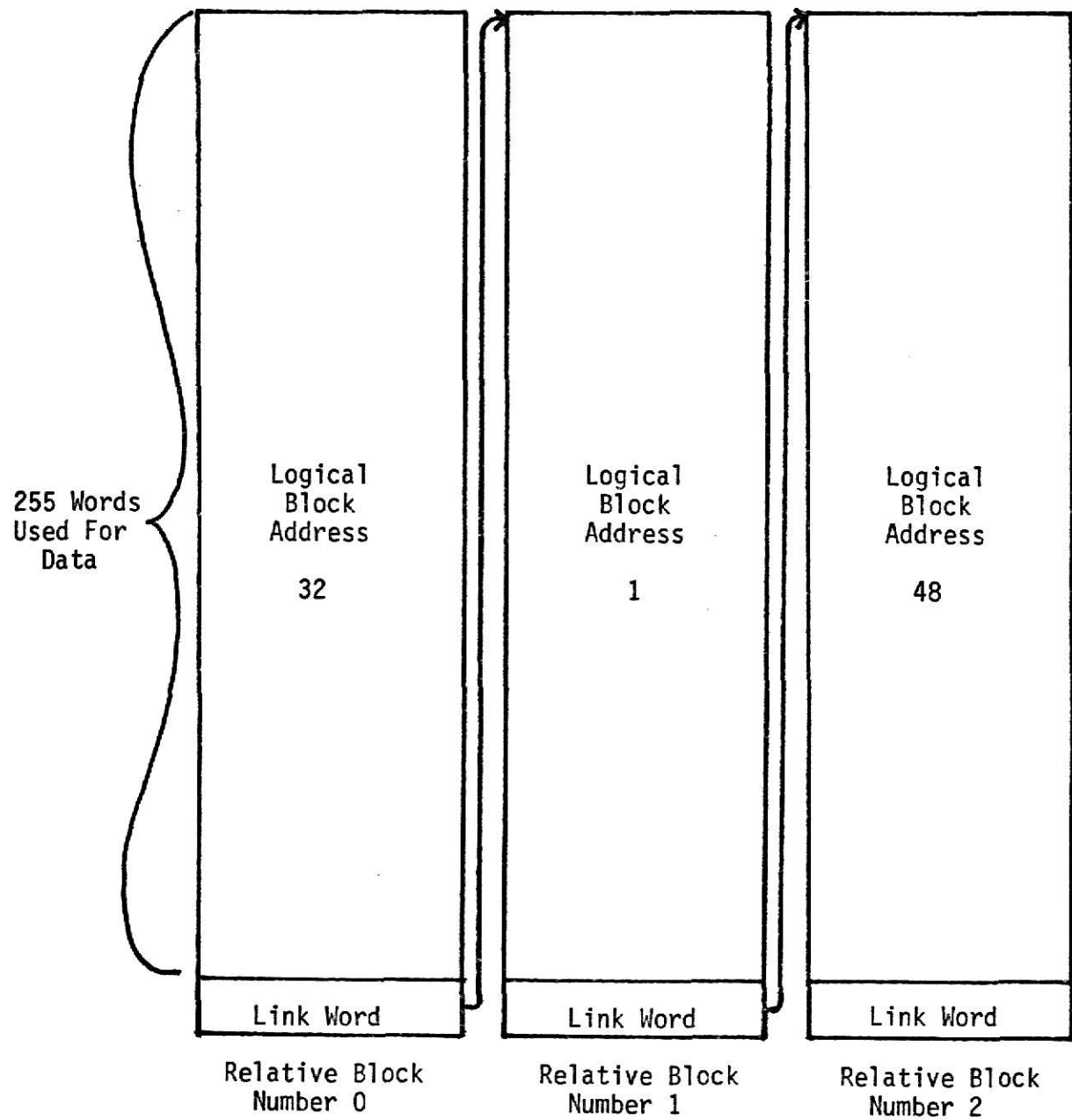


Figure 3-11

job number, spool table index, and JCL are entered into the job queue to await call for execution.

3.26 ALGORITHM 25: JOB QUEUE SEARCH

This subroutine, see algorithm 25 appendix A, searches the job queue for the next job to be run. It is called by the Interdata. If the job queue is empty, a message saying so is returned. If not the job's ID in the job queue and its JCL are sent to the Interdata. Next the job's index into the file table is taken from the job queue and stored in the job's PCB in the Nova. This is so the Nova can index into the job's spool table entry for its various files when needed. The subroutine then returns.

3.27 ALGORITHM 26: OBJECT DECK FILE NAME

This subroutine, see algorithm 26 appendix A, is used to create an object deck file name when requested by the Interdata. After it has been created it is entered in the job's spool table entry. This subroutine will get called by the Interdata prior to a source program being compiled. This is necessary since the Nova handles all I/O files in the system and in the system's view the object deck produced is a form of I/O.

3.28 ALGORITHM 27: RETRIEVE OBJECT DECK FILE NAME

This subroutine, see algorithm 27 appendix A, is used to retrieve a job's object deck file name by the Interdata. It is necessary for the Interdata to have a job's object file name prior to requesting its first page fault. This is because the file name is used when retrieving pages from Level 3 memory. Therefore if

the Interdata has not used algorithm 26 to produce an object deck file name as a result of compilation, it must retrieve the one that was created as a result of the object deck being read in with the user's program deck.

3.29 ALGORITHM 28: CREATE OUTPUT FILE NAME

This subroutine, see algorithm 28 appendix A, is used to create the output file name where a user's output will be stored. This routine is called by the Interdata prior to doing output. After the file name has been created, it is entered in the spool table for future reference.

3.30 ALGORITHM 29: LIST STACK DEPTH COUNTS FOR PAGE OPTIONS 1 OR 3

This subroutine, see algorithm 29 appendix A, is called by the Interdata whenever a stack depth count is desired while running under page option one or three, as described in section 3.10, for a given job. This process will actually involve two calls. One to this routine and the other to the I/O driver.

First a call is made to this routine. This routine then transfers the requested job's LRU depth counts to the page buffer in Level 2 memory that has been reserved for an I/O work buffer. After the transfer, the depth counts are reset to zero. Next the starting and ending address of this I/O work buffer that is being used are obtained. The subroutine then returns, passing these addresses to the Interdata. The Interdata then has to issue a call for I/O and at which time the requested counts are transferred to the job's output file to be printed at the termination of the job.

3.31 ALGORITHM 30: LIST STACK DEPTH COUNTS FOR PAGE OPTION 2

This subroutine, see algorithm 30 appendix A, is identical to algorithm 29 just described with the exception that the depth count variable does not have to be qualified. This is because the stack count is global and not local. It is used in place of algorithm 29 while running under page option 2.

CHAPTER FOUR

4.1 INTRODUCTION

In this chapter we will discuss possible future modifications to the HIMICS system. We will conclude with a summary of the HIMICS system.

4.2 SYSTEM MODIFICATION

Because of the modular design of this system, future changes can easily be made. The following suggested modifications are intended to make the present system design into a more desirable and sophisticated system. Some of the suggestions may or may not be able to be incorporated into a given system. This will obviously depend upon the mini-computers being used and the sophistication of their operating systems.

The original operational design of the system was designed with simplicity in mind. This was done so as to get an operational version of the system running in as short of time period as possible. As a result some rather primitive techniques have been used. Further restrictions, as mentioned above, were related to the characteristics of the operating system being used. The main restriction here being that there is no dynamic allocation of in-core memory.

4.2.1 I/O FILES

One obvious objection to the present system is that the system is only set up to handle one input and one output file per

job. This is limited by the fixed size of the spool table. (See Figure 2-10, chapter two.) This can easily be changed by a modification to the spool table. Either the spool table can be expanded as an array similar to its present structure, or perhaps a more desirable alternative would be to construct the spool table from a linked list. This way the spool table can be allocated on the basis of need.

Another change dealing with I/O concerns the locking in of pages in Level 1 memory while I/O is being performed. The algorithms given in chapter three, locks pages in in Level 1 memory for both input and output. They need to be locked in for output but only the starting and ending page or pages need to be for input.

The alternative would be to only lock in the starting and ending page(s) on input. Data to the starting and ending page(s) would then be transferred directly to their Level 1 address. All input to the page(s) lying between the starting and ending pages, if any existed, would be input to Level 3 memory. These pages would then be transferred back to Level 1 memory upon demand.

There may be an advantage in doing this in that it frees up some Level 1 memory for other task. This advantage may however be offset. This would result from having to have an extra routine to search the extended page fault table and putting the released page(s) found in Level 1 memory back on the list of available page frames. This in itself should not be much of a disadvantage as it is handled by the Level 2 processor and does not take CPU time away from the host machine. The main disadvantage would extend from the LRU theory itself. That is, the working set contains those pages

which are most likely to be referenced next. It would appear that if a request is made for input to a page or pages already in the working set, that that would tend to increase the possibility that those pages will be referenced soon. If this is the case it would appear that locking the pages in and doing I/O directly to their Level 1 address would save time in the long run. This time being the time to process the page faults to bring the pages back into Level 1 memory from Level 3 memory. Only actual test runs can be used to answer which way system performance would be optional.

4.2.2 ROLLIN AND ROLLOUT

Rollout (1,15) involves swapping a program from main memory onto secondary storage. Rollin (1,15) is the process of bringing the program back into main memory. It differs from paging in that the whole program is swapped, not just one page.

Under the present system, Level 1 memory is divided into fixed partitions which are set at system generation time. A future consideration would be to apply the variable working set option (page option 1, section 2.2.2 of chapter two) to Level 1 memory. If this were done, it could become beneficial at times to rollout a job in Level 1 memory if paging becomes excessive. The excess memory created by rolling out a job could then be redistributed to jobs currently in Level 1 memory to reduce an excessive paging rate. Later when the paging slowed down, the job rolled out could be rolled back in. Also one might want the rollout and rollin capabilities to handle high priority jobs. With rollout and rollin

capabilities, high priority jobs would enter the executable state upon entry into the system, by rolling out a lower priority job if one existed.

4.2.3 MULTI-TASKING

As the system is generated in its present state it will handle up to three jobs in a multi-tasking environment. A more desirable system generation procedure would be one to include a parameter that would generate the number of partitions, or level of multi-tasking the user desired. To do this efficiently would involve dynamic allocation of the various tables used by the file management and paging routines. If this were done, the number of tables needed for any number of jobs could be allocated with a minimal use of memory.

4.2.4 PRIORITY

Another desirable feature would include some form of priority system. This way when jobs are spooled to the disk a priority for that job could be entered into the system spool table. Then when the job scheduler picks a job to run, the job with the highest priority would be picked. This system could also have some scheme where jobs that have equal priorities are executed according to which job has been spooled the longest.

4.2.5 PARAMETER PASSING FOR SUBROUTINE INDEPENDENCY

The original intent in the design of this system was to make each subroutine completely independent of all other routines. The reason was to enhance system modification. As can be seen in

appendix B, C and D, this was not strictly followed. This could be corrected by parameter passing between subroutines, to produce the independency desired. This may be enhanced by some restructuring of the existing data structures.

4.3 TESTING OF ALGORITHMS

All of the algorithms given in appendix A have been hand simulated. This was done by generating all of the possible requests that can be sent by the Interdata, and calling the routines necessary to process the request. As each routine was called, the routine was executed by hand, instruction by instruction. A "variable" trace of the result of executing each instruction was kept. As the instruction was executed the instruction was marked in order to identify that it had been executed. It is necessary to mark the instructions as some routines have multiple paths of execution depending upon input parameters. These then have to be re-simulated with different input parameters until each instruction has been executed.

The hand simulation was done by writing all of the data structures and variable names out on a sheet of paper. As a subroutine was called and executed, a variable trace through the subroutine was made. That is, as each variable changed value, its new value was recorded below its previous value and this value then used as its current value. The results at the completion of execution of the subroutine were then checked against the expected results. If they match, the subroutine was accepted as performing its assigned task.

For an example we will look at algorithm 15 in appendix A. In order to check algorithm 15, all the algorithms that precede it in calling sequence have to be simulated first. This is necessary in order to initialize the data structures it will use. We will assume here that this has been done. The initialized values of the variables that will be accessed by algorithm 15 are given in Figure 4-1A. These are the values obtained by executing preceding algorithms in a normal calling sequence.

The next step is to construct some logical input parameters for the subroutine. For this example the subroutine requires three input parameters. We will let them have values as follows: `SYS_JOB_ID=2`, `PG_NUM=2`, `LI_ADD=6144`. The next step is to write down the asserted values the routine should have upon completion of execution with the given input parameters. These values are shown in Figure 4-1C.

The hand execution as described above is next performed. As the instructions are executed they are marked off as in Figure 4-1B. The variables that change are shown crossed off in Figure 4-1D, along with the new values written in below them. The new results, upon completion of execution, are then compared to the asserted results in Figure 4-1C. Since they match, we are confident that the algorithm performed as expected.

Looking back at Figure 4-1B, it can be seen that not all the instructions have been executed. Therefore a new set of input parameters are generated to cause the flow of execution through the subroutine to take the unexecuted path. For this example we will let the new input parameters have the following values --

Algorithm Testing

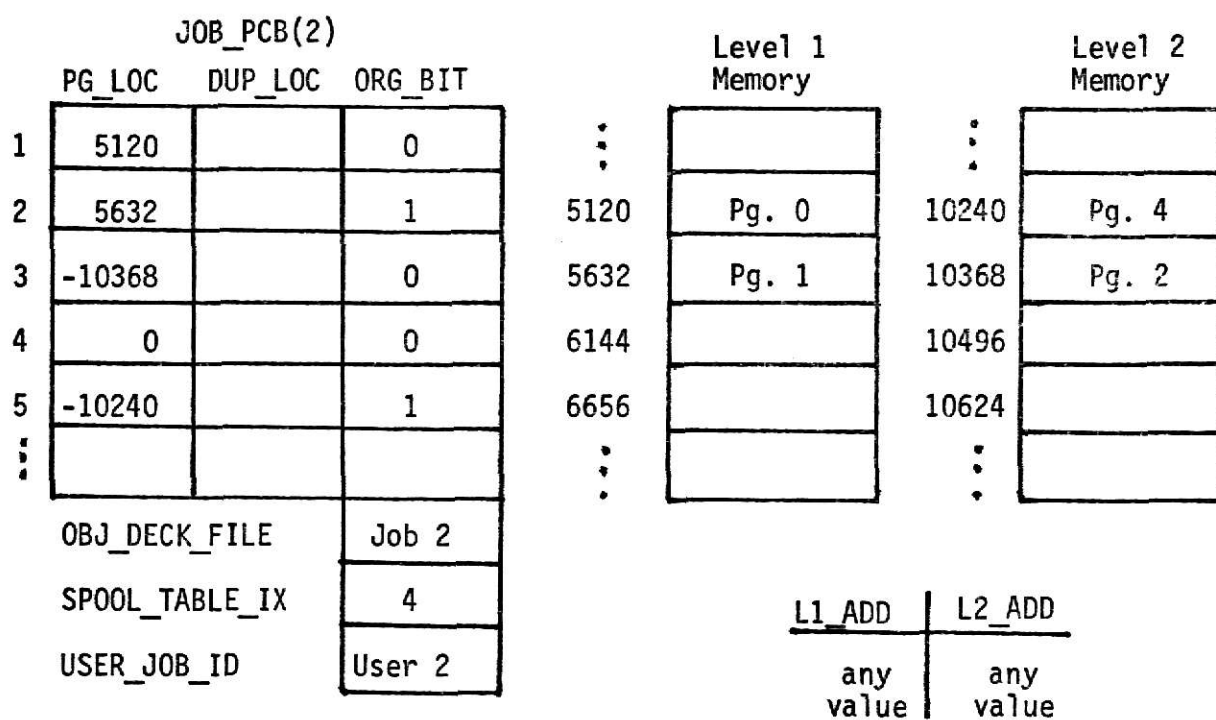


Figure 4-1A

Algorithm Testing

ALGORITHM 15: RETRIEVES PAGE REQUESTED BY INTERDATA

```
SUBROUTINE PG_IN(SYS_JOB_ID,PG_NUM,L1_ADD);
```

```
IF JOB_PCB(SYS_JOB_ID).PG_LOC(PG_NUM+1)<C THEN
```

```
DO;
```

```
CALL L2_2_L1(SYS_JOB_ID,PG_NUM,L1_ADD);
```

```
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LOC=
```

```
    JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC;
```

```
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC=L1_ADD;
```

```
END;
```

```
ELSE
```

```
DO;
```

```
FILE_ID=OBJ_DECK_FILE(SYS_JOB_ID);
```

```
CALL L3_2_L1(SYS_JOB_ID,PG_NUM,FILE_ID,L1_ADD);
```

```
END;
```

```
SEND INTERDATA MESSAGE "PAGE TRANSFERRED."
```

```
RETURN;
```

```
END;
```

Statements Executed For First Example

Statements Executed For Second Example

Figure 4-1B

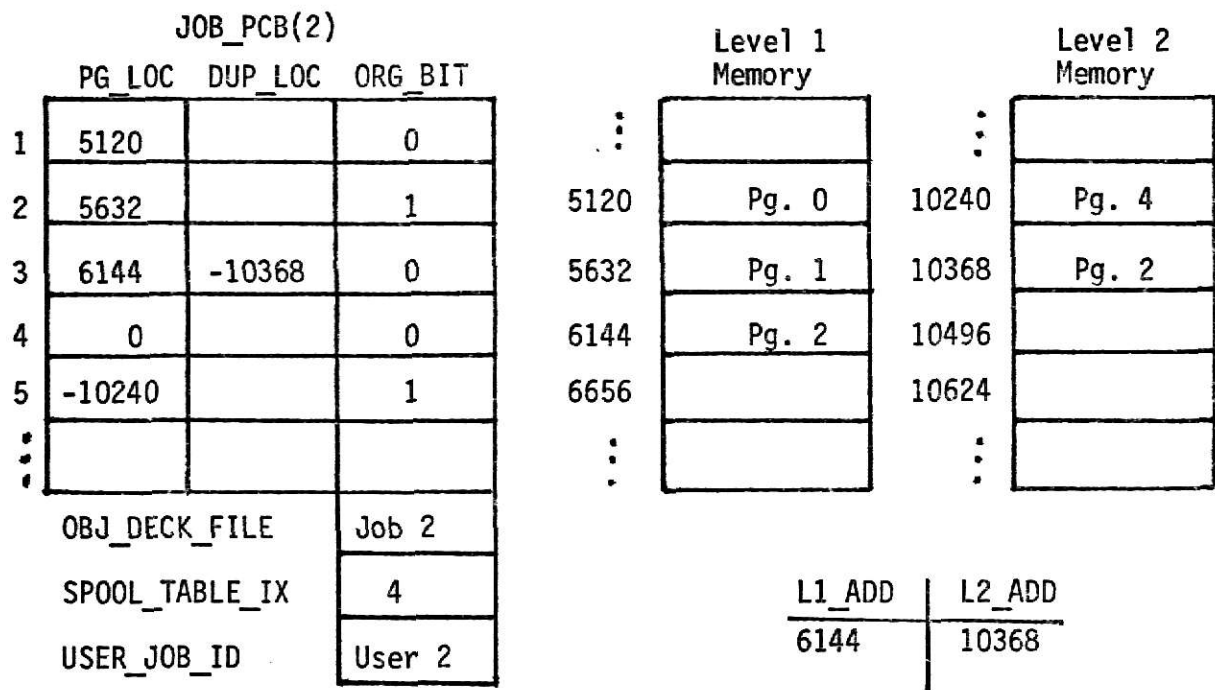
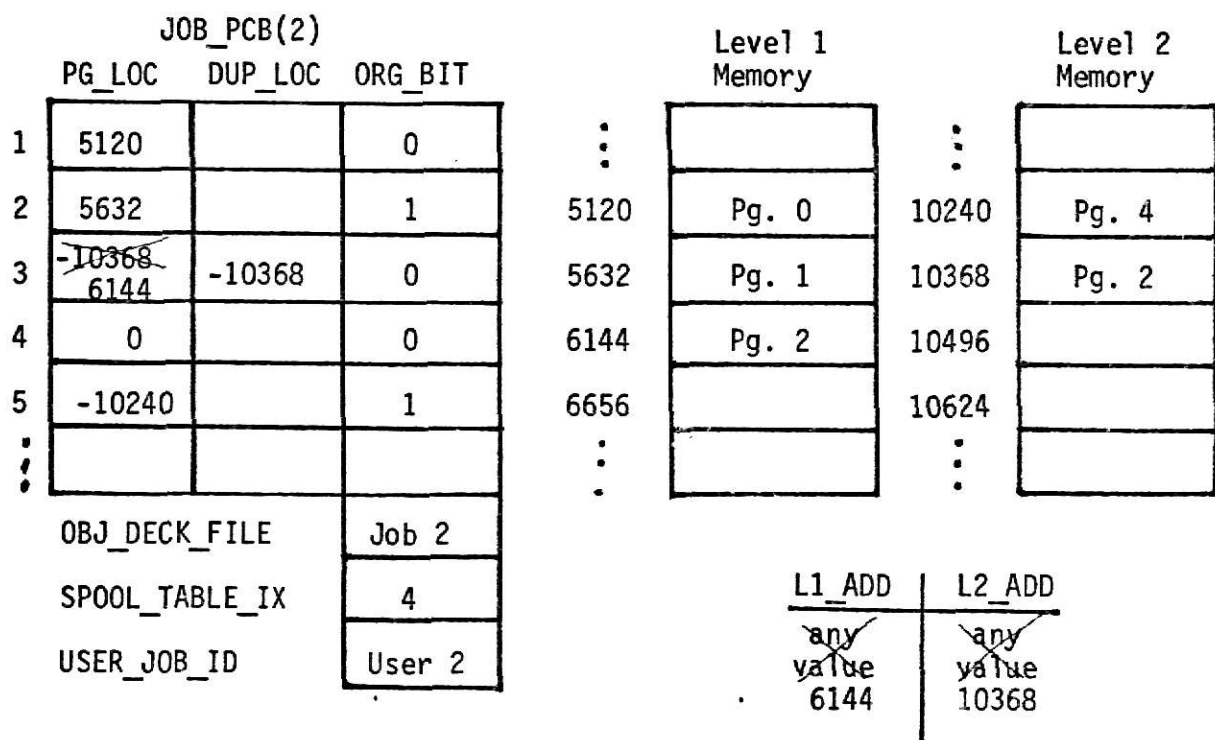


Figure 4-1C



Input: SYS_JOB_ID=2, PG_NUM=2, L1_ADD=6144

Figure 4-1D

SYS_JOB_ID=2, PG_NUM=3, LI_ADD=6656. The asserted values are given in Figure 4-1E. The values after execution are given in Figure 4-1F. Again they are as expected. Looking at Figure 4-1B it can be seen that all instructions have been executed. This completes the check out for algorithm 15. All the other algorithms have been verified in a like manner.

4.4 CONCLUSION

Since this system has not been implemented as of this writing, no statistics of system performance can be given. It would appear conceivable that a system as presented in this paper should increase throughput compared to a similar paging system running with only one CPU.

4.4.1 THEORETIC TIME ADVANTAGE

For a theoretic comparison we will take two mini-computer systems. Let one system consist of a single Interdata model 85 modified to run in a virtual environment. This means the system will have incorporated in it a paging system and will use primary and secondary memory. Primary memory being in-core memory in the host machine and secondary memory being disk storage. For this example an IBM 2315 disk cartridge is used. This is because the Interdata OS/16-MT Manual furnishes reference times for this particular disk drive. Let the other system consist of an identical machine as just described, but add to this system a second mini-computer. For this example we will use a Nova computer. The Nova in this case is used to handle all of the I/O and paging for the

Algorithm Testing

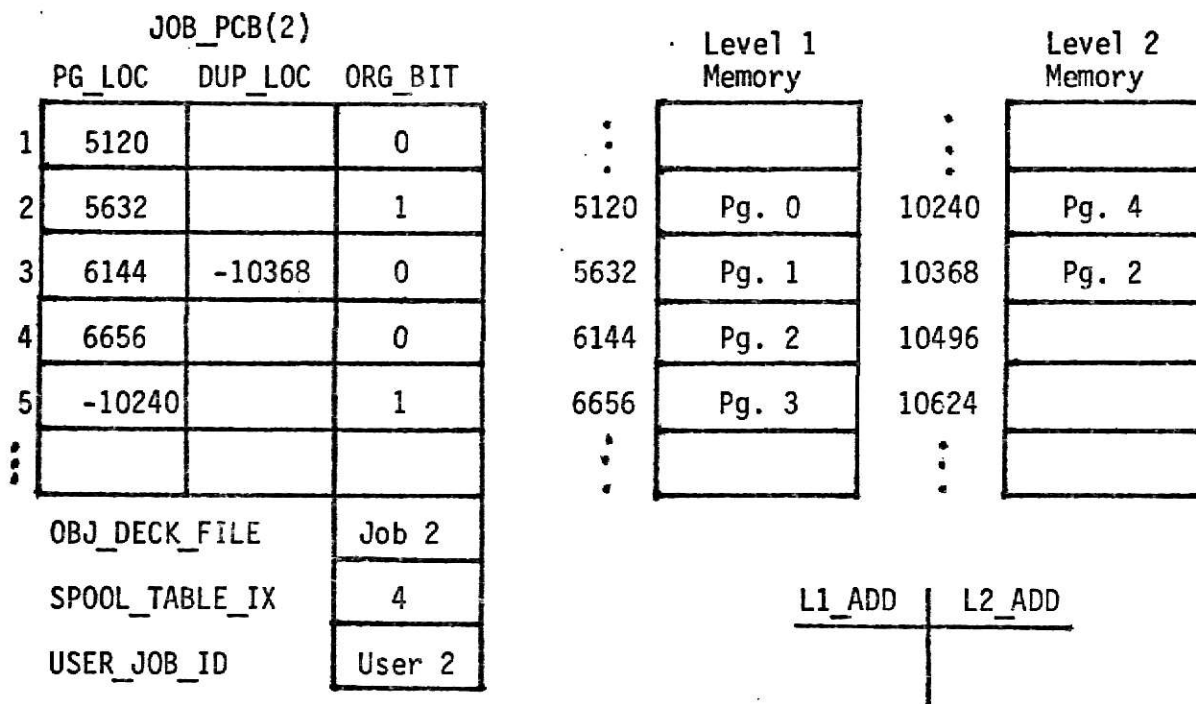
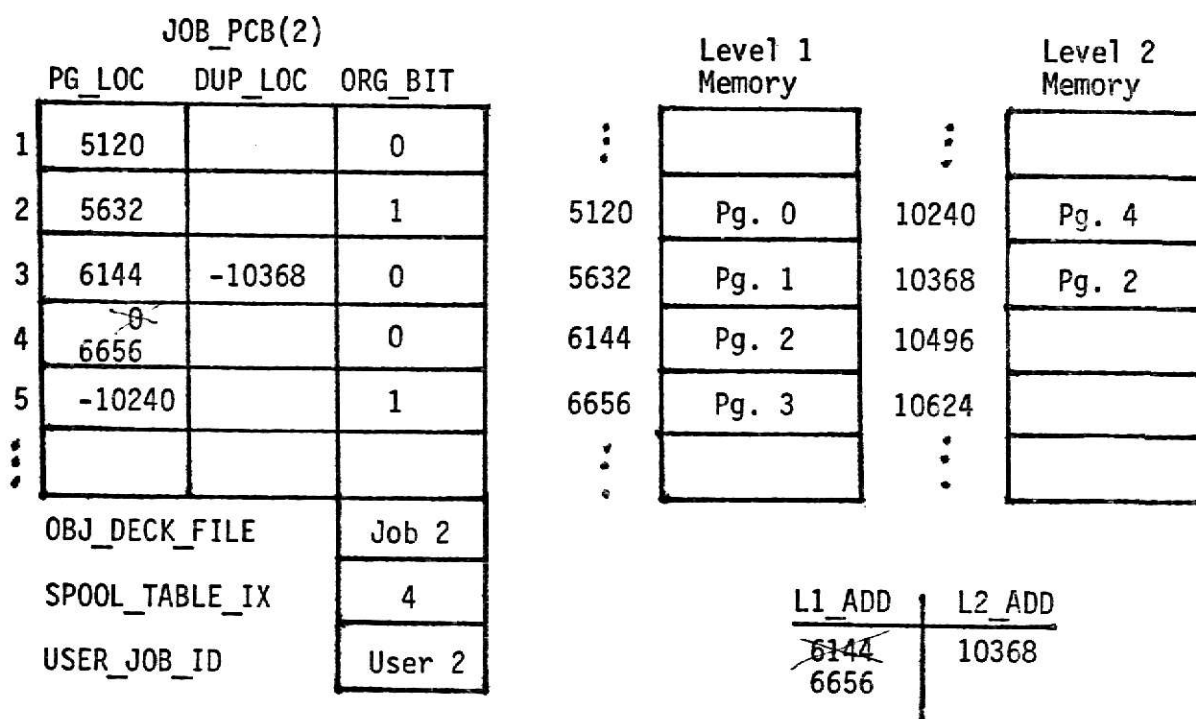


Figure 4-1E



Input: SYS_JOB_ID=2, PG_NUM=3, L1_ADD=6656

Figure 4-1F

host machine and to function as an extension to the host machine's primary memory. Therefore for the second system we will have three levels of memory. This will represent the system as presented in this paper. For this example let both systems be running under a maximum load. That is, each system has a backlog of programs waiting to be run. Let both systems be generated to multi-task among three user jobs at a time. Since each system is running under a maximum load, there will be three user jobs currently competing for the CPU'S time in the multi-task environment. Now let a page fault occur in each of the two systems.

Under the first system, with only one CPU, the task that was executing has to be put in the wait state until the requested page can be fetched. The time breakdown of the minimum events that have to occur are as follows. First an immediate interrupt occurs. This takes an average time of 4.42 microseconds. Next an interleaved data channel read is issued taking an average time of 1.85 microseconds. Since secondary storage is on disk, this requires a disk access. The average disk latency time is 20 milliseconds or 20,000 microseconds. The average head positioning and settling time takes another 70 milliseconds or 70,000 microseconds. Then the actual data transfer time takes 2.77 microseconds. This comes to a total of 90,009.04 microseconds. This would be the minimum time required for any page fault. The reason it is a minimum time is because for the times given it is assumed that primary memory is still unsaturated. A more realistic figure to use for comparison would be one that included two page transfers. This is because

once primary memory becomes saturated, for each page transferred into primary memory, a page will have to be transferred back to secondary memory. The minimum additional times required to transfer a page to secondary memory are as follows. First an interleaved data channel write is issued requiring 2.07 microseconds. Again this requires disk access, so the latency time of 20 milliseconds and the disk seek time of 70 milliseconds are included. The data transfer time for a page is again the same 2.77 microseconds. The above adds an additional 90,004.84 microseconds to the previous minimum time of 90,009.04 microseconds. This gives an average time of 180,013.88 microseconds to process a page fault with the single CPU once primary memory is saturated. The above figures are tabulated in Figure 4-2.

Running under system two, the system as presented in this paper under the same set of conditions as given above, the following time is taken away from the host computer's CPU. The average time required to process an immediate interrupt, which is 4.42 microseconds. This is because once the interrupt has been processed, the host machine can continue processing other task. This is made possible as a result of parallel processing utilizing the second CPU. The second CPU will handle all of the required page traffic while the host machine continues to process other task. This parallel processing in reality frees the host CPU to be used as a dedicated arithmetic and logic unit. Thus we are looking at a page fault rate time for a system as presented in this paper as 4.42 microseconds as compared to a similar single CPU virtual

Minium Page Fault Time For Single CPU

Page-in	Milliseconds	Microseconds
Immediate Interrupt		4.42
Data Channel Read		1.85
Disk Latency Time	20	20,000.00
Disk Head Seek	70	70,000.00
Data Transfer		2.77
<u>Sub Total</u>		<u>90,009.04</u>
Page-out		
Data Channel Write		2.07
Disk Latency Time	20	20,000.00
Disk Head Seek	70	70,000.00
Data Transfer		2.77
<u>Sub Total</u>		<u>90,004.84</u>
<u>Total</u>		<u>180,013.88</u>

Figure 4-2

address mini-computer system page fault rate time of 180,013.88 microseconds. This represents a savings of 180,009.46 microseconds per page fault or a 99% plus savings in time. During the additional time period of 180,009.46 microseconds required by the stand-a-lone CPU, 178,227 addition instructions could have been performed in the dual CPU system using parallel processing. This not only holds true for page faults but also I/O, since all I/O is handled by the second CPU in a like manner to paging.

This level of parallel execution obviously can not be met in a two machine two level processor system. It could be achieved if the sum of the distance (time) between page faults and I/O interrupts across all tasks is greater than the time required to process one page fault or I/O interrupt. This can be represented as: $\sum_{i=1}^N T_i \text{ interrupt} > T_i \text{ process interrupt}$. This implies that there is always a task in the ready state whenever an executing task is interrupted. Since the process times of present day machines can not meet this demand, "N" number of processors would have to be incorporated into a system network to keep up with the host machine.

4.4.2 SUGGESTED SYSTEM LOADS FOR SYSTEM PERFORMANCE EVALUATION

The actual performance gain of one system over the other can only be proved by the actual running of programs under the two operating systems and then comparing the execution times. Four basic types of programs should be included in the test. The first being small programs requiring little or no paging or I/O. Running under these conditions both systems would be expected to

perform about the same. The dual processor should be a little faster as all programs will require I/O to read in the program and for listings. The second type of programs should consist of small programs containing lots of I/O processing. This should indicate the advantage of parallel processing to perform I/O operations. The third type of program should consist of large programs designed to induce paging, but requiring little I/O processing. This should indicate the advantage of parallel processing to handle page faults. The fourth type of programs should consist of large programs to induce paging and also designed to induce I/O processing. This should indicate the advantage of parallel processing to handle both I/O and paging. The above types of programs can be run in homogeneous and heterogeneous groupings and evaluations made from the results.

4.4.3 COST OF MEMORY

The cost of hardware is often a determining factor in the design of a computer system. With the design given in this paper a slower speed processor can be used as extended memory and to handle the I/O and paging, thus reducing the high cost of high speed in-core memory, and giving the system added capacity. In this paper the Nova, used as the level two processor, is four times cheaper per given core memory size as the host machine. Thus using a multi-processor system more memory can be purchased for a given price and still meet the needs of the overall system.

A P P E N D I X A

ALG. #	ALGORITHM INDEX	PAGE
1.	MAIN DRIVER: MAIN DRIVER FOR NOVA ROUTINES.....	119
2.	SYS_GEN: GENERATES THE SYSTEM.....	121
3.	INPUT_OUTPUT: MAIN INPUT/OUTPUT DRIVER.....	123
4.	DATA_IN: INPUT DRIVER.....	125
5.	DATA_OUT: OUTPUT DRIVER.....	127
6.	TRANS_IN: TRANSFERS DATA TO INPUT ADDRESS.....	129
7.	TRANS_OUT: TRANSFERS DATA TO OUTPUT FILE.....	131
8.	ERROR_CODE: TRANSLATES NOVA I/O ERROR CODE.....	133
9.	PG_OPTION: PAGE OPTION 1 MAIN ROUTINE.....	134
10.	INIT_PG_OPTION: PAGE OPTION 1 INITIAL ROUTINE.....	138
11.	PG_OPTION: PAGE OPTION 2 MAIN ROUTINE.....	139
12.	INIT_PG_OPTION: PAGE OPTION 2 INITIAL ROUTINE.....	142
13.	PG_OPTION: PAGE OPTION 3 MAIN ROUTINE.....	144
14.	INIT_PG_OPTION: PAGE OPTION 3 INITIAL ROUTINE.....	147
15.	PG_IN: RETRIEVES PAGE REQUESTED BY INTERDATA.....	148
16.	EOJ: END OF JCB ROUTINE.....	149
17.	L3_2_L2_PG: TRANSFERS A PAGE FROM LEVEL 3 MEMORY TO LEVEL 2 MEMORY.....	151
18.	L3_2_L1_PG: TRANSFERS A PAGE FROM LEVEL 3 MEMORY TO LEVEL 1 MEMORY.....	151
19.	L2_2_L3_PG: TRANSFERS A PAGE FROM LEVEL 2 MEMORY TO LEVEL 3 MEMORY.....	152

ALG. #	PAGE
20. L1_2_L2_PG: TRANSFERS A PAGE FROM LEVEL 1 MEMORY TO LEVEL 2 MEMORY UNDER PAGE OPTIONS 1 AND 2.....	152
21. L1_2_L2_PG: TRANSFERS A PAGE FROM LEVEL 1 MEMORY TO LEVEL 2 MEMORY UNDER PAGE OPTION 3.....	153
22. L1_2_L3_PG: TRANSFERS A PAGE FROM LEVEL 1 MEMORY TO LEVEL 3 MEMORY.....	153
23. L2_2_L1_PG: TRANSFERS A PAGE FROM LEVEL 2 MEMORY TO LEVEL 1 MEMORY.....	154
24. SPOOL_SOURCE: SPOOLS JCB FROM READER.....	155
25. NEXT_JOB: SEARCHES JOB QUEUE FOR NEXT JOB.....	157
26. CREATE_ODF: CREATE OBJECT DECK FILE NAME.....	157
27. GET_CDF: GETS THE OBJECT DECK FILE NAME FOR THE REQUESTED JOB.....	158
28. CREATE_OUT_FILE: CREATE JOB OUTPUT FILE NAME.....	158
29. STACK_DEPTH_COUNTS: LIST STACK DEPTH COUNTS FOR PAGE OPTIONS 1 OR 3.....	159
30. STACK_DEPTH_COUNTS: LIST STACK DEPTH COUNTS FOR PAGE OPTION 2.....	159

ALGORITHM 1: MAIN DRIVER FOR NCVA ROUTINES

MAIN DRIVER;

```

DCL 1 JOB_PCB(3),
    2 PMT(128),
    3 PG_LOC,
    3 DUP_LOC,
    3 ORG_BIT,
    2 OBJ_DECK_FILE,
    2 SPOOL_TABLE_IX,
    2 USER_JOB_IDEN;
DCL 1 SPCOL_TAB(100,5);
DCL 1 PG_TRAF_TAB(3),
    2 L3_2_L2_CCUNT,
    2 L3_2_L1_COUNT,
    2 L2_2_L3_CCUNT,
    2 L2_2_L1_CCUNT,
    2 L1_2_L2_CCUNT,
    2 L1_2_L3_CCUNT,
    2 IO_L3_2_L2_COUNT,
    2 IO_L2_2_L3_CCUNT,
    2 TOT_COUNT;

DO I=1 TO 100;
  SPOOL_TAB(I,1)=0;
END;
SPCOL_TAB_INDEX=1;
20  ENABLE INTERRUPTS;

```

```
DC UNTIL MESSAGE SENT BY INTERDATA OR READER INTERRUPT;
WAIT FOR MESSAGE;
END;
DISABLE INTERRUPTS;
IF CASE (MESSAGE_NUM);
1:  CALL SYS_GEN(PG_OPTION);
2:  CALL PG_IN(SYS_JOB_ID,PG_NUM,L1_ADD);
3:  CALL FOJ(SYS_JOB_ID);
4:  CALL INPUT_OUTPUT(UPB,SGPB);
5:  CALL PG_OPTION(SYS_JOB_ID,JOB_ACC_TIME,PG_NUM,
    L1_ADD,ORG_BIT);
6:  CALL SPOCL_SOURCE;
7:  CALL NEXT_JOB(SYS_JOB_ID);
8:  CALL CREATE_ODF(SYS_JOB_ID);
9:  CALL GET_CDF(SYS_JOB_ID);
10: CALL CREATE_OUT_FILE(SYS_JOB_ID);
11: CALL STACK_DEPTH_COUNTS(SYS_JOB_ID,BEG_ADD,END_ADD);
END CASE;
    ALL OTHER SUBROUTINES THAT ARE PART OF THE NOVA SYSTEM
    WILL GO HERE AS THEY ARE ALL INTERNAL TO THIS MAIN
    ROUTINE.  SAVES PASSING DATA BASE.
GO TO 20;
END;
```

ALGORITHM 2: GENERATES THE SYSTEM

```

SUBROUTINE SYS_GEN(PG_OPTION);
L2_FREE_CORE=L2_UPPER_BOUND-L2_LOWER_BOUND+1;
IF PG_OPTION=3 THEN
  DO;
    TOT_USED=0;
    INTEGER_3RD=L2_FREE_CORE/3;
    DO I=1 TO 3;
      WSCB(I).MAX_PG_F=INTEGER_3RD;
      TOT_USED=TOT_USED+INTEGER_3RD;
    END;
    DO I=1 TO 2 WHILE TOT_USED<L2_FREE_CORE;
      WSCB(I).MAX_PG_F=WSCB(I).MAX_PG_F+1;
      TOT_USED=TOT_USED+1;
    END;
    DO I=1 TO 3;
      DO J=1 TO WSCB(I).MAX_PG_F;
        WSCB(I).AVAIL_PG_FRAME(J)=CORE ADDRESS OF NEXT PAGE FRAME;
      END;
    CALL INIT_PG_OPTION(I);
  END;
END;
ELSE IF PG_OPTION=2 THEN
  DO;
    DO I=1 TO L2_FREE_CORE;
      AVAIL_PG_FRAME(I)=CORE ADDRESS OF NEXT PAGE;
    
```

```

END;

AVAIL_PG_F_INDEX=L2_FREE_CORE;

CURRENT_PG_FRAME=AVAIL_PG_FRAME(AVAIL_PG_F_INDEX);

HEAD=0;

END;

ELSE

DO;

    DC J=1 TO L2_FREE_CORE;

    AVAIL_PG_FRAME(J)=CCRE ADDRESS OF NEXT PAGE FRAME;

    END;

    AVAIL_PG_F_INDEX=L2_FREE_CORE;

    CURRENT_PG_FRAME=AVAIL_PG_FRAME(AVAIL_PG_F_INDEX);

    AVAIL_PG_F_INDEX=AVAIL_PG_F_INDEX-1;

    DC I=1 TO 3;

        DO J=1 TO 128;

            JOB_PCB(I).PMT(J).PG_LCC=0;

            END;

        CALL INIT_PG_OPTION(I);

    END;

END;

SEND INTERDATA MESSAGE "SYSTEM READY."

RETURN;

END;

```

ALGORITHM 3: MAIN INPUT/OUTPUT DRIVER

```
SUBROUTINE INPUT_OUTPUT(LPB,SGPB);
```

```
DCL 1 UPB,
```

```
2 WORD_1,
```

```
3 FUNCT_CODE,
```

```
3 LU,
```

```
2 WORD_2,
```

```
3 STATUS,
```

```
3 DEV_ADD,
```

```
2 START_V_ADD,
```

```
2 END_V_ADD,
```

```
2 REL_ADD,
```

```
2 WORD_6,
```

```
3 W_KEY,
```

```
3 R_KEY;
```

```
DCL 1 SGPB,
```

```
2 SYS_JOB_ID,
```

```
2 USER_JOB_ID,
```

```
2 ACC_JOB_TIME,
```

```
2 CRG_BITS,
```

```
2 RETURN_ERROR_CODE,
```

```
2 FILE_TYPE;
```

```
DCL PG_BUFF(256);
```

```
IF START_V_ADD NOT EVEN
```

```
ELSE IF END_V_ADD NOT ODD THEN
```

```
DO;
```



```

STATUS=X'84';
DEV_ADD=PHYSICAL ADDRESS OF DEVICE REQUESTED;
RETURN;
END;

NUM_WRDS_2B_TRANS=((END_V_ADD-START_V_ADD)+1)/2;
START_PG_NUM=START_V_ADD(BITS 0-6);
END_PG_NUM=END_V_ADD(BITS 0-6);
IF FUNCT_CODE=20,24,30,40,44,50 THEN
    SEND INTERDATA MESSAGE I/C STARTED;
IF FUNCT_CODE=20,24,28,2C,30 OR 38 THEN
    DO;
    CALL DATA_OUT;
    RETURN;
    END;
ELSE IF FUNCT_CODE=40,44,48,4C,50 OR 58 THEN
    DO;
    CALL DATA_IN;
    RETURN;
    END;
ELSE DO;
    STATUS=X'C0';
    DEV_ADD=PHYSICAL ADDRSS OF DEVICE REQUESTED;
    RETURN;
    END;

```

ALGORITHM 4: INPUT DRIVER

```
SUBROUTINE DATA_IN;
```

```
  OFF_SET=START_V_ADD(BITS 7-15);
```

```
  CONVERT INTERDATA FUNCTION CODE TO EQUIVALENT NOVA  
    INPUT COMMAND.
```

```
  IF START_PG_NUM=END_PG_NUM THEN
```

```
    CALL TRANS_IN(OFF_SET,NUM_WRDS_2B_TRANS);
```

```
  ELSE
```

```
    DO;
```

```
      NUM_WRDS=(512-START_V_ADD, BITS 7-15)/2;
```

```
      CALL TRANS_IN(OFF_SET,NUM_WRDS);
```

```
      NUM_WRDS_2B_TRANS=NUM_WRDS_2B_TRANS-NUM_WRDS;
```

```
      DO WHILE NUM_WRDS_2B_TRANS>=256;
```

```
        START_PG_NUM=START_PG_NUM+1;
```

```
        CALL TRANS_IN(0,256);
```

```
        NUM_WRDS_2B_TRANS=NUM_WRDS_2B_TRANS-256;
```

```
      END;
```

```
      START_PG_NUM=START_PG_NUM+1;
```

```
      IF NUM_WRDS_2B_TRANS>0 THEN
```

```
        CALL TRANS_IN(0,NUM_WRDS_2B_TRANS);
```

```
      END;
```

```
  IF ERROR DURING I/O THEN
```

```
    DO;
```

```
      RETURN_ERROR_CODE=AC2;
```

```
      CALL ERROR_CODE(AC2,STATUS,DEV_ADD);
```

```
    END;
```

```
ELSE WORD_2=0;  
RETURN;  
END;
```

ALGORITHM 5: OUTPUT DRIVER

```

SUBROUTINE DATA_OUT;
OFF_SET=START_V_ADD(BITS 7-15);
CONVERT INTERDATA OUTPUT FUNCTION CODE TO EQUIVALENT NOVA
  OUTPUT CCOMMAND.
IF START_PG_NUM=END_PG_NUM THEN
  CALL TRANS_OUT(OFF_SET,NUM_WRDS_2B_TRANS);
ELSE
  DO;
    NUM_WRDS=(512-START_V_ADD)/2;
    CALL TRANS_OUT(OFF_SET,NUM_WRDS);
    NUM_WRDS_2B_TRANS=NUM_WRDS_2B_TRANS-NUM_WRDS;
    DO WHILE NUM_WRDS_2B_TRANS>=256;
      START_PG_NUM=START_PG_NUM+1;
      CALL TRANS_OUT(0,256);
      NUM_WRDS_2B_TRANS=NUM_WRDS_2B_TRANS-256;
    END;
    START_PG_NUM=START_PG_NUM+1;
    IF NUM_WRDS_2B_TRANS>0 THEN
      CALL TRANS_OUT(0,NUM_WRDS_2B_TRANS);
    END;
  IF ERROR DURING I/O THEN
    DO;
      RETURN_ERROR_CODE=AC2;
      CALL ERROR_CODE(AC2,STATUS,DEV_ADD);
    END;

```

```
ELSE WORD_2=0;  
RETURN;  
END;
```

ALGORITHM 6: TRANSFERS DATA TO INPUT ADDRESS

```

SUBROUTINE TRANS_IN(OFF_SET,NUM_WRDS);
FILE_IDEN=SPOOL_TAB(JOB_PCB(SYS_JOB_ID).SPOOL_TABLE_IX,FILE_TYPE);
FILE_IDEN=SPOOL_TAB(SPOOL_TABLE_IX(SYS_JOB_ID),FILE_TYPE);
IF JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1)>0 THEN
  DO;
    BEG_ADD=JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1)+OFF_SET;
    READ IN SEQUENTIALLY THE NUMBER OF WORDS (NUM_WRDS) FROM
      THE GIVEN FILE (FILE_IDEN), INTO LEVEL 1 MEMORY
      BEGINNING AT ADDRESS (BEG_ADD).
    JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1).ORG_BIT=1;
    RETURN;
  END;
ELSE IF JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1)<0 THEN
  DO;
    BEG_ADD=
      (JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1)*-1)+(OFF_SET/2);
    READ IN SEQUENTIALLY THE NUMBER OF WORDS (NUM_WRDS) FROM
      THE GIVEN FILE (FILE_IDEN), INTO LEVEL 2 MEMORY
      BEGINNING AT ADDRESS (BEG_ADD).
    JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1).ORG_BIT=1;
    RETURN;
  END;
ELSE
  DO;
    FILE_ID=JOB_PCB(SYS_JOB_ID).CBJ_DECK_FILE;

```

```
CALL L3_2_L2_PG(SYS_JOB_ID, START_PG_NUM, FILE_ID, PG_BUFF);  
BEG_ADD=OFF_SET/2+1;  
READ IN SEQUENTIALLY THE NUMBER OF WORDS (NUM_WRDS),  
    FROM THE GIVEN FILE (FILE_IDEN), INTO (PG_BUFF)  
    BEGINNING AT ADDRESS PG_BUFF(BEG_ADD).  
CALL RDOS TO TRANSFER 256 CONSECUTIVE WORDS BEGINNING IN  
    LEVEL 2 MEMORY AT ADDRESS (PG_BUFF), TO FILE (FILE_ID),  
    AT RELATIVE BLOCK NUMBER (PG_NUM).  
RETURN;  
END;
```

ALGORITHM 7: TRANSFERS DATA TO OUTPUT FILE

```
SUBROUTINE TRANS_OUT(OFF_SET,NUM_WRDS);
```

```
OUT_FILE=SPOOL_TAB(SPOOL_TABLE_IX(SYS_JCB_ID),FILE_TYPE);
```

```
IF JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1)>0 THEN
```

```
DO;
```

```
BEG_ADD=JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1)+OFF_SET;
```

```
WRITE TO THE JOB'S OUTPUT FILE (OUT_FILE), SEQUENTIALLY  
THE NUMBER OF WORDS REQUESTED (NUM_WRDS), FROM  
LEVEL 1 MEMORY BEGINNING AT ADDRESS (BEG_ADD).
```

```
RETURN;
```

```
END;
```

```
ELSE IF JOB_PCB(SYS_JOB_ID).PMT(START_PG_NUM+1)<0 THEN
```

```
DO;
```

```
BEG_ADD=
```

```
(JOB_PCB(SYS_JCB_ID).PMT(START_PG_NUM+1)*-1)+(OFF_SET/2);
```

```
WRITE TO THE JOB'S OUTPUT FILE (OUT_FILE), SEQUENTIALLY  
THE NUMBER OF WORDS REQUESTED (NUM_WRDS), FROM  
LEVEL 2 MEMORY BEGINNING AT ADDRESS (BEG_ADD).
```

```
RETURN;
```

```
END;
```

```
ELSE
```

```
DO;
```

```
FILE_ID=JOB_PCB(SYS_JCB_ID).OBJ_DECK_FILE;
```

```
CALL L3_2_L2_PG(SYS_JOB_ID,START_PG_NUM,FILE_ID,PG_BUFF);
```

```
BEG_ADD=OFF_SET/2+1;
```

```
WRITE TO THE JOB'S OUTPUT FILE (OUT_FILE), SEQUENTIALLY THE
```


NUMBER OF WORDS REQUESTED (NUM_WRDS), FROM
LEVEL 2 MEMORY BEGINNING AT ADDRESS (PG_BUFF(BEG_ADD)).
RETURN;
END;

ALGORITHM 8: TRANSLATES NCVA I/O ERROR CODE
TO INTERDATA I/O ERROR CODE

```
SUBROUTINE ERROR_CODE(AC2,STATUS,DEV_ADD);  
IF DEVICE UNAVAILABLE THEN STATUS=X'A0';  
  ELSE IF END OF MEDIUM THEN STATUS=X'90';  
    ELSE IF END OF FILE THEN STATUS=X'88';  
      ELSE IF UNRECOVERABLE ERROR THEN STATUS=X'84';  
DEV_ADD=PHYSICAL ADDRESS OF DEVICE REQUESTED (8 BITS);  
RETURN;  
END;
```

ALGORITHM 9: PAGE OPTION 1 MAIN ROUTINE

```

SUBROUTINE PG_OPTION(SYS_JOB_ID,JOB_ACC_TIME,PG_NUM,
  L1_ADD,IORG_BIT);
DCL  1  WSCB(3),
      2  PREV_ACC_TIME,
      2  PREV_TIME_SPAN,
      2  HEAD,
      2  EPFT(128),
      3  PG_NUM,
      3  DEPTH_COUNT;
DCL AVAIL_PG_F_INDEX;
DCL AVAIL_PG_FRAME(128);
TEMP_PG_NUM=PG_NUM;
CURRENT_TIME_SPAN=JOB_ACC_TIME-WSCB(SYS_JOB_ID).PREV_ACC_TIME;
IF JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LCC<0 THEN
  DO;
  IF IORG_BIT=1 THEN
    DO;
    CALL L1_2_L2_PG(SYS_JOB_ID,PG_NUM,L1_ADD,IORG_BIT);
    CURRENT_PG_FRAME=JOB_PCB(SYS_JOB_ID).PMT(PG_NUM).DUP_LCC*-1;
    JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LCC=0;
    END;
  SEND INTERDATA MESSAGE, "PAGE REMOVED FROM LEVEL 1 MEMORY."
  I=1;
  DO WHILE WSCB(SYS_JOB_ID).EPFT(I).PG_NUM/=PG_NUM;
    I=I+1;

```

```

END;
WSCB(SYS_JOB_ID).EPFT(I).DEPTH_COUNT=
  WSCB(SYS_JOB_ID).EPFT(I).DEPTH_COUNT+1;
DO J=I TO WSCB(SYS_JOB_ID).HEAD-1;
  WSCB(SYS_JOB_ID).EPFT(J).PG_NUM=
    WSCB(SYS_JOB_ID).EPFT(J+1).PG_NUM;
END;
IF CURRENT_TIME_SPAN>WSCB(SYS_JOB_ID).PREV_TIME_SPAN*1.10 &
  WSCB(SYS_JOB_ID).HEAD>1 THEN
  CALL D_CREASE_WSS(1);
END;
ELSE
  DO;
    DO;
      CALL L1_2_L2_PG(SYS_JOB_ID,PG_NUM,L1_AD,ICRG_BIT);
      SEND INTERDATA MESSAGE, "PAGE REMOVED FROM LEVEL 1 MEMORY."
    END;
    IF CURRENT_TIME_SPAN<WSCB(SYS_JOB_ID).PREV_TIME_SPAN*.90 THEN
      IF AVAIL_PG_F_INDEX>0 THEN
        DO;
          AVAIL_PG_F_INDEX=AVAIL_PG_F_INDEX-1;
          CURRENT_PG_FRAME=AVAIL_PG_FRAME(AVAIL_PG_F_INDEX);
          HEAD=HEAD+1;
        END;
      ELSE
        DO;
          IF CURRENT_TIME_SPAN>

```

```

        WSCB(SYS_JCB_ID).PREV_TIME_SPAN*1.10 E
        WSCB(SYS_JOB_ID).HEAD>1 THEN
        N=2;
        ELSE N=1;
        CALL D_CREASE_WSS(N);
        END;

    END;

    WSCB(SYS_JCB_ID).PREV_TIME_SPAN=CURRENT_TIME_SPAN;
    WSCB(SYS_JOB_ID).PREV_ACC_TIME=JOB_ACC_TIME;
    WSCB(SYS_JOB_ID).EPFT(HEAD)=TEMP_PG_NUM;
    RETURN;

    *****

    SUBROUTINE D_CREASE_WSS(N);
    DO I=1 TO N;
    PG_NUM=WSCB(SYS_JOB_ID).EPFT(I).PG_NUM;
    IF JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).CRG_BIT=1 THEN
        DO;
        FILE_ID=JOB_PCB(SYS_JOB_ID).OBJ_DECK_FILE;
        CALL L2_2_L3_PG(SYS_JOB_ID,PG_NUM,FILE_ID);
        END;
    AVAIL_PG_F_INDEX=AVAIL_PG_F_INDEX+1;
    AVAIL_PG_FRAME(AVAIL_PG_F_INDEX)=
    JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LCC*-1;
    JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC=0;
    END;
    CURRENT_PG_FRAME=AVAIL_PG_FRAME(AVAIL_PG_F_INDEX);
    DO I=1 TO WSCB(SYS_JOB_ID).HEAD-N;

```

```
WSCB(SYS_JOB_ID).EPFT(I).PG_NUM=WSCB(SYS_JOB_ID).EPFT(I+N).PG_NUM  
END;  
HEAD=HEAD-N+1;  
RETURN;  
*****  
END;
```

ALGORITHM 10: PAGE OPTION 1 INITIAL ROUTINE

```

SUBROUTINE INIT_PG_OPTION(I);
  WSCB(I).HEAD=0;
  WSCB(I).PREV_ACC_TIME=0;
  WSCB(I).PREV_TIME_SPAN=0;
  L3_2_L2_COUNT(I)=0
  L3_2_L1_COUNT(I)=0
  L2_2_L3_COUNT(I)=0
  L2_2_L1_COUNT(I)=0
  L1_2_L3_COUNT(I)=0
  L1_2_L2_COUNT(I)=0
  I/O_L3_2_L2_COUNT(I)=0;
  I/O_L2_2_L3_COUNT(I)=0;
  TCT_COUNT(I)=0;
  DO J=1 TO 128;
    IF JOB_PCB(I).PMT(J).PG_LCC<0 THEN
      DO;
        AVAIL_PG_F_INDEX=AVAIL_PG_F_INDEX+1;
        AVAIL_PG_FRAME(AVAIL_PG_F_INDEX)=
          JOB_PCB(I).PMT(J).PG_LCC*-1;
        JOB_PCB(I).PMT(J).DUP_LCC=0;
      END;
      JOB_PCB(I).PMT(J).PG_LCC=0;
      WSCB(SYS_JOB_ID).EPFT(I).DEPTH_COUNT=C;
    END;
  RETURN;
END;

```

ALGORITHM 11: PAGE OPTION 2 MAIN ROUTINE

```

SUBROUTINE PG_OPTION(SYS_JOB_ID,JOB_ACC_TIME,PG_NUM,
    L1_ADD,IORG_BIT);
DCL 1 WSCB,
    2 EPFT(128),
    3 PG_NUM,
    3 DEPTH_COUNT
    2 AVAIL_PG_FRAME(128),
    2 AVAIL_PG_F_INDEX,
    2 HEAD,
TEMP_PG_NUM=PG_NUM;
IF JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LOC<0 THEN
DO;
    IF IORG_BIT=1 THEN
        DO;
            CALL L1_2_L2_PG(SYS_JOB_ID,PG_NUM,L1_ADD,ICRG_BIT);
            CURRENT_PG_FRAME=JOB_PCB(SYS_JOB_ID).PMT(PG_NUM).DUP_LOC*-1;
            JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LOC=0;
        END;
    SEND INTERDATA MESSAGE, "PAGE REMOVED FROM LEVEL 1 MEMORY."
    I=1;
    DO WHILE EPFT(I).PG_NUM/=PG_NUM&EPFT(I).JOB_IDEN/=SYS_JOB_ID
        I=I+1;
    END;
    WSCB.EPFT(I).DEPTH_COUNT=WSCB.EPFT(I).DEPTH_COUNT+1;
    DO J=1 TO HEAD-1;

```



```

        EPFT(I).PG_NUM=EPFT(I+1).PG_NUM;
        EPFT(I).JOB_IDEN=EPFT(I+1).JOB_IDEN;
        END;
    END;
ELSE
    DO;
        CALL L1_2_L2_PG(SYS_JOB_ID,PG_NUM,L1_ADD,ICRG_BIT);
        SEND INTERDATA MESSAGE, "PAGE REMOVED FROM LEVEL 1 MEMORY."
        IF AV_PG_F_INDEX>1 THEN
            DO;
                AV_PG_F_INDEX=AV_PG_F_INDEX-1;
                CURRENT_PG_FRAME=AVAIL_PG_FRAME(AV_PG_F_INDEX);
                HEAD=HEAD+1;
            END;
        ELSE
            DO;
                PG_NUM=EPFT(1).PG_NUM;
                PMT_SYS_JOB_ID=EPFT(1).JOB_IDEN;
                IF JOB_PCB(PMT_SYS_JOB_ID).PMT(PG_NUM+1).ORG_BIT=1 THEN
                    DO;
                        FILE_ID=JOB_PCB(SYS_JOB_ID).OBJ_DECK_FILE;
                        CALL L2_2_L3_PG(PMT_SYS_JOB_ID,PG_NUM,FILE_ID);
                    END;
                JOB_PCB(PMT_SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC=0;
                CURRENT_PG_FRAME=
                    JOB_PCB(PMT_SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC*-1;
                DO I=1 TO HEAD-1;

```

```
        EPFT(I).PG_NUM=EPFT(I+1).PG_NUM;  
        EPFT(I).JOB_IDEN=EPFT(I+1).JOB_IDEN;  
    END;  
END;  
END;  
EPFT(HEAD).PG_NUM=TEMP_PG_NUM;  
EPFT(HEAD).JOB_IDEN=SYS_JOB_ID;  
RETURN;  
END;
```

ALGORITHM 12: PAGE OPTION 2 INITIAL ROUTINE

```

SUBROUTINE INIT_PG_OPTION(I);
DCL  1  TEMP_EPFT(128),
      2  PG_NUM,
      2  JOB_IDEN;

L3_2_L2_CCUNT(I)=0
L3_2_L1_CCUNT(I)=0
L2_2_L3_CCUNT(I)=0
L2_2_L1_CCUNT(I)=0
L1_2_L3_CCUNT(I)=0
L1_2_L2_CCUNT(I)=0
I/O_L3_2_L2_CCUNT(I)=0;
I/O_L2_2_L3_CCUNT(I)=0;
TOT_COUNT(I)=0;

L=1;
DO WHILE EPFT(L).JOB_IDEN/=I;
  L=L+1;
END;

DO K=L TO HEAD;
  IF EPFT(K).JOB_IDEN/=I THEN
    DO;
      EPFT(L).PG_NUM=EPFT(K).PG_NUM;
      EPFT(L).JOB_IDEN=EPFT(K).JOB_IDEN;
      L=L+1;
    END;
  ELSE

```

```
DC;  
AVAIL_PG_F_INDEX=AVAIL_PG_F_INDEX+1;  
PG_NUM=EPFT(K).PG_NUM;  
AVAIL_PG_FRAME(AVAIL_PG_F_INDEX)=  
    JOB_PCB(I).PMT(PG_NUM+1).PG_LCC*-1;  
END;  
  
END;  
  
HEAD=L-1;  
CURRENT_PG_FRAME=AVAIL_PG_FRAME(AV_PG_F_INDEX);  
RETURN;  
END;
```

ALGORITHM 13: PAGE OPTION 3 MAIN ROUTINE

```

SUBROUTINE PG_OPTION(SYS_JOB_ID,JOB_ACC_TIME,PG_NUM,
    LI_ADD,IORG_BIT);
DCL  1  WSCB(3),
      2  EPFT(128),
      2  AVAIL_PG_FRAME(128),
      2  AVAIL_PG_F_INDEX,
      2  HEAD,
      2  CURRENT_PG_FRAME,
      2  MAX_PG_F,
      2  DEPTH_COUNT;
TEMP_PG_NUM=PG_NUM;
IF JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LOC<0 THEN
    DO;
    IF IORG_BIT=1 THEN
        DO;
        CALL LI_2_L2_PG(SYS_JOB_ID,PG_NUM,LI_ADD,IORG_BIT);
        WSCB(SYS_JOB_ID).CURRENT_PG_FRAME=
            JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LOC*-1;
        JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LOC=0;
        END;
    SEND INTERDATA MESSAGE, "PAGE REMOVED FROM LEVEL 1 MEMORY."
    I=1;
    DO WHILE WSCB(SYS_JOB_ID).EPFT(I)~=PG_NUM;
    I=I+1;
    END;

```

```

WSCB(SYS_JOB_ID).DEPTH_COUNT=WSCB(SYS_JOB_ID).DEPTH_COUNT+1;
DO J=1 TO HEAD-1;
WSCB(SYS_JOB_ID).EPFT(J)=
    WSCB(SYS_JOB_ID).EPFT(J+1);
END;
END;
ELSE
DO;
CALL L1_2_L2_PG(SYS_JOB_ID,PG_NUM,L1_ADD,ICRG_BIT);
SEND INTERDATA MESSAGE, "PAGE REMOVED FROM LEVEL 1 MEMORY."
IF WSCB(SYS_JOB_ID).AVAIL_PG_F_INDEX>1 THEN
DO;
WSCB(SYS_JOB_ID).AVAIL_PG_F_INDEX=
    WSCB(SYS_JOB_ID).AVAIL_PG_F_INDEX-1;
WSCB(SYS_JOB_ID).CURRENT_PG_FRAME=
    WSCB(SYS_JOB_ID).AVAIL_PG_FRAME(WSCB(SYS_JOB_ID).
    AVAIL_PG_F_INDEX);
WSCB(SYS_JOB_ID).HEAD=WSCB(SYS_JOB_ID).HEAD+1;
END;
ELSE;
DO;
PG_NUM=WSCB(SYS_JOB_ID).EPFT(1);
IF JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).ORG_BIT=1 THEN
DO;
FILE_ID=JOB_PCB(SYS_JOB_ID).OBJ_DECK_FILE;
CALL L2_2_L3_PG(SYS_JOB_ID,PG_NUM,FILE_ID);
END;

```

```
WSCB(SYS_JOB_ID).CURRENT_PG_FRAME=  
    JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC*-1;  
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC=0;  
DO I=1 TO HEAD-1;  
    WSCB(SYS_JOB_ID).EPFT(I)=WSCB(SYS_JOB_ID).EPFT(I+1);  
END;  
END;  
END;  
WSCB(SYS_JOB_ID).EPFT(HEAD)=TEMP_PG_NUM;  
RETURN;  
END;
```

ALGORITHM 14: PAGE OPTION 3 INITIAL ROUTINE

```
SUBROUTINE INIT_PG_OPTION(I);  
WSCB(I).AVAIL_PG_F_INDEX=MAX_PG_F(I);  
WSCB(I).HFAD=0;  
WSCB(I).CURRENT_PG_FRAME=WSCB(I).AVAIL_PG_FRAME(MAX_PG_F(I));  
L3_2_L2_CCUNT(I)=0  
L3_2_L1_CCUNT(I)=0  
L2_2_L3_CCUNT(I)=0  
L2_2_L1_CCUNT(I)=0  
L1_2_L3_CCUNT(I)=0  
L1_2_L2_CCUNT(I)=0  
I/O_L3_2_L2_CCUNT(I)=0;  
I/O_L2_2_L3_CCUNT(I)=0;  
TOT_COUNT(I)=0;  
  DO J=1 TO 128;  
    JOB_PCB(I).PG_LOC(J)=0;  
    JOB_PCB(I).DUP_LOC(J)=0;  
    WSCB(I).DEPTH_COUNT=0;  
  END;  
RETURN;  
END;
```


ALGORITHM 15: RETRIEVES PAGE REQUESTED BY INTERDATA

```
SUBROUTINE PG_IN(SYS_JOB_ID,PG_NUM,L1_ADD);  
IF JOB_PCE(SYS_JOB_ID).PG_LOC(PG_NUM+1)<0 THEN  
  DO;  
    CALL L2_2_L1(SYS_JOB_ID,PG_NUM,L1_ADD);  
    JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).DUP_LOC=  
      JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC;  
    JOB_PCR(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC=L1_ADD;  
  END;  
  ELSE  
    DO;  
      FILE_ID=OBJ_DECK_FILE(SYS_JOB_ID);  
      CALL L3_2_L1(SYS_JOB_ID,PG_NUM,FILE_ID,L1_ADD);  
    END;  
  SEND INTERDATA MESSAGE "PAGE TRANSFERRED."  
  RETURN;  
END;
```

ALGORITHM 16: END OF JOB ROUTINE

```

SUBROUTINE EOJ(SYS_JOB_ID);
TOT_COUNT(SYS_JOB_ID)=L3_2_L2_COUNT(SYS_JOB_ID)+
  L3_2_L1_COUNT(SYS_JOB_ID)+L2_2_L3_COUNT(SYS_JOB_ID)+
  L1_2_L2_COUNT(SYS_JOB_ID)+L1_2_L3_COUNT(SYS_JOB_ID)+
  L2_2_L1_COUNT(SYS_JOB_ID);
L3_2_L2_COUNT(SYS_JOB_ID)=
  L3_2_L2_COUNT(SYS_JOB_ID)-IO_L3_2_L2_COUNT(SYS_JOB_ID);
L2_2_L3_COUNT(SYS_JOB_ID)=
  L2_2_L3_COUNT(SYS_JOB_ID)-IO_L2_2_L3_COUNT(SYS_JOB_ID);
CALL ON RDCS TO ADD TO THE JOB'S DATA OUT FILE THE PAGE MIGRATION
COUNTS, WITH HEADINGS AND GIVEN VALUES,
(I.E. LEVEL 3 TO LEVEL 2 COUNT=865).
THESE ARE THE PAGE COUNTS OF PAGE TRANSFERS BETWEEN MEMORY
LEVELS. IO ARE COUNTS CAUSED BY I/C REQUEST.
IF SOURCE LISTING REQUESTED THEN
  DO;
    SPOOL_OUT=SPOOL_TAB(SPOOL_TAB_IX(SYS_JOB_ID),2);
    ISSUE RDCS THE COMMAND TO SPOOL TO OUTPUT THE JOB'S SOURCE
      FILE (SPOOL_OUT).
  END;
  SPOOL_OUT=SPOOL_TAB(SPOOL_TABLE_IX(SYS_JOB_ID),5);
  ISSUE RDCS THE COMMAND TO SPOOL TO OUTPUT THE JOB'S OUTPUT
  FILE (SPOOL_OUT).
  DO L=1 TO 5;
    DELETE_FILE=SPOOL_TAB(SPOOL_TABLE_IX(SYS_JOB_ID),L);

```

```
CALL RDOOS TO DELETE FILE (DELETE_FILE);  
END;  
SPOOL_TAB(SPOOL_TABLE_IX(SYS_JOB_ID),1)=0;  
CALL INIT_PG_OPTICN(SYS_JOB_ID);  
SEND INTERDATA MESSAGE "JOB COMPLETED."  
RETURN;  
END;
```

ALGORITHM 17: TRANSFERS A PAGE FROM LEVEL 3 MEMORY
TO LEVEL 2 MEMORY

```
SUBROUTINE L3_2_L2_PG(SYS_JOB_ID,PG_NUM,FILE_ID,PG_BUFF);  
CALL RDOS TO READ 256 CONSECUTIVE WORDS FROM GIVEN  
FILE (FILE_ID), AT RELATIVE BLOCK NUMBER (PG_NUM),  
INTO (PG_BUFF).  
RETURN;  
END;
```

ALGORITHM 18: TRANSFERS A PAGE FROM LEVEL 3 MEMORY
TO LEVEL 1 MEMORY

```
SUBROUTINE L3_2_L1_PG(SYS_JCB_ID,PG_NUM,FILE_ID,L1_ADD);  
CALL RDOS TO READ 256 CONSECUTIVE WORDS FROM GIVEN FILE  
(FILE_ID), AT RELATIVE BLOCK NUMBER (PG_NUM), INTO  
LEVEL 1 MEMORY BEGINNING AT ADDRESS (L1_ADD).  
JOB_PCB(SYS_JCB_ID).PMT(PG_NUM+1).PG_LOC=L1_ADD;  
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).ORG_BIT=0;  
RETURN;  
END;
```

ALGORITHM 19: TRANSFERS A PAGE FROM LEVEL 2 MEMORY
TO LEVEL 3 MEMORY

```
SUBROUTINE L2_2_L3_PG(SYS_JOB_ID,PG_NUM,FILE_ID);
L2_ADD=JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC*-1;
CALL RDOS TO TRANSFER 256 CONSECUTIVE WORDS BEGINNING IN
    LEVEL 2 MEMORY AT ADDRESS (L2_ADD), TO FILE (FILE_ID),
    AT RELATIVE BLOCK NUMBER (PG_NUM).
RETURN;
END;
```

ALGORITHM 20: TRANSFERS A PAGE FROM LEVEL 1 MEMORY
TO LEVEL 2 MEMORY UNDER PAGE OPTIONS 1 AND 2

```
SUBROUTINE L1_2_L2_PG(SYS_JOB_ID,PG_NUM,L1_ADD,ICRG_BIT);
L2_ADD=CURRENT_PG_FRAME;
CALL RDOS TO TRANSFER 256 CONSECUTIVE WORDS BEGINNING IN
    LEVEL 1 MEMORY AT ADDRESS (L1_ADD), TO LEVEL 2 MEMORY
    BEGINNING AT ADDRESS (L2_ADD).
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).CRG_BIT=
    JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).CRG_BIT|ICRG_BIT;
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC=L2_ADD*-1;
RETURN;
END;
```

ALGORITHM 21: TRANSFERS A PAGE FROM LEVEL 1 MEMORY
TO LEVEL 2 MEMORY UNDER PAGE OPTION 3

```
SUBROUTINE L1_2_L2_PG(SYS_JOB_ID,PG_NUM,L1_ADD,IORG_BIT);
L2_ADD=WSCH(SYS_JOB_ID).CURRENT_PG_FRAME;
CALL RDOS TO TRANSFER 256 CONSECUTIVE WORDS BEGINNING IN
LEVEL 1 MEMORY AT ADDRESS (L1_ADD), TO LEVEL 2 MEMORY
BEGINNING AT ADDRESS (L2_ADD).
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC=L2_ADD*-1;
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).ORG_BIT=
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).CRG_BIT||IORG_BIT;
RETURN;
END;
```

ALGORITHM 22: TRANSFERS A PAGE FROM LEVEL 1 MEMORY
TO LEVEL 3 MEMORY

```
SUBROUTINE L1_2_L3_PG(SYS_JOB_ID,PG_NUM,FILE_ID,L1_ADD);
CALL RDOS TO TRANSFER 256 CONSECUTIVE WORDS BEGINNING IN
LEVEL 1 MEMORY AT ADDRESS (L1_ADD), TO FILE (FILE_ID),
AT RELATIVE BLOCK NUMBER (PG_NUM).
JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LOC=0;
RETURN;
END;
```

ALGORITHM 23: TRANSFERS A PAGE FROM LEVEL 2 MEMORY
TO LEVEL 1 MEMORY

```
SUBROUTINE L2_2_L1_PG(SYS_JOB_ID,PG_NUM,L1_ADD);  
L2_ADD=JOB_PCB(SYS_JOB_ID).PMT(PG_NUM+1).PG_LCC*-1;  
CALL RDOS TO TRANSFER 256 CONSECUTIVE WORDS FROM LEVEL 2  
MEMORY BEGINNING AT ADDRESS (L2_ADD) TO LEVEL 1 MEMORY  
BEGINNING AT ADDRESS (L1_ADD).  
RETURN;  
END;
```

ALGORITHM 24: SPOOLS JOB FROM READER

```

SUBROUTINE SPCCL_SOURCE;

LIMIT=1;

DO UNTIL (SPCOL_TAB(SPCCL_TAB_INDEX,1)=C WHILE LIMIT<100;
SPCOL_TAB_INDEX=SPCCL_TAB_INDEX+1;
IF SPCOL_TAB_INDEX>100 THEN SPCOL_TAB_INDEX=1;
LIMIT=LIMIT+1;
END;

IF LIMIT=100 THEN
DO;
SEND INTERDATA MESSAGE, "SPOOL TABLE FULL, JOB NOT SPCOLED."
RETURN;
END;

CREATE UNIQUE JOB ID (JOB_NUM);
SPCOL_TAB(SPCCL_TAB_INDEX,1)=JOB_NUM;
IF SOURCE DECK THEN
DO;
CREATE UNIQUE SOURCE DECK FILE ID (SDF_ID);
SPCOL_TAB(SPCOL_TAB_INDEX,2)=SDF_ID;
READ IN SOURCE DECK INTO A "SEQUENTIALLY ORGANIZED FILE."
END;
ELSE IF OBJECT DECK THEN
DO;
CREATE UNIQUE OBJECT DECK FILE ID (OBJDF_ID);
SPCOL_TAB(SPCOL_TAB_INDEX,4)=OBJDF_ID;
READ IN OBJECT DECK INTO A "CONTIGUOUSLY ORGANIZED FILE."

```



```
END;  
IF DATA THEN;  
  DO;  
    CREATE UNIQUE DATA FILE ID (DF_ID);  
    SPOOL_TAB(SPOOL_TAB_INDEX,3)=DF_ID;  
    READ IN DATA INTO A "SEQUENTIALLY ORGANIZED FILE."  
  END;  
  ENTER JOB'S JOB NUMBER (JOB_NUM) IN JOB QUEUE;  
  ENTER JOB'S SPOOL TABLE INDEX (SPOOL_TAB_INDEX) IN JOB QUEUE;  
  ENTER JOB'S JCL IN JOB QUEUE;  
  RETURN;  
END;
```

ALGORITHM 25: SEARCHES JOB QUEUE FOR NEXT JOB

```

SUBROUTINE NEXT_JOB(SYS_JCB_ID);
SEARCH JOB QUEUE FOR NEXT JOB AND SAVE ITS
    SPOOL TABLE INDEX (SP_TB_INDEX).
IF JOB IN QUEUE THEN
    DO;
    USER_JOB_IDEN(SYS_JCB_ID)=SPOOL_TAB(SP_TB_INDEX,1);
    JOB_PCB(SYS_JCB_ID).SPOOL_TABLE_IX=SP_TB_INDEX;
    SEND INTERDATA (USER_JOB_IDEN) AND THE JOB'S JCL;
    ELSE RETURN INTERDATA MESSAGE, "JOB QUEUE EMPTY."
RETURN;
END;

```

ALGORITHM 26: CREATE OBJECT DECK FILE NAME

```

SUBROUTINE CREATE_ODF(SYS_JCB_ID);
CREATE UNIQUE OBJECT DECK FILE ID (OBJDF_ID);
SPOOL_TAB(JOB_PCB(SYS_JCB_ID).SPOOL_TABLE_IX,4)=OBJDF_ID;
OBJ_DECK_FILE(SYS_JCB_ID)=OBJDF_ID;
RETURN (OBJDF_ID);
END;

```

ALGORITHM 27: GETS THE OBJECT DECK FILE NAME FOR THE
REQUESTED JOB

```
SUBROUTINE GET_ODF(SYS_JOB_ID);  
JOB_PCB(SYS_JOB_ID).OBJ_DECK_FILE=  
    SPOOL_TAB(JOB_PCB(SYS_JOB_ID).SPCCL_TABLE_IX,4);  
RETURN (OBJ_DECK_FILE);  
END;
```

ALGORITHM 28: CREATE JOB OUTPUT FILE NAME

```
SUBROUTINE CREATE_OUT_FILE(SYS_JOB_ID);  
CREATE UNIQUE OUTPUT FILE ID (OUT_FILE);  
SPOOL_TAB(SPCCL_TABLE_IX(SYS_JOB_ID),5)=OUT_FILE;  
RETURN (OUT_FILE);  
END;
```

ALGORITHM 29: LIST STACK DEPTH CCUNTS FOR PAGE OPTIONS 1 AND 3

```

SUBROUTINE STACK_DEPTH_CCUNTS(SYS_JOB_ID,BEG_ADD,END_ADD);
DO I=1 TO 128;
PG_BUFF(I)=WSCB(SYS_JOB_ID).EPFT(I).DEPTH_COUNT;
WSCB(SYS_JOB_ID).EPFT(I).DEPTH_COUNT=0;
END;
BEG_ADD=ADDRESS OF PG_BUFF(1);
END_ADD=ADDRESS OF PG_BUFF(128);
RETURN;
END;

```

ALGORITHM 30: LIST STACK DEPTH CCUNTS FOR PAGE OPTION 2

```

SUBROUTINE STACK_DEPTH_CCUNTS(SYS_JOB_ID,BEG_ADD,END_ADD);
DO I=1 TO 128;
PG_BUFF(I)=WSCB.EPFT(I).DEPTH_COUNT;
WSCB.EPFT(I).DEPTH_COUNT=0;
END;
BEG_ADD=ADDRESS OF PG_BUFF(1);
END_ADD=ADDRESS OF PG_BUFF(123);
RETURN;
END;

```

APPENDIX B

(1) MAIN DRIVER

(9) PG OPTION

(2) SYS GEN

(10) INIT PG OPTION

(16) EQJ

(29) STACK DEPTH COUNTS

(3) INPUT OUTPUT

(4) DATA IN

(5) DATA OUT

(6) TRANS IN

(7) TRANS OUT

(8) ERROR CODE

(15) PG IN

(17) L3 2 L2 PG

(18) L3 2 L1 PG

(19) L2 2 L3 PG

(20) L1 2 L2 PG

(22) L1 2 L3 PG

(23) L2 2 L1 PG

(24) SPOOL SOURCE

(25) NEXT JOB

(26) CREATE ODF

(27) GET ODF

(28) CREATE OUT FILE

System Configuration For Page Option 1

APPENDIX C

(1) MAIN DRIVER

(11) PG OPTION

(2) SYS GEN

(12) INIT PG OPTION

(16) EQJ

(30) STACK DEPTH COUNTS

(3) INPUT OUTPUT

(4) DATA IN

(5) DATA OUT

(6) TRANS IN

(7) TRANS OUT

(8) ERROR CODE

(15) PG IN

(17) L3 2 L2 PG

(18) L3 2 L1 PG

(19) L2 2 L3 PG

(20) L1 2 L2 PG

(22) L1 2 L3 PG

(23) L2 2 L1 PG

(24) SPOOL SOURCE

(25) NEXT JOB

(26) CREATE ODF

(27) GET ODF

(28) CREATE OUT FILE

System Configuration For Page Option 2

(1) MAIN DRIVER

(13) PG OPTION

(2) SYS GEN

(14) INIT PG OPTION

(16) EOJ

(29) STACK DEPTH COUNTS

(3) INPUT OUTPUT

(4) DATA IN

(5) DATA OUT

(6) TRANS IN

(7) TRANS OUT

(8) ERROR CODE

(15) PG IN

(17) L3 2 L2 PG

(18) L3 2 L1 PG

(19) L2 2 L3 PG

(20) L1 2 L2 PG

(22) L1 2 L3 PG

(23) L2 2 L1 PG

(24) SPOOL SOURCE

(25) NEXT JOB

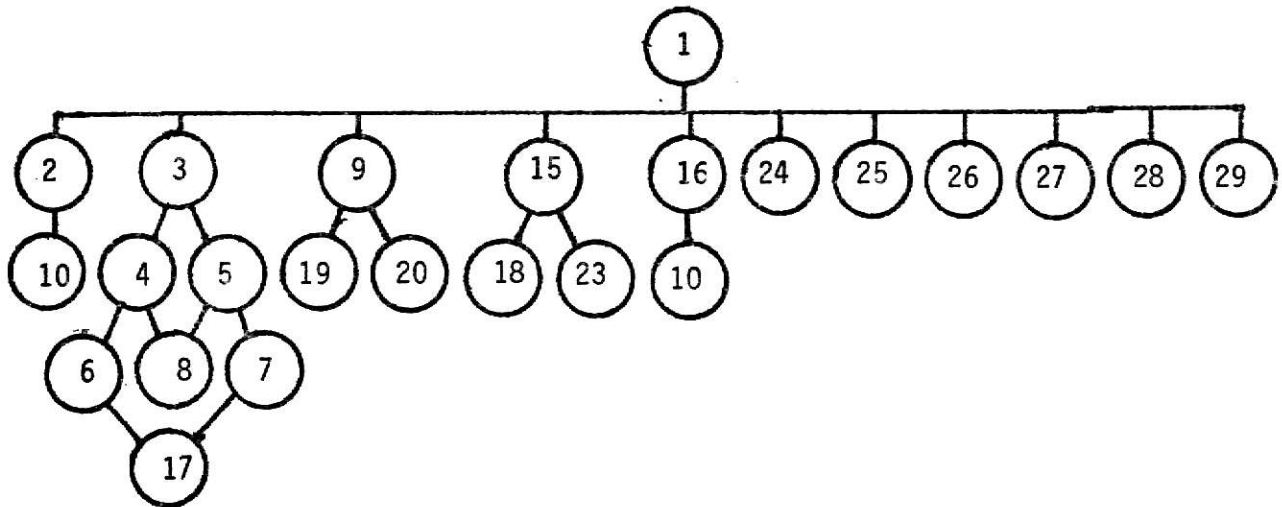
(26) CREATE ODF

(27) GET ODF

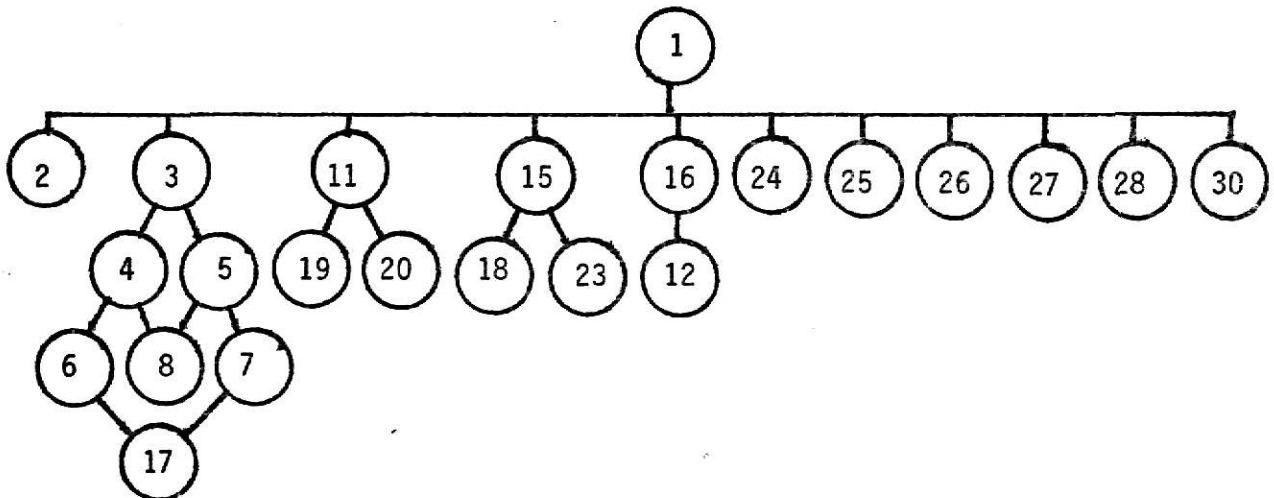
(28) CREATE OUT FILE

System Configuration For Page Option 3

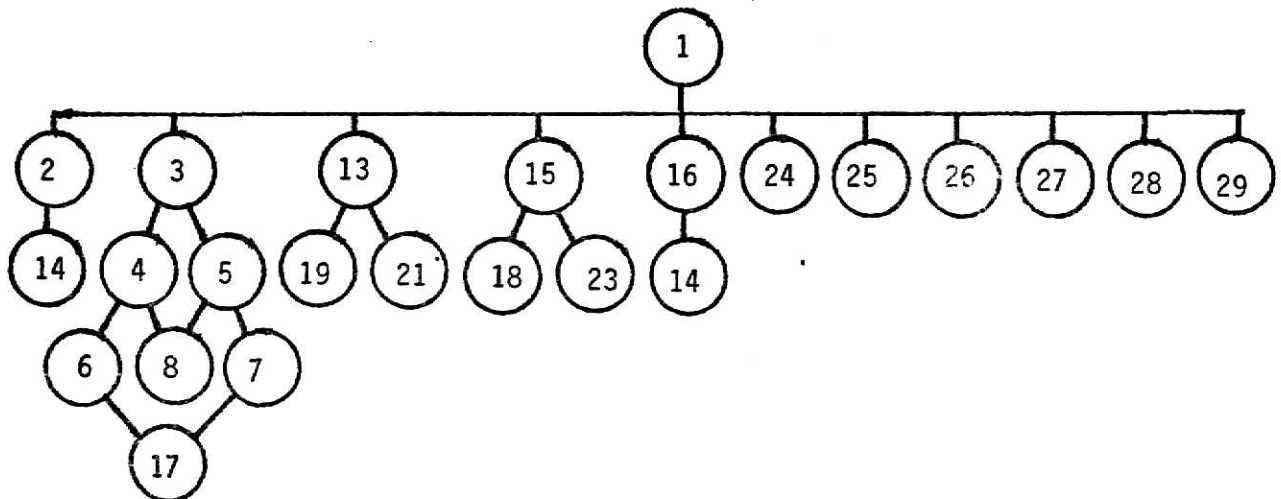
Page Option 1 Calling Order



Page Option 2 Calling Order



Page Option 3 Calling Order



REFERENCES

- 1) Madnick, Stuart E. and Donovan, John J., Operating Systems, McGraw-Hill Computer Science Series, 1974.
- 2) Denning, Peter J., "Virtual Memory", ACM Computing Surveys, Vol. 2, No. 3, pp. 153-190, Sept. 1970.
- 3) Stevens, Myers, and Constantine, "Structured Design", IBM Systems Journal, Vol. 13, No. 2.
- 4) Dijkstra, E. W. "The Structure of the T.H.E. Multiprogramming System", CACM, Vol. 11, No. 5, pp. 341-346, May 1968.
- 5) Smith, Douglas E., "HIMICS: A Virtual Memory Environment for Mini-Computers and a Description of its Level 1 Processor", Computer Science Department, KSU, Manhattan, Kansas.
- 6) Pankhurst, R.J., "Program Overlay Techniques," CACM, Vol. 11, No. 2, pp. 119-125, Feb. 1968.
- 7) Randell, B. and Kuehner, C. J., "Demand Paging in Perspective," Proceedings, AFIPS, 1968, FJCC, Vol. 33, Pt. 2, pp. 1011-1018.
- 8) Denning, Peter J., "On Modeling Program Behavior," Spring Joint Computer Conference, 1972.
- 9) Ferrari, Domenico, "Improving Locality by Critical Working Sets," CACM, Vol. 17, No. 11, pp. 614-620, Nov. 1974.
- 10) Mattson, R. L., Gecsei, J., Sultz, D. R., and Traiges, I. L., "Evaluation Techniques for Storage Hierarchies," IBM Systems Journal, Vol. 9, No. 2., 1970.
- 11) Adnerson, Gary, "Hierarchical Structure," Computer Science Department, Kansas State University, Manhattan, Kansas.
- 12) Katzan, H. Jr., "Storage Hierarchy Systes," Proceedings, AFIPS, 1971, SJCC, Vol. 38, pp. 325-336.
- 13) Denning, P. J., "The Working Set Model for Program Behavior," CACM, Vol. 11, No. 5, pp. 323-333, May 1968.
- 14) Denning, P. J., "Thrashing: Its Causes and Prevention," Proceedings, AFIPS, 1968, FJCC, Vol. 33, pp. 915-922.
- 15) Denning, P. J., "Properties of the Working Set Model," CACM, Vol. 15, No. 3, pp. 191-198, Mar. 1972.

- 16) Data General Corp. Manual, Real Time Disk Operating System User's Manual (RDOS), Data General Corp. Publication Number 093-000075-04, 1973.
- 17) Interdata Inc. Manual, 05/16 Multi-Tasking Operating System Reference Manual, Interdata Inc. Publication Number 829-367, 1974.

HIMICS: A VIRTUAL MEMORY ENVIRONMENT FOR MINI-COMPUTERS AND A
DESCRIPTION OF ITS LEVEL 2 PROCESSOR

by

ARLAN E. BENTZ

B.S., Kansas State University, 1968

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1975

The HIMICS system is a hierarchical virtual memory system for a hierarchy of interconnected mini-computers. This paper describes the design of the software system. The software system design in this paper is a hierarchical design with two major processor levels. An overall description of both processors is given and then a detailed description of its level 2 processor is presented. The detailed description includes the algorithms, written in a dialect of PL/1, along with a written description of them. The HIMICS system will provide a virtual memory system for a network of mini-computers and also allow the emulation of high level languages. The implementation of this system should result in an increase of processor efficiency and system throughput for the mini-computers involved in the network. The paper is concluded with a dialectic comparison of a single processor system versus a multi-processor system.