THE DEVELOPMENT OF A PARTITIONED SEGMENTED MEMORY MANAGER FOR THE UNIX OPERATING SYSTEM

Harvey William Emery

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DEVELOPMENT OF A
PARTITIONED SEGMENTED MEMORY MANAGER
FOR THE
UNIX OPERATING SYSTEM

by

Harvey William Emery, Jr.

June 1976

Thesis Advisor: G. L. Barksdale, Jr.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Development of a Partitioned Segmented Memory Manager for the UNIX Operating System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1976 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Harvey William Emery, Jr.		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1976 13. NUMBER OF PAGES 91
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

UNIX
Memory Management
Minicomputer
PDP-11

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis reports the results of an investigation of the applicability of paging and segmentation to memory management in modified UNIX operating systems on the PDP-11/50 minicomputer system at the Naval Postgraduate School Signal Processing and Display Laboratory. Two memory managers are specifically considered: a partitioned segmented memory manager that was designed and implemented; and a simpler, segmented memory

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 68 IS OBSOLETE
S/N 0102-014-6601 |
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (cont.)

manager that was designed based on the performance of the partitioned segmented memory manager. Recommendations are given for future work.

The Development of a Partitioned Segmented Memory Manager for the UNIX Operating System

by

Harvey William Emery, Jr.
Captain, United States Marine Corps
B. S., Massachusetts Institute of Technology, 1968

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
JUNE 1976

ABSTRACT

This thesis reports the results of an investigation of the applicability of paging and segmentation to memory management in modified UNIX operating systems on the PDP-11/50 minicomputer system at the Naval Postgraduate School Signal Processing and Display Laboratory. Two memory managers are specifically considered: a partitioned segmented memory manager that was designed and implemented; and a simpler, segmented memory manager that was designed based on the performance of the partitioned segmented memory manager. Recommendations are given for future work.

CONTENTS

List of Tables

List of Figures

Acknowledgment

I wish to express my gratitude to two individuals who made special contributions to the success of this work. First, I wish to thank my wife, Vicki, who not only provided constant encouragement but also spent many hours editing this document. Second, I wish to thank my advisor, Professor Barksdale, who was always ready to lend me a helping hand when I needed it. The time he contributed, including weekends and evenings, was above and beyond the call of duty.

I. INTRODUCTION

The Naval Postgraduate School Signal Processing and Display Laboratory [1] is used for research and education in the areas of operating systems, computer graphics, signal processing and hybrid computing. The laboratory's equipment configuration is illustrated in Fig. 1. The system is built around two Digital Equipment Corporation PDP-11/50 processors that share some memory, some peripherals, and access to a signal processing subsystem consisting of a Computer Signal Processors Incorporated CSP 125 controller with an array processor and an analog to digital converter. The peripheral suite may be divided into two major groups: the data acquisition group and the display group. The display group consists of dynamic graphics display units that are designed to support real-time, interactive applications. The data acquisition group consists of terminals, disks, tapes, and unit record equipment that serve the dual purposes of supporting a healthy environment for program development and providing for the acquisition of data for the graphics and signal processing applications.

Choosing an operating system for the laboratory presented significant challenges. Traditionally, real-time operating systems have provided poor environments for program development. The operating systems that are

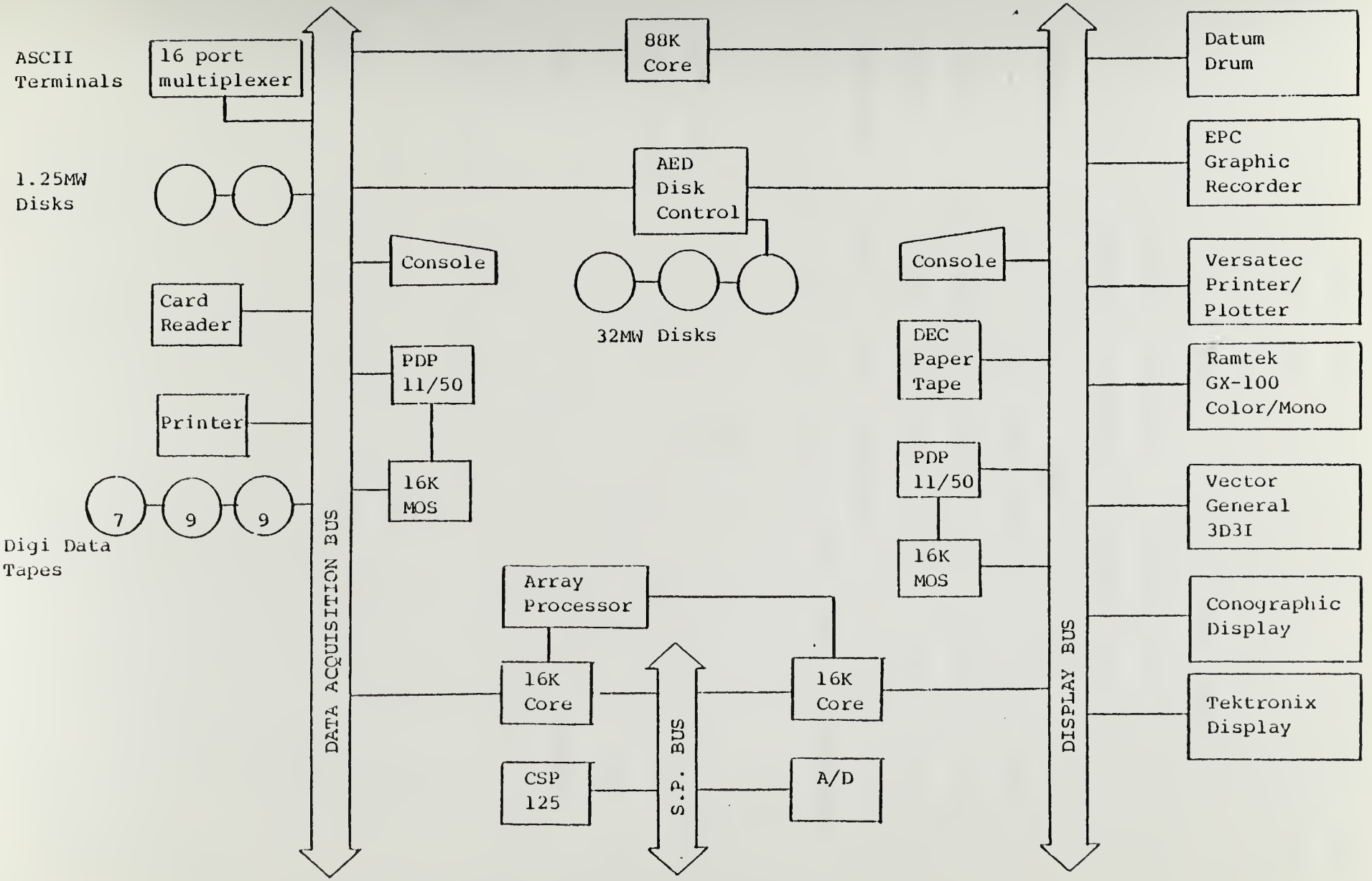

Figure 1. Equipment Configuration

responsive to the demands of program development have provided very poor real-time and interactive environments. When the equipment was acquired, no single PDP-11 operating system met all the needs envisioned for the laboratory. It would not have been surprising if the decision had been made to support separate operating systems tailored to the demands of the two areas. Because of the difficulties inherent in maintaining and scheduling multiple operating systems on one computer system, it was decided to attempt to develop a unified operating system having specialized subsystems to support both foreground real-time, interactive processing and background program development processing.

The baseline operating system selected was UNIX, a time-sharing system developed at Bell Laboratories. One of the important advantages of UNIX was that source code in a high level language was furnished with the system. The availability of source code and UNIX's excellent support for program development promised a climate favorable to the anticipated system modifications.

The UNIX system has proved to be a good selection. Several projects designed to augment UNIX are in progress or have been completed. One area of particular concern has been memory management. Figure 1 reveals that the several different types of memory in the configuration present some unusual memory management problems; but the figure does not reveal the complex memory management problems introduced by the real-time applications, especially those involving

direct memory access by display devices. Some of these problems have been approached in earlier work [2], [3], [4]. One area of interest that had not been investigated prior to this thesis was the applicability of advanced memory management techniques to the laboratory operating system. This area was particularly attractive because the PDP-11/50 processors have a Memory Management Unit capable of supporting relocation, paging, and some segmentation. The purpose of this thesis is to present the results of an investigation into the suitability of segmentation and paging in the modified UNIX operating evironment at the Naval Postgraduate School Signal Processing and Display Laboratory.

II. THE PDP-11/50

A. GENERAL ARCHITECTURE

The PDP-11/50 [5] is a powerful, medium scale, general purpose, 16-bit minicomputer. It is well designed to support multiprogramming or real-time applications. Its features include a priority interrupt structure, two general purpose register sets, and three processor states (Kernel, Supervisor, and User). Two of the most important aspects of the PDP-11 architecture are its input/output scheme and its dependence on processor stacks. Both of these have important impacts on the memory management methods considered in this thesis.

The most important component of the PDP-11 input/output scheme is the UNIBUS. The UNIBUS is a high speed, bidirectional, asynchronous bus that connects the CPU, peripherals, and memory. Devices attach to the UNIBUS with hardware control and data registers that have simulated locations assigned in the uppermost 4,096 words of the address space. This simplifies the programming of peripheral devices because no special class of input/output instructions is required. Data and control information is entered into or retrieved from the devices' registers as if they were actual memory locations. The UNIBUS can be controlled either by the CPU or by a peripheral device.

This makes it possible for the devices to access main memory with almost no processor intervention. The arbitration unit that assigns control of the UNIBUS to a requesting device or CPU gives maximum priority to a direct memory access request from a peripheral device. Because the PDP-11/50 does not feature a lock and key memory protection scheme, the protection of the operating system from DMA devices is a significant memory management problem.

A PDP-11 stack is an area of memory set aside under program control for temporary storage, subroutine linkage, and interrupt service linkage. In concept, it is a classic "last-in, first-out" stack of the type described in ref. [6]. Each processor state has a register specifically designed to be its processor stack pointer. The intructions that are provided for standard PDP-11 routine linkage and interrupt handling "push" and "pop" parameters, linkage information, and status information, using the current processor stack. Other instructions are provided to facilitate stack manipulation. Properly used, stacks provide automatic nesting of subroutines, reentrancy, and recursion. They also help to decrease the overhead that is inherent in linkage and interrupt processing. The only major disadvantage of using stacks is that properly controlling dynamically growing and shrinking stacks is a significant memory management problem.

B. MEMORY MANAGEMENT FEATURES

1. Concepts

Even though the PDP-11/50 computer has a 16-bit word length, its basic addressing logic uses an 18-bit direct byte physical address. In the PDP-11/50 system's simplest configuration, the two most significant bits of the 18-bit address are not implemented. The address space is limited to 32K words (32 * 1,024 words). The address space is used to reference up to 28K words of main memory and the 4K words of UNIBUS device registers. To expand main memory beyond 28K bytes, the PDP-11 Memory Management Unit (MMU) must be added to the configuration. This unit interprets 16-bit addresses as virtual addresses from which it constructs 18-bit physical addresses. Up to 124K words of main memory can be made addressable with the MMU.

2. Address Formation

With the MMU installed, the PDP-11/50 supports two 32K word virtual address spaces for each of its three processor states. Each state has an Instruction Space (I-space) and a Data Space (D-space). Instruction fetches, index words, absolute addresses, and immediate operands reference I-space. All other references are to D-space. The MMU constructs physical addresses using the information in six sets of relocation and descriptor registers. The set used depends on the processor mode and the type of address reference. These registers and MMU control registers are

assigned simulated memory locations and may be accessed in the same way as UNIBUS device registers. Each register set consists of eight Page Address Registers (PARs) and eight Page Descriptor Registers (PDRs). Address formation is shown in Fig. 2. The MMU subdivides each 16-bit virtual address into three fields: the displacement in block (DIB), bits 0 to 5; the block number (BN), bits 6 to 12; and the active page field (APF), bits 13 to 15. The APF is used to select a PAR. The twelve low order bits, or page address field (PAF), of the PAR are added to the BN to form a 12-bit physical page block number (PBN). The DIB is concatenated with the PBN to form the 18-bit physical address. There are two very important implications of this scheme: the basic granularity of the memory is the 64-byte block and a page may begin on any block boundary in the memory. The concept of the page frame is not applicable to the PDP-11.

3. Access Control

The PDRs are used to control access to pages, to specify their lengths, and to provide memory management data. The format of a PDR is shown below.

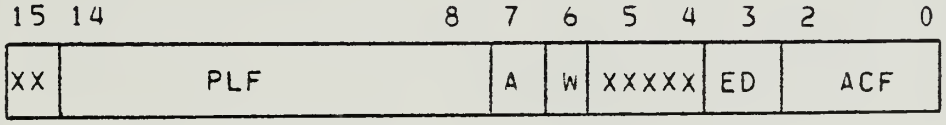

Figure 3. PDR Format

The fields in the PDR are: the access control field (ACF), bits 0 to 2; the expansion direction bit (ED), bit 3; the written bit (W), bit 6; the accessed bit (A), bit 7; and the

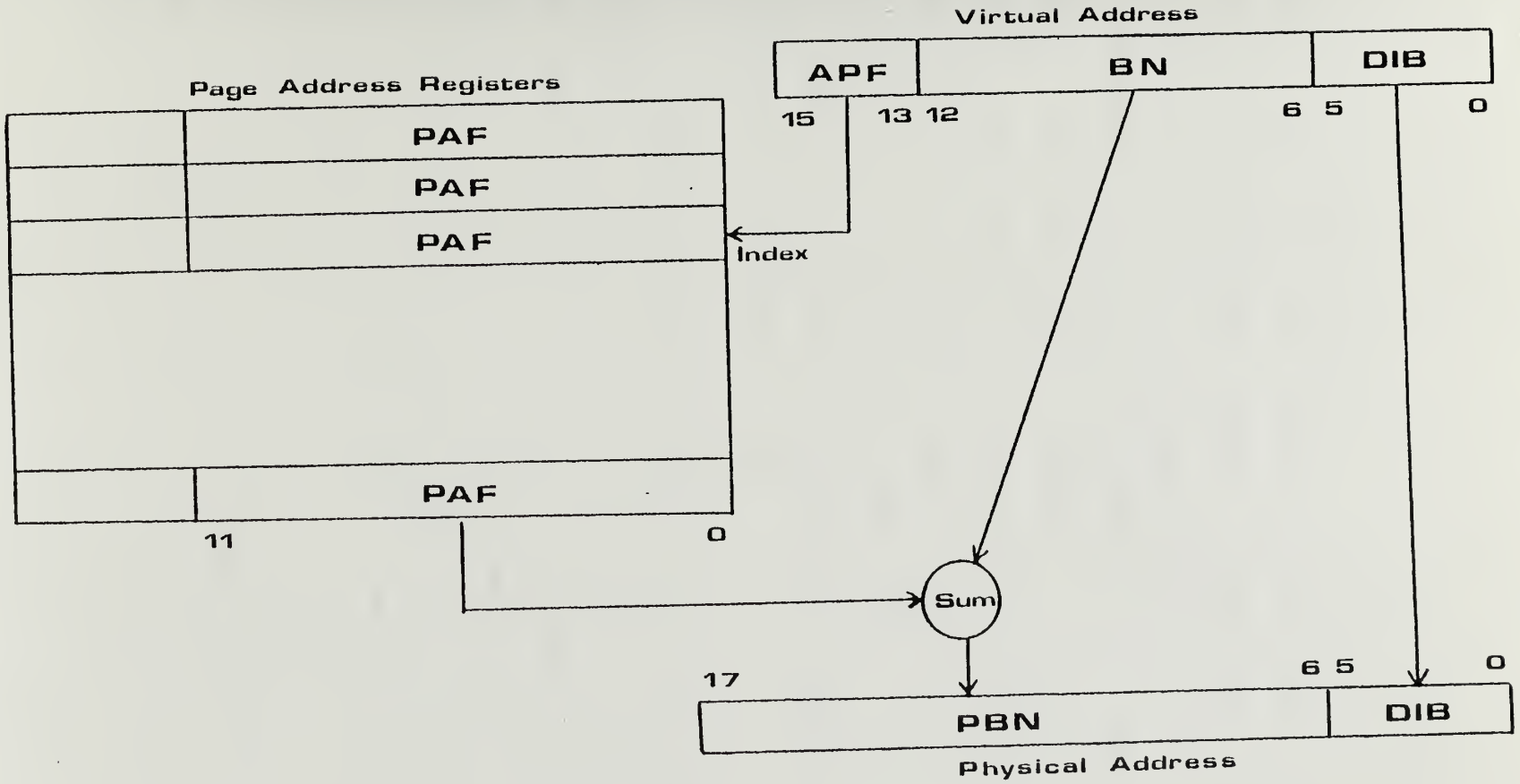

Figure 2. Physical Address Formation

page length field (PLF), bits 8 to 14. Among access options that may be specified in the ACF are: read only, abort on write; read/write, no aborts; and non-resident, abort all accesses. Because a page need not be a full 128 blocks in length, the PLF and the ED bits are used to validate the virtual BN before memory access is permitted. If the ED bit is not set, the PLF is the page length in blocks minus one. In this case, any attempt to access this page with a BN greater than the PLF is aborted. The ED bit is to be set when the page contains a stack extending downward from the upper end of the page's address range. If the ED bit is set, the PLF is 128 minus the page length in blocks. In this case, any attempt to access this page with a BN less than the PLF is aborted.

The MMU provides a mechanism by which a Kernel mode supervisory routine may be invoked when a memory management abort occurs. Enough information is preserved in MMU status registers to describe the type of abort and to identify the offending instruction and address. This feature could be used in a demand paged memory manager, for example, to resolve page faults.

The W and A bits provide page reference data for the memory management software. The W bit is set if the page has been modified since the PDR was loaded. The A bit is set if the page has been accessed since the PDR was loaded. Both bits are reset whenever the PDR is modified.

III. THE UNIX TIME-SHARING SYSTEM

A. CONCEPTS

UNIX [7] is a terminal oriented, interactive, time-sharing, operating system developed at Bell Laboratories for use on the PDP-11 family of minicomputers. Most of UNIX is written in "C," [8] a high level language also developed by Bell Laboratories. A small part of UNIX is written in "as," [9] a Bell Laboratories variant of the PDP-11 assembler language. Among the most significant of UNIX's features are: a hierarchical file system, a device independent input/output scheme, and multi-tasking.

The basic unit of work under UNIX is the process. Each process is an execution of a program file from the UNIX file system. When UNIX "bootstraps" itself into memory at system initiation time, it "handcrafts" the first two processes: process 0 and process 1. Process 0, which may be thought of as an execution of UNIX, is the master control process. Process 1 sets up the system; all subsequent processes are descendents of process 1.

All processes except process 0 execute in User processor state. If a process requires service from UNIX, it communicates its request by means of a system call. System calls are mechanized by use of TRAP instructions which

interrupt the processor, change the processor state to Kernel, and cause the approriate service routine to be executed. When the system call completes, it returns to the calling process with the processor in User state.

A process creates descendents by use of the "fork" system call. This system call creates an exact duplicate of the calling process. The new process is referred to as a child process and the original process is referred to as a parent process. The parent may continue to execute, perhaps creating more children, or it may use the "wait" system call to suspend execution until its child has completed execution. The child may continue to execute the same program as its parent or it may invoke a new program by use of the "exec" system call. A child may also create children of its own. When a child completes processing, it terminates by means of the "exit" system call. Among other actions, "exit" notifies the parent of the child's demise.

Process 1 begins its role as grandsire by creating one child for each terminal in the system. Each child opens its assigned terminal, sends a message requesting a user to log in, and awaits a reply. When a user successfully completes the log in procedure, the child invokes a new program called the Shell. The Shell interprets commands specified by the user and creates children which invoke other programs to carry out the user's commands. When the user logs off, his teminal's Shell process terminates. Process 1, which has been patiently waiting for this to happen, is notified and

it creates a new child for the terminal. The new child reopens the terminal and sends another log in request.

B. MEMORY MANAGEMENT

In concept, the UNIX memory manager is a relocatable partitioned memory manager [10] with swapping and limited segmentation. Each process has an image that must reside in a contiguous partition while the processor is executing on behalf of the process. The image remains in memory during the execution of other processes unless it must be written out (swapped out) to a direct access device to satisfy the memory requirements of a higher priority process. Relocation and storage protection are accomplished with the MMU. When the processor is executing on behalf of a process, the memory management registers are loaded so that the process can access only its own image and, if applicable, a text segment shared with other processes. Because a process executes in User mode, its address space is the User address space; however, a part of the process's image called the UVECTOR is established in Kernel D-space. The UVECTOR contains process status information needed by UNIX and the Kernel mode processor stack to be used while the process is active. Not all process control information is located in the UVECTOR. Information that must remain addressable even when the process is not executing remains resident for the life of the process in Kernel D-space in a control block called a PROC. If the process shares its text

with other processes, its PROC contains a pointer to yet another control block resident in Kernel D-space. This control block is called the TEXT. It contains information that UNIX uses to control the sharing of the text segment. APPENDIX A contains detailed information on the UVECTOR, PROC, and TEXT.

A process's image is created when "fork" copies its parent's image. Whenever a process uses "exec" to invoke a new program, the process's image is recreated according to specifications in the program file to be executed. A process's image differs depending on whether or not it shares text. The image of a text sharing process consists of its UVECTOR, data, and User mode processor stack. The shared text is established in memory independently of the images of the processes sharing it. In the image of a non-sharing process, the text is lumped together with, and considered to be part of, the data. If the process shares text, "exec" checks to see if a copy of the text is available in the system. If it is not, "exec" establishes a copy.

The User address space of a text sharing process may be established in two different ways: separated instruction and data spaces or combined instruction and data spaces. The User address space of a non-sharing process may only be established with combined instruction and data spaces. If a process's address spaces are combined, its I-space and D-space are identical. A separation flag, determined by

"exec" from the file type and kept in the process's UVECTOR, controls the method used to establish the address space of a text sharing process. If a process's address spaces are separated, its shared text segment is addressable beginning at User I-space address 0 and its data is addressable beginning at User D-space address 0. If a combined process has shared text, its shared text segment is addressable beginning at address 0 in both User I-space and User D-space; and its data is addressable beginning at the first 4K word boundary (page boundary) beyond the end of the text in both User I-space and User D-space. If a combined process does not have shared text, its text is addressable beginning at address 0 in both User I-space and User D-space; and its data is addressable beginning at the first word boundary beyond the end of the text in both User I-space and User D-space. A process's stack is addressable extending downward from the highest address in User D-space or in I-space and D-space. The access control specified in the PDRs of shared text pages is read only. All other pages, including non-shared text, are read-write access.

There are two system calls that a process may use to change the size of its image without invoking a new program. These system calls are "brk" and "sbrk" [11]. These are used to increase or decrease the size of the process's data area. They are used mainly in programs with storage requirements that have great variations depending on input. The size of a process's image may increase in another way.

If its User mode processor stack grows beyond the space initially allocated to it, UNIX dynamically increases the amount of memory provided.

C. INPUT OUTPUT SYSTEM

1. Standard Input/Output

The UNIX standard input/output (I/O) system [12] is designed to separate the user from device dependencies, to keep control blocks out of the User address space, and to preserve process relocatability. Two classes of devices are supported under the standard I/O system: character-devices and block-devices. Character-devices are read and written one byte at a time using a UNIBUS device register as an I/O port. Block-devices are read and written in 512 byte blocks using direct memory access (DMA). UNIX provides the expected system calls to create, open, access, and close files on both device types. It also provides buffer areas in Kernel D-space for character-device I/O queues and for a pool of block device buffers.

When a process requests a write to a character-device, an I/O support routine (device driver routine) moves the data to the device's output queue or directly to the device. When a process requests a character-device read, the device driver receives the data from the device and moves it to the User address space.

When a block write is requested, the device driver acquires a buffer from the pool in Kernel D-space, moves the data from the User address space to the buffer, and places the buffer on the device's output queue. When a block read is requested, the driver places the request in the device's input queue, and acquires a buffer to which the device will transfer the data when the request is honored. After the device has transferred the data to the buffer, the driver moves the data to the User address space.

The significance of standard I/O from the memory management point of view is the way in which it localizes DMA accesses to Kernel D-space. This is important because a DMA device must be given the physical address of the area from which or to which the data will be transferred. In Kernel D-space, virtual and physical addresses are identical. All addresses used to move data between the User address space and the Kernel D-space are virtual; no device or routine needs to know the physical address of the requesting process's I/O area. This means that standard I/O is completely compatible with dynamic relocation of the requesting process.

2. Raw Block-device Input/Output

Raw block-device I/O [12] is a scheme whereby I/O takes place directly between the requesting process's memory and the device. The advantages of this type I/O are that it allows the use of blocks larger than 512 bytes and that it

avoids the overhead of moving the data between the User address space and the Kernel D-space. Although it might seem that the data moving overhead would be small, this is not the case because the PDP-11 lacks a block-move instruction. The only way to move a group of data words is with a program loop. This means that several instructions must be fetched and executed to move each word of data or, in some cases, to move each byte of data. The major disadvantage of this type of I/O is that the block-device must be given the physical address of the input or output area in the User address space; thus, the requesting process must be "locked" in memory during the I/O operation. This prevents the memory manager and process controller from making optimum use of system resources.

In practice, raw I/O does not have an important effect on operating system performance because it is very rarely used. It is important because future applications will use it and because supporting it presents some difficult memory management problems.

IV. MEMORY MANAGER DESIGN

A. SEGMENTATION

The fundamental problem addressed in this thesis is a pragmatic one: how best to modify the UNIX memory manager in order to make more complete use of the capabilities of the available memory management hardware. Shared text is already well segmented in UNIX, so the first step taken was determining a good method of dividing the process image into segments. The important considerations were that the segments be logically separate and independently managable in main memory. It proved to be possible and desirable to use the natural division already discussed: UVECTOR, data (including non-shared text), and the User processor stack. This division has the appeal of being straightforward and of being an efficient way of coping with the problem of managing the dynamic size changes of the data and stack areas.

A study of the UNIX system showed that reestablishing the process image when the data or stack changes size is a major source of memory management overhead. Whenever either the stack or the data changes size, the UNIX memory manager has to acquire memory for an entire new image and copy the UVECTOR, data, and stack to it. The cost of this copying is about 10 micro-seconds of processor time per word if memory

is available for the new image, and several seconds of elapsed time if memory has to be acquired by swapping other processes out of memory. If changing size were an infrequent occurrence, this cost would not be excessive but studies have shown that some of the most commonly used programs change size often. Among these programs are: "cc," the "C" language compiler; "ed," the text editor; and "as," the assembler. With the data and stack in separate segments, overhead is reduced because only the segment that changes size has to be copied. If the segments were divided into pages, only the last page of the segment would have to be copied, reducing overhead even more.

There is another important reason for establishing separate segments for the data and the stack. Several applications [1] planned for the Signal Processing and Display Laboratory will require a memory manager that can place a process's data segment in a specific area of main storage. Specific placement will be required to reduce the overhead of processor to processor and processor to device communication and to protect processes from destruction by errant operations by DMA devices. An example is a process that requires the CSP 125 array processor. The CSP 125 can access only a portion of the memory available in the laboratory system. Any process that communicates with the CSP 125 must, therefore, have its data segment placed within the memory that the CSP 125 can access.

Another example is a real-time graphics process using the Vector General 3D3I display unit. This DMA device retrieves and interprets its display list forty times per second. Instructions in the display list can cause the device to store data anywhere within the 32K word memory segment containing the list. To protect the UVECTOR and stack of the process using this device, the display list must be isolated in a 32K word memory segment.

B. ALLOCATION OF MEMORY

After the method of segmentation was decided, the specifics of the memory allocation technique were considered. It was assumed at first that a paged segmented memory manager [13] would be the best choice. After careful consideration, a partitioned segmented memory manager was chosen. Partitioned segmentation is a variant of paged segmentation first suggested by Randall [14] based on simulation studies of simple segmentation and paged segmentation. The differences between paged and partitioned segmentation are: the basic quantum of storage allocated and the method of physical address formation required.

The allocation quantum in paged segmentation is the page frame, an area of memory large enough to hold exactly one virtual page. Each request for storage is the same size (page size) and each free area is an integral multiple of this size. This strategy reduces the loss of storage due to external fragmentation because no area of free storage is

ever too small to satisfy a request for a page of memory. It also simplifies the memory allocation and deallocation primitives because they need to respond to memory requests of only one size. The disadvantage of paged allocation is that even a partial virtual page is allocated a full page frame. The unused memory in page frames allocated to partial pages causes a loss of usable storage called internal fragmentation.

The basic idea behind partitioned segmented allocation is the reduction of internal fragmentation. A partitioned segmented memory manager allocates storage in a quantum that is smaller than, but an integral divisor of, the page size. The largest contiguous area allocated is equal to the page size; but, when memory is allocated for a partial page, only the smallest number of quanta larger than the size is allocated. Internal fragmentation still occurs but the average loss per partial page is only half a quantum rather than half a page frame. There are two disadvantages of partitioned segmentation: the memory manager must be more complex because it must respond to requests for a number of different memory sizes; and external fragmentation will occur.

Physical address formation in a paged segmented system is simple because pages reside in memory at physical addresses that are integral multiples of the page size. Because the page size is always chosen as a power of two, the physical address is formed from the virtual address by

concatenating the virtual address's displacement-in-page with the physical page frame number. In a partitioned segmented system, pages are placed in memory at physical addresses that are integral multiples of the quantum of allocation. Because the quantum of allocation is chosen as a power of two, the physical address is formed by adding the physical page quantum number to the virtual address's quantum-within-page and then concatenating the virtual address's displacement-within-quantum with the result.

Because the PDP-11/50 MMU can support either a paged or partitioned segmented memory manager, the important consideration in deciding between the two memory managers was: how the loss of storage utilization caused by external fragmentation with a partitioned segmented memory manager would compare to the loss of storage utilization caused by internal fragmentation with a paged segmented memory manager. A definitive answer to this question is not possible unless both memory managers are tested; however, it is easy to show that the storage losses with a paged segmented memory manager would be unacceptable in the UNIX environment. The argument for this premise is based on the very large (4K word) page size imposed by the memory management hardware, the extremely small number of page frames available, and the small segment sizes that result when the process image is divided into three segments. The most important consideration is the comparison between segment size and page size. If the segments were very large

compared to the pages, the percentage of partial pages would be small and the resulting loss of utilization insignificant. If the segments were small compared to the pages, almost all pages would be partial pages and the resulting loss of utilization would be high.

The UVECTOR segment is fixed in length at .5K words. The initial stack segment allocation for all processes is .64K words. Stacks grow dynamically, but observations have shown that this is a rare occurrence. Estimates of the average sizes of the text and data segments were determined by examining the program files of 70 frequently used programs. The mean text segment size was determined to be 1.9K words and the mean data segment size at the start of execution was determined to be 2.2K. Data segments frequently grow but the largest increase that has been observed is about .5K words for one phase of the "C" compiler. Even making the generous assumption that average combined data and stack growth is .5K, these figures show that the paged segmented memory manager would waste more memory than it used productively. It would allocate four pages (16K words) for the average shared text process of which an average of 5.7K words would be used.

The segment size data is completely counter to experience gained in large scale computer systems. It was completely unexpected. Because the mean segment sizes observed were all less than a page, doubts were raised about the desirability of selecting any memory management

technique more complicated than simple segmentation. In spite of the doubts, it was decided to continue with development of a UNIX with a partitioned segmented memory manager (PSUNIX). It was believed that this effort would best serve to explore all memory management options. Design work was started, however, on a version of UNIX with a simple segmented memory manager (SUNIX). SUNIX was to be a fall-back position if the performance of PSUNIX proved to be unsatisfactory.

When the partitioned segmented memory manager was chosen, the only remaining design question was the size of the quantum of allocation. The 64-byte block, the smallest quantum supported by the memory management hardware, was selected. This choice was based on the practical consideration that it required no change to the existing UNIX memory allocation and deallocation primitives.

V. MODIFICATIONS TO UNIX

A. OVERVIEW AND PHILOSPHY

Modifying the UNIX memory manager to produce PSUNIX was a formidable task that required writing approximately 500 lines of new code and comprehending and modifying existing programs totaling more than 1,800 lines of code. The approach that was taken to this problem was to avoid putting large sections of new code into existing programs. The new code that was needed to support the memory manager was centralized in several small self-contained functions (primitives). Wherever possible, changes to existing routines were limited to removing calls to the old memory management primitives and replacing them with calls to the new memory management primitives. The goals of this approach were to make the the general structure of memory management in PSUNIX seem familiar to those who understood the UNIX structure and to simplify debugging by localizing new code. These goals were realized.

The changes made to implement PSUNIX can be divided into five areas:

1. Control Block Modifications
2. Memory Management Support Modifications
3. Swap Space Allocation Modifications

4. Raw I/O Support Modifications

5. Support Program Modifications

Each of these areas is described in a subsequent section of this chapter. Supporting documentation can be found in the appendices. APPENDIX A contains detailed information on control blocks related to memory management. APPENDIX B contains detailed documentation of memory management routines found in UNIX, PSUNIX, and the proposed simple segmented version, SUNIX.

B. CONTROL BLOCK MODIFICATIONS

The only control blocks modified were the PROC (process control) and the TEXT (shared text segment control). No new control blocks were added except page tables which are allocated in temporary storage and used as work space during memory allocation and deallocation. In UNIX, the PROC contains the size and address of the image. In PSUNIX, the PROC was expanded so that it could fulfill the same role as the MULTICS [15] Segment Map Table. The image size and address were removed and the segment sizes and page addresses were added. In both versions of the operating system, the TEXT performs the same function as an entry in the MULTICS Active Segment Table. The only modification required for PSUNIX was the addition of a page address array. APPENDIX A provides detailed information on the PROC, TEXT, and several other control blocks that are important to memory management.

C. MEMORY MANAGEMENT SUPPORT MODIFICATIONS

The basic form of the memory management modifications has been presented in the previous chapter. APPENDIX B provides complete documentation of the new memory management primitives under the heading "page.c." It also provides documentation of the changes made to existing UNIX programs that call these primitives. Functions of particular interest are: "newproc," which creates a process's image by copying the image of the process's parent; "exec," which recreates a process's image when the process invokes a new program; "xalloc," which establishes shared text segments; "expand," which changes a process's image size; "sched," which swaps processes into and out of memory; "xfree," which removes shared text segments from the system; and "exit," which terminates processes and frees their resources.

D. SWAP SPACE ALLOCATION MODIFICATIONS

The UNIX method of controlling swap space is to allocate it to a process when the process is to be swapped out and to free it as soon as the process returns to memory. The advantage of this is that it keeps the demand for swap space at a minimum but the disadvantage is that it requires an allocation and a deallocation whenever a process is swapped. UNIX swap I/O is extremely efficient because the process image is contiguous in memory. This allows UNIX to swap a process with one I/O operation.

The PSUNIX method of controlling swap space is to allocate it to a process the first time the process is swapped out and to allow the process to keep the swap space when it returns to memory. A process loses its swap space when it terminates or when one of its segments becomes too large for the space that was allocated. In both versions of the operating system, the process is allocated a contiguous area of swap space. This reduces the allocation overhead. PSUNIX incurs the overhead of one I/O operation per page in the process's image. This means that PSUNIX has between three and four times as much swap I/O overhead as UNIX for the average process.

The programs concerned with swapping are documented in APPENDIX B. Functions of particular interest are: "xswap," which swaps processes out of memory; "swap," the swap I/O call; "pswap," a PSUNIX function that swaps several pages to or from memory; and "prswap," a PSUNIX function that returns a swapped process to memory.

E. RAW I/O SUPPORT MODIFICATION

As has been described previously, raw I/O requires a DMA transfer directly from or directly to a process's address space. The data is transferred to or from a contiguous area of main memory. In PSUNIX this presents a serious problem if the data area spans a page boundary because the pages will probably not be contiguous. The only efficient solution to this problem is to make the pages contiguous.

The function "physio," which sets up raw I/O transfers, was modified to determine if each transfer spans a page boundary. When a boundary spanning transfer is requested, "physio" calls "contig," a memory management primitive, that reallocates the data and stack segments of the requesting process so that both of them are allocated contiguous memory. The process is also flagged as one that requires contiguous allocation. Whenever memory is allocated for the process in the future, the segments are given contiguous areas of storage. The process remains flagged until it either invokes a new program with "exec" or terminates. APPENDIX B contains more detail on this subject.

F. SUPPORT PROGRAM MODIFICATIONS

During the course of this thesis two program development tools were perfected and two program errors in UNIX commands were encountered and corrected. These programs and corrections have been documented elsewhere and are only briefly discussed here.

The program development tools are: an assembly language program to dump memory onto the swap file without operating system support and a C language program, CRASHSAV, to retrieve the dump from the swap file and place it into the UNIX file system. The dump program was made part of the operating system. It was executed from the system control panel following a system failure to preserve the contents of memory for analysis. This system of taking and retrieving

complete memory dumps was extremely valuable. PSUNIX could not have been developed without it.

The two program errors were discovered in "nm," the command for printing symbol tables of compiled programs, and in "sysfix," the command that adjusts the format of a UNIX operating system image so that it can be "bootstrapped" into memory. The errors were discovered because the PSUNIX image exceeded 64K bytes. Neither program executed properly with the PSUNIX image as data because both programs used counters that overflowed at 64K. The problem was solved by increasing the size of the counters.

VI. PERFORMANCE STUDY

A. EXPERIMENTAL DESIGN

A series of experiments was done to determine the relative performance of the UNIX and PSUNIX versions of the operating system under a variety of operating conditions. The elapsed execution times of standard streams of processes (benchmarks) was chosen as the metric for system performance. Two benchmarks were used: a monoprogramming benchmark, BENCH1; and a multiprogramming benchmark, BENCH2. Both benchmarks contained the same processes. Most processes chosen for the benchmarks are representative of the I/O limited program development environment but several of them (notably a "Rings of Hanoi" calculation) are processor limited. APPENDIX C contains listings of the commands in BENCH1 and BENCH2. Reference [11] documents these commands.

The computer system used to execute the benchmarks was the "B" side of the laboratory configuration shown in Fig. 1. The peripheral devices used were a single terminal and two 1.25 mega-word RK05 cartridge direct access storage devices [16]. The file system for the operating system was mounted on one of the RK05 devices. To reduce I/O contention, the other RK05 device was used for swapping processes into and out of main memory. The main memory

available in the system was the parameter varied in the experiments.

B. PRESENTAION OF RESULTS

The timing data presented in this section was obtained by executing the benchmarks under control of the "time" command [11]. The times are determined by sampling the processor state at a 60 hz rate. "Real" time is actual elapsed test time reported to the nearest second. "User" time is the time the processor spent executing instructions in the User state. "Sys" time is the time that the processor spent executing instructions in the Kernel and Supervisor states. Both "User" and "Sys" times are reported to the nearest tenth of a second. The difference between "Real" time and the sum of "User" and "Sys" times is processor idle time. Earlier work [4] suggests that timings of the same benchmark may have a standard deviation of as much as 8 percent of the mean values. No statistical study of these timings was performed in the course of this thesis but limited observations indicate that the deviation is much less than 8 percent.

Six experiments were performed. The first was an execution of BENCH1, the monoprogramming benchmark, under both operating systems. The results of this experiment are shown in Table 1. The next five experiments consisted of executing BENCH2 under both operating systems, varying the amount of dynamic memory availible from 32K words to 64K

words in 8K word steps. The results of these experiments are shown in Tables 2 to 6 respectively. Combined results are shown in Fig. 4, a graph of elapsed times against dynamic memory available.

C. ANALYSIS OF RESULTS

The experimental results clearly indicate that the performance of PSUNIX and the performance of UNIX are almost identical over a wide range of sizes of available dynamic memory. Both the amounts of processor idle time and of supervisory overhead are approximately equal in all corresponding experiments. The approximate equality of idle times indicates that the disadvantage of increased swapping overhead in PSUNIX is offset by a reduction in the number of processes swapped because of reduced external fragmentation. The approximate equality of the supervisory overhead indicates that the advantage of reduced segment copying in PSUNIX is offset by the increased complexity of the memory management routines.

	UNIX	PSUNIX
REAL	3:32.0	3:29.0
USER	1:30.3	1:29.8
SYS	29.7	31.2

Table 1. BENCH1, 32K Words

	UNIX	PSUNIX
REAL	4:17.0	4:20.0
USER	1:32.5	1:31.6
SYS	30.6	33.0

Table 2. BENCH2, 32K Words

	UNIX	PSUNIX
REAL	3:50.7	3:49.0
USER	1:31.7	1:31.8
SYS	31.5	32.1

Table 3. BENCH2, 40K Words

	UNIX	PSUNIX
REAL	3:49.0	3:38.0
USER	1:32.0	1:30.8
SYS	32.1	33.0

Table 4. BENCH2, 48K Words

	UNIX	PSUNIX
REAL	3:38.0	3:34.0
USER	1:31.3	1:31.3
SYS	31.9	32.8

Table 5. BENCH2, 56K Words

	UNIX	PSUNIX
REAL	3:32.0	3:29.0
USER	1:30.5	1:30.8
SYS	30.1	32.2

Table 6. BENCH2, 64K Words

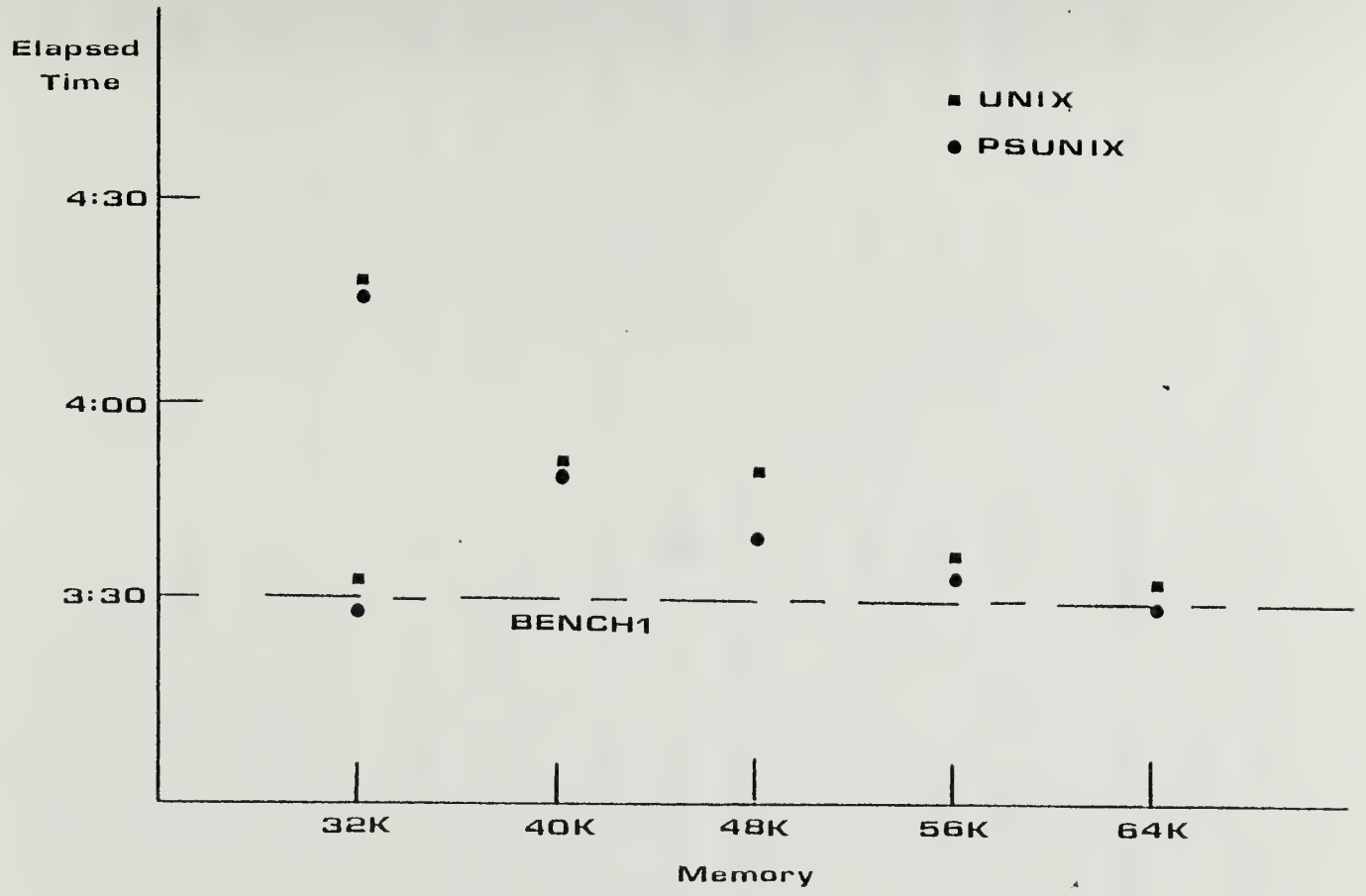

Figure 4. Experimental Results

VII. CONCLUSIONS AND RECOMMENDATIONS

The most interesting result of this thesis is that it confirms the axiom that a simple program is very often a good program. From the standpoint of performance alone, the simpler UNIX memory manager is as efficient as PSUNIX. PSUNIX is attractive because it provides an additional feature, segmentation, with no performance penalty. Although PSUNIX could be placed into production use at once, it would probably be a better idea to proceed with the development of SUNIX. From the data presented on segment sizes, it appears that SUNIX will perform at least as well as PSUNIX. SUNIX memory management routines will have the advantages of being both smaller and simpler. Because of this, they will require less storage, be easier to maintain, and cause less memory management overhead.

No matter which version is implemented, many parts of the memory manager will remain potential candidates for improvement. The basic memory allocation primitive, "malloc," is an example. In ref. [6], Knuth suggests many possible improvements to the simple algorithm used in "malloc." Although Knuth's logic is compelling, we may again be surprised by the correlation between simplicity and goodness.

A more important project will be the development of the system calls that will be used to place a process's data segment in specific areas of memory. Successful completion of this project will be required before the full benefits of segmentation will be attained.

APPENDIX A: MEMORY MANAGEMENT CONTROL BLOCKS

A. DOCUMENTATION FORMAT

This appendix contains documentation of the control blocks in the UNIX, SUNIX, and PSUNIX versions of the operating system that are direcly or indirectly concerned with memory management. The source code for these control blocks is found in files in the directory /usr/sys. Each control block is documented under an upper case roman letter. The name of the source code file containing the control block is noted following the control block name. The documentation of each control block is divided into two subsections: overview and significant data elements. Only the data elements significant to memory management are documented. Because this appendix was designed to be used with a copy of the system code, the data elements are documented in the order in which they appear in the source.

In accordance with the non-disclosure terms of the software agreement with Western Electric, program listings are not provided as part of this thesis. Authorized users of the UNIX operating system may obtain machine-readable copies of programs produced for this thesis by contacting the Naval Postgraduate School.

B. COREMAP, systm.h

1. Overview

COREMAP is an array of structures that keeps track of the unallocated areas of main storage. Each structure contains the starting physical block number and the size of an unallocated storage area. COREMAP is sorted so that its entries are in physical block number sequence. See the documentation of "malloc.c" in APPENDIX B for descriptions of the memory management primitives that manipulate COREMAP.

2. Significant Data Elements

a. char *m←size

This is the size of the free area in 64-byte blocks.

b. char *m←addr

This is the memory physical block number of the start of the free area. If this field is zero, it marks the end of COREMAP.

C. SWAPMAP, systm.h

1. Overview

SWAPMAP is an array of structures that keeps track of the unallocated areas on the swap device. Each structure contains the starting physical sector number and the size of

an unallocated area. SWAPMAP is sorted so that its entries are in physical sector number sequence. The same primitives used to maipulate COREMAP are used for SWAPMAP. See APPENDIX B.

2. Significant Data Elements

a. char *m←size

This is the size of the free area in 256-word sectors.

b. char *m←addr

This is the memory physical sector number of the start of the free area. If this field is zero, it marks the end of SWAPMAP.

D. PROC, proc.h

1. Overview

The array "proc" is an array of structures referred to as PROCs in this thesis. One of these structures is allocated to each active process in the system for the life of the process. The array is located in the Kernel address space and is permanently resident in main memory. A process's PROC contains all process information that cannot be swapped out of main memory.

2. Significant Data Elements

a. char p←flag

This is a word of flags. Bit 0 of this word is the SLOAD flag. If it is set, the process is in main memory. Bit 2 of this word is the SLOCK flag. If it is set, the process is locked in memory and may not be swapped out. Bit 4 of this word is the SSWAP flag; if it is set, the process is being swapped out. In PSUNIX, bit 15 of this word is the CONT flag. If the bit is set, it means that both the process's data and the process's stack must be contiguous in main memory.

b. int p←addr

This field is present only in UNIX. If the process is resident in main memory, it is the physical block number of the process's UVECTOR. If the process is swapped out, it is the swap device block number of the swapped image.

c. int p←size

This field is present only in UNIX. It is the size of the process's swappable image measured in 64-byte blocks.

d. int *p←textp

This is a pointer to the process's TEXT. If the value is zero, the process does not have shared text.

e. int p->cadar

This field is present only in SUNIX and PSUNIX. It is the main memory physical block number of the process's UVECTOR if the process is in memory.

f. int p->padar[8]

This array is present only in PSUNIX. The integers in the array are the memory physical block numbers of the process's pages. The order of pages is: first data pages from low to high virtual address and then stack pages from high to low virtual address.

g. int p->dadar

This field is present only in SUNIX and PSUNIX. It is the swap device block number of the process's swap space. If it is zero, the process has no swap space.

h. int p->dsize

This field is present only in SUNIX and PSUNIX. It is the size of the process's data in 64-byte blocks.

i. int p->ssize

This field is present only in SUNIX and PSUNIX. It is the size of the process's stack in 64-byte blocks.

E. TEXT, text.h

1. Overview

The array "text" is an array of structures called TEXTs in this thesis. Each segment of sharable code is assigned a TEXT for the life of the segment. The text contains all data on the usage and status of the segment. The "text" array is found in the Kernel address space and is permanently resident in main memory.

2. Significant Data Elements

a. int x←daddr

This is the swap device block number of the text segment's image.

b. int x←caddr, int xcaddr[8]

In UNIX and SUNIX, this is the memory physical block number of the start of the text segment. In PSUNIX this array contains the memory physical block numbers of the pages of the text segment.

c. int x←size

This is the size of the text segment in 64-byte blocks.

d. char x^count

This is the number of processes sharing the text segment.

e. char x^ccount

This is the number of memory resident processes using the text segment.

F. UVECTOR, user.h

1. Overview

The structure "user" is referred to as the UVECTOR in this thesis. One of these structures is part of the swappable image of each process. The UVECTOR contains all process data that is not needed when the process is not in control of the processor. When the process is in control of the processor, the process's UVECTOR resides at virtual Kernel data space address 140000 octal. The portion of the UVECTOR not used by process data elements is used as the Kernel mode processor stack.

2. Significant Data Elements

a. int u^uisa[16]

In UNIX this array contains the 64-byte block displacements from the start of the region of the process's data and stack pages. In SUNIX and PSUNIX this array is not used.

b. int u.u_uisd[16]

This array contains the prototypes of the process's user I-space and D-space page descriptor registers.

c. int u.u_tsize

This is the size of the process's shared text segment.

d. int u.u_dsize

This is the size of the process's data.

e. int u.u_ssize

This is the size of the process's stack.

G. PAGET, page.h

1. Overview

An array of "paget" structures is referred to as a PAGET in this thesis. This is a page table containing the sizes and addresses of the process's pages. A PAGET is always organized so that data pages appear first from low to high virtual address followed by stack pages from high to low virtual address.

2. Significant Data Elements

a. int t←paddr

This is the memory physical block number of the start of the page.

b. int t←psize

This is the size of the page in 64-byte blocks.

APPENDIX B: MEMORY MANAGEMENT ROUTINES

A. DOCUMENTATION FORMAT

This appendix contains documentation of functions within the UNIX, SUNIX, and PSUNIX versions of the operating system that are directly or indirectly concerned with memory management. Because this appendix is designed to be used with a copy of the source code, the documentation is divided into sections that correspond to the divisions of the source code. The UNIX source is divided into several blocks of code containing related functions. These blocks of code are stored as files in two directories: /usr/sys/dmr for device management functions and /usr/sys/ken for the remainder of the system. The documented functions in each block of code are grouped under an upper case roman letter. Each function within each block is listed under an arabic number. The functions are documented in the same order in which they are found in the code blocks with any new functions appearing last. The documentation of each function is divided into the following subsections: parameters, functional description, returned values, and error conditions. Any differences in function documentation among the three versions of the operating system are noted in the appropriate subsection.

B. main.c

1. sureg()

a. Parameters

The current process's UVECTOR and PROC are implied parameters.

b. Functional Description

This function loads the User descriptor and address registers in the memory management unit. The descriptor registers are obtained from the array u←uisd[] in the current UVECTOR. Address register loading is controlled by the values of u←tsize, u←dsize, u←ssize, and u←sep in the current UVECTOR. In UNIX the page address registers are determined based on: the region address, p←addr, in the current PROC; the text segment address, x←caddr, in the current TEXT; and the page displacements in the array u←uisa[] in the current UVECTOR. The page displacements are not used in SUNIX or UNIX. In SUNIX the segment addresses, p←daddr and p←saddr, are used instead of p←addr. In PSUNIX the page addresses in the array p←paddr[] in the PROC and x←caddr[] in the TEXT are used.

c. Returned Values

The values returned by this function are not checked.

d. Error Conditions

This function has no error conditions.

2. estabur(nt,nd,ns,sep)

a. Parameters

The first three parameters are the sizes of the current process's data and stack in 64-byte blocks. The value of "sep" is a flag that is set if the process has split instruction and data space. The current process's UVECTOR is an implied input.

b. Functional Description

This function first checks the validity of its arguments. It loads the prototypes of the memory management page descriptor registers into the array u.u_uisd[] in the current UVECTOR. In UNIX it also loads page start displacements in blocks measured from the beginning of the region or text into the array u.u_uisa[] in the current UVECTOR. In both SUNIX and PSUNIX, u.u_uisa[] is not loaded; the values of the parameters are placed in the variables u.u_tsize, u.u_dsize, u.u_ssize, and u.u_sep in the current UVECTOR. In all versions, "sureg()" is called to load the actual memory management registers.

c. Returned Values

If the parameters are invalid, minus one is returned; otherwise, zero is returned.

d. Error conditions

The minus one return indicates an error to the caller.

3. nseg(n)

a. Parameters

The parameter is a number of memory blocks.

b. Functional Description

This function calculates the number of pages, rounded up, in the number of blocks specified in the parameter.

c. Returned Values

The returned value is the number calculated.

d. Error Conditions

This function has no error conditions.

4. cksize(nt,nd,ns,sep)

a. Parameters

See "estabur(nt,nd,ns,sep)."

b. Functional Description

This function is present only in SUNIX and PSUNIX. It checks its parameters to see if they are valid.

c. Returned Values

This function returns minus one if the parameters are invalid and zero if they are valid.

d. Error Conditions

A minus one return indicates an error to the caller.

C. malloc.c

1. malloc(mp,size)

a. Parameters

The parameters are a pointer to a map array and a size specified in blocks to be allocated from the map.

b. Functional Description

This function allocates space in main memory and on the swap device. If memory is to be allocated, the first parameter must point to COREMAP. If swap space is to be allocated, it must point to SWAPMAP. The amount of space needed must be specified in 64-byte blocks if memory is to be allocated and in 256-word sectors if swap space is to be allocated.

c. Returned Values

This function returns the physical block number of the space if allocation is successful and zero if

allocation is unsuccessful.

d. Error Conditions

A return value of zero indicates allocation failure to the caller.

2. mfree(mp,size,aa)

a. Parameters

The first two parameters are the same as those of "malloc". The third parameter is a physical block number of an area of main storage or swap space.

b. Functional Description

This function frees the specified area of main storage or swap space. If memory is to be freed, the first parameter must point to COREMAP. If swap space is to be freed, it must point to SWAPMAP. The size must be specified in 64-byte blocks if memory is to be freed and in 256-word sectors if swap space is to be freed.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

D. sig.c

1. core()

a. Parameters

The current process's UVECTOR, PROC, TEXT and address space are implied parameters.

b. Functional Description

This function creates a memory image file consisting of the current process's UVECTOR, data, and stack. In UNIX this function uses "estabur" to redefine the process's virtual address space to make the data and stack contiguous. It then writes the data and stack in one output operation. In SUNIX and PSUNIX this is impossible because the data and stack may not be physically contiguous. Two output operations are used, one output operation for the data and one for the stack. If an output error occurs an indication is left in u.error in the current UVECTOR.

c. Returned Values

This function returns zero if it is successful and one if an output error occurs.

d. Error Conditions

The one return indicates an error to the caller.

2. grow(sp)

a. Parameters

The parameter is the value of the current process's User stack pointer. The current process's UVECTOR and PROC are implied parameters.

b. Functional Description

This function is called asynchronously when the current process's stack attempts to expand beyond the size allocated to it. This function calculates the number of blocks that the stack must be increased, validates the new stack size, and acquires the memory that is needed. In UNIX "expand" is called to establish the new, larger address space. In PSUNIX "sexpand" is called to establish the new, larger stack. In SUNIX this function attempts to acquire space for the new stack. If it is unsuccessful, it calls "ceswap" to acquire the space. If it is successful, it copies the old stack to the new and frees the old memory. In all versions the newly acquired space is cleared and "estabur" is called to reload the memory management registers.

c. Returned Values

This functon returns a zero if it is unsuccessful and a one if it is successful.

d. Error Conditions

A zero return indicates an error to the caller.

E. slo.c

1. sched()

a. Parameters

The PROCs and TEXTs of all processes are implied parameters.

b. Functional Description

This function searches for swapped out processes that "deserve" to be returned to memory. It selects the most "deserving" process; acquires space for it by swapping out other processes, if necessary; and returns it to main memory. In SUNIX and PSUNIX two new functions are used: "pralloc" to acquire main memory for the process and "prswap" to swap it in.

c. Returned Values

This function does not return. It is the basic instruction loop of Process 0. It goes to sleep and is reawakened about once per second by the clock function.

d. Error Conditions

In UNIX, if a swap input or output error occurs, the message "swap error" will be sent to the console and the

system will crash.

2. newproc(nrp)

a. Parameters

The parameter is a pointer to a PROC to be established for a child process. The current process's UVECTOR, PROC, and TEXT are implied parameters.

b. Functional Description

This function creates an exact duplicate of the current process as a child of the current process. It first makes the appropriate entries in the child and parent PROCs and in the TEXT if one exists. It then attempts to acquire memory for the child. If it is successful, it simply copies the parent's image to the new memory. If it fails, it swaps out a copy of the parent's image to be returned to memory as the child. In SUNIX and PSUNIX, a new function, "pralloc," is used in the attempt to acquire memory for the child. In PSUNIX a new function "prcopy" is used to copy the parent's image.

c. Returned Values

This function returns zero to the parent process. The return to the child does not come from this function but from the scheduling function "swtch". The child can identify itself as the child because "swtch" returns a one to it. This is one of the most important and

subtle phenomena in UNIX.

d. Error Conditions

If the PROC pointed to by the parameter is already allocated to an active process, the message "no procs" will be sent to the console and the system will crash.

3. expand(newsize), expand(newd, news)

a. Parameters

In UNIX this function is called with a single parameter, the new region size. In SUNIX and PSUNIX this function is called with two parameters: the new data size and the new stack size. The current process's PROC and UVECTOR are implied parameters.

b. Functional Description

In UNIX this function is called whenever the size of the current process's address space changes. It puts the new size in p←size in the current PROC. If the new size is smaller, it frees the unwanted storage. If the new size is larger, it attempts to acquire a new region for the process. If it succeeds, it copies the process's image to the new region. If it fails, it causes the process to be swapped out with the new size noted in its PROC. When the process returns to memory, it will return at the new size. If the process is swapped out, this function calls "swtch"

to change current processes. In SUNIX and PSUNIX, this function is called only to acquire an address space for a process that currently has only a UVECTOR. In these systems it puts the two new sizes in p←dsize and p←ssize in the current PROC. In UNIX the function "xswap" is called to swap out the process; in SUNIX and PSUNIX "ceswap" is used. In UNIX, "sureg" is called to load memory management registers; in SUNIX and PSUNIX, this is not necessary.

c. Returned Values

The return codes of this function are not checked. The caller has no way of determining if the process was increased in size directly or by swapping. In UNIX, if the process is swapped, this function does not return to its caller. The return comes from a subsequent call to "swtch" after the process has returned to memory.

d. Error Conditions

This function has no error conditions.

4. ceswap(ods,oss)

a. Parameters

The parameters are the current process's data and stack sizes in 64-byte blocks.

b. Functional Description

This function is present only in SUNIX and PSUNIX. It is called to do the housekeeping that is necessary for expansion swapping. It calls "xswap" to perform the actual swapping and then it calls "swtch" to place a new process in control of the processor.

c. Returned Values

This function does not return to its caller. The return to the caller comes from a subsequent call to "swtch" after the process has returned to memory.

d. Error Conditions

This function has no error conditions.

5. swfree(rp)

a. Parameters

The parameter is a pointer to a PROC.

b. Functional Description

This function frees any swap space belonging to the process indicated by the parameter, and it zeroes out p←daddr, the pointer to the space in the PROC.

c. Returned Values

The values returned by this function are not checked.

d. Error Conditions

This function has no error conditions.

6. sexpand(news)

a. Parameters

The parameter is the new, larger stack size.

b. Functional Description

This function is present only in PSUNIX. It is called from "grow" to increase the stack size. See the description of "expand".

c. Returned Values

See "expand".

d. Error Conditions

See "expand".

7. dexpand(newd)

a. Parameters

The parameter is the new, larger data size.

b. Functional Description

This function is present only in PSUNIX. It is called from "sbreak" to increase the data size. See the description of "expand".

c. Returned Values

See "expand".

d. Error Conditions

See "expand".

8. contig(rp)

a. Parameters

The parameter is a pointer to the current process's PROC.

b. Functional Description

This function is present only in PSUNIX. It is called from "physio" to make both the stack and the data of the current process physically contiguous in main storage. It indicates that the process requires contiguous segments by setting the CONT bit in the p_flag in the process's PROC; then it attempts to acquire physically contiguous main storage with "palloc". If it succeeds, it copies the old noncontiguous pages to the new contiguous ones. If it fails, it calls "ceswap" to swap out the process. When it returns to main memory, the CONT flag will cause its storage to be allocated contiguously.

c. Returned Values

The values returned by this function are not checked.

d. Error Conditions

This function has no error conditions.

F. sys1.c

1. exec()

a. Parameters

The current process's UVECTOR, PROC, and TEXT are implied parameters. Because this function is a system call, the array u.u_arg[] in the UVECTOR contains additional arguments. See Ref. [11].

b. Functional Description

This system call is used by the current process to invoke a new program. It copies any program arguments to a buffer, unlinks from the old TEXT, frees its old main storage, establishes a new TEXT if the new program has shared code, acquires storage for the new data and stack, clears the region acquired, reads in the new data, copies the arguments to the new stack, and changes the memory management registers to make them compatible with the new address space. In UNIX "expand" is used to free the old main storage; in SUNIX and PSUNIX a new function, "prfree", is used. In UNIX "estabur" is used to validate the storage requirements of the new program; in SUNIX and PSUNIX "cksize" is used.

c. Returned Values

This system call returns to the caller only if it encounters an error. If no error has occurred, it returns to the first instruction of the new program.

d. Error Conditions

This function returns to the caller if the storage requirements of the new program are invalid.

2. exit()

a. Parameters

The current process's UVECTOR, PROC, and TEXT are implied parameters.

b. Functional Description

This function is the system call used to terminate the current process. It clears signals, closes any open files, unlinks from the current TEXT, acquires a block on the swap device, copies the first 256 bytes of the current UVECTOR to the block, and frees main storage. In SUNIX and PSUNIX, old main storage is freed by "prfree," a new function. Because of the different method of controlling swap space, SUNIX and PSUNIX use "swfree" to free any swap space allocated to the process.

c. Returned Values

This system call does not return to its caller. The next function invoked for this process is "wait" which completes the cleanup.

d. Error conditions

This system call has no error conditions.

3. sbreak()

a. Parameters

The current process's UVECTOR and PROC are implied parameters. Because this function is a system call, an additional argument, the virtual address of the new end of the data, is found in the array u.u_arg[] in the UVECTOR.

b. Functional Description

This function is the system call used to change the size of the current process's data area. It calculates the new data size desired by the current process and validity checks the current process's total storage requirement. If the requirement is not valid it returns without fulfilling the requirement. In UNIX, "expand" is used to establish the new region. In PSUNIX, "dexpand" is used to change the size of the data. In SUNIX, this function attemots to do the work itself. It puts the new size in p->dsize in the current PROC. If the new size is smaller, it frees the excess storage. If the new size is

larger, it attempts to acquire it. If it fails, it calls "ceswap" to acquire the space by swapping. In all systems the newly acquired space is cleared.

c. Returned Values

Values returned by this system call are not checked.

d. Error Conditions

If the new storage requirement is not valid, this system call returns without allocating the storage. This will usually cause the process to terminate abnormally because of a memory management error.

G. text.c

1. xswap(p,ff,os), xswap(p,ff,ods,oss)

a. Parameters

In UNIX this function is called with three parameters. In SUNIX and PSUNIX, it is called with four parameters. The first parameter is a pointer to the PROC of a process to be swapped out of main memory. The second parameter is the memory free flag. In UNIX the third parameter is the process's region size in 64-byte blocks. In SUNIX and PSUNIX the third and fourth parameters are the process's data and stack sizes in 64-byte blocks.

b. Functional Description

In UNIX this function allocates swap space for the process and swaps it out. In SUNIX and PSUNIX this function allocates swap space only for those processes that do not already have it. In all versions of the operating system, memory is freed if the memory free flag is set. This flag will not be set if this function has been called by "newproc" to create a copy of a parent process.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

If swap space must be allocated but none is available, the message "out of swap space" will be sent to the console and the system will crash. If an output error occurs during the swap, the message "swap error" will be sent to the console and the system will crash.

2. xfree()

a. Parameters

The current process's UVECTOR and TEXT are implied parameters.

b. Functional Description

This function relinquishes use of the current process's shared text. If the current process has no shared text, this function just returns. If the current process has shared text, "xccdec" is called to decrement the TEXT's in-memory use count. The TEXT's active-process use count is then decremented. If this count has reached zero and if the text segment is not to be retained, the TEXT's swap space and the TEXT itself are freed.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

3. xalloc(ip)

a. Parameters

The parameter is a pointer to the inode of the text segment that is to be established or located. The current process's UVECTOR and PROC and all TEXTs are implied parameters.

b. Functional Description

This function establishes the shared text segment requested by the current process. If the current

process does not require shared text, this function returns. If the process does require shared text, this function searches the array of TEXTs for a previously established TEXT for the requested segment. If one is found its active-process use count is incremented. If the requested segment is in main memory, the TEXT's in-memory use count is also incremented and the function returns. If a TEXT has not been previously established, an unallocated TEXT is located and allocated to the text segment. Swap space is allocated for the text segment. The current process's address space is increased using "expand" to get space into which the text segment can be read. The text segment is read into memory and then it is written out to the swap space acquired for it. The memory acquired for the text read is freed using "expand" in UNIX and "prfree" in SUNIX and PSUNIX. The address of the TEXT is placed in p←textp in the current PROC. The current process is swapped out with "xswap". When it returns to memory, the text segment will return with it.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

If a TEXT must be allocated and one is not available, the message "out of text" will be sent to the console and the system will crash. If swap space must be

allocated and none is available, the message "out of swap space" will be sent to the console and the system will crash.

4. xccdec(xp)

a. Parameters

The parameter is a pointer to a PROC. The PROC's TEXT is an implied argument.

b. Functional Description

If the PROC has no TEXT this function returns. If it has a TEXT, the TEXT's in-memory use count is decremented. If the count reaches zero, the memory occupied by the TEXT's text segment is freed.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

H. page.c

1. ptbuild(tab,size1,size2,ad)

a. Parameters

The first parameter is a pointer to a PAGET. The next two parameters are the sizes in 64-byte blocks of a process's data and stack. The last parameter is a pointer to an array containing the memory physical block numbers of a process's pages.

b. Functional Description

This function is present only in PSUNIX. It puts the sizes and addresses of a process's pages into the PAGET. The order within the PAGET is data pages first from low to high virtual address followed by stack pages from high to low virtual address. Unused PAGET entries are zeroed.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

2. ptsize(tab,size1,size2)

a. Parameters

The first parameter is a pointer to a PAGET. The last two parameters are the sizes in 64-byte blocks of a process's data and stack.

b. Functional Description

This function is present only in PSUNIX. It puts page sizes into the PAGET. The order of sizes within the PAGET is data pages from low to high virtual address followed by stack pages from high to low virtual address.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

3. palloc(ptab,ds,ss,cont)

a. Parameters

The first parameter is a pointer to a PAGET. The next two parameters are a process's data segment and stack segment sizes in 64-byte blocks. The last parameter is the contiguity flag.

b. Functional Description

This function is present only in PSUNIX. It calls "ptsize" to put the page sizes into the PAGET, allocates main memory for each PAGET page with a nonzero size, and puts the starting block numbers of the allocated memory into the PAGET. If allocation fails for any page, all memory allocated for the process is freed. If the

contiguity flag is set, contiguous memory is allocated for both the data and stack pages.

c. Returned Values

If allocation for all pages is successful, minus one is returned. If allocation fails for any page, zero is returned.

d. Error Conditions

A returned value of zero indicates an allocation error to the caller.

4. pfree(tab)

a. Parameters

The parameter is a pointer to a PAGET.

b. Functional Description

This function is found only is PSUNIX. It frees the memory allocated to the pages in the PAGET.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

5. pcopy(otab,ntab)

a. Parameters

The first parameter is a pointer to the origin PAGET. The second parameter is a pointer to the destination PAGET.

b. Functional Description

This function is present only in PSUNIX. It copies pages pointed to by the origin PAGET to corresponding pages pointed to by the destination PAGET. A zero page block number in either PAGET terminates the copying. The number of blocks copied per page is determined by the size of the origin page.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

6. pralloc(pr)

a. Parameters

The parameter is a pointer to a PROC. The PROC's TEXT is an implied input.

b. Functional Description

This function is present only in SUNIX and PSUNIX. It acquires memory for the process's UVECTOR, data segment, stack segment, and, if necessary, shared text segment. Space for the text segment is acquired only if the text is not main memory resident. If any allocation fails, all memory previously allocated is freed.

c. Returned Values

If all allocations are successful, the memory block number of the first block allocated for the UVECTOR is returned. If any allocation fails, zero is returned.

d. Error Conditions

A return value of zero indicates an error to the caller.

7. pswap(daddr,tab,num,flag)

a. Parameters

The first parameter is a block number on the swap device. The second parameter is a pointer to a PAGET. The third parameter is the number of pages to swap. The last parameter is the read-write flag.

b. Functional Description

This function is present only in PSUNIX. It swaps the indicated number of pages to or from main memory.

If the read-write flag is set, the pages are swapped into the memory locations specified by the PAGET. If it is not set, the pages are swapped to the swap device beginning at the specified block number.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

8. prswap(rp)

a. Parameters

The parameter is a pointer to a PROC. The PROC's TEXT is an implied input.

b. Functional Description

This function is present only in SUNIX and PSUNIX. It swaps a process's UVECTOR, data segment, stack segment, and, if necessary, text segment into main memory.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

If an input error occurs during the swap, the message "swap error" will be sent to the console and the system will crash.

I. bio.c

1. physio(strat, abp, dev, rw)

a. Parameters

The first parameter is a pointer to the I/O initiation routine routine for the device to be used. The second parameter is to a buffer header that contains control information about the I/O operation to be performed. The third parameter is the device identifier. The fourth parameter is a read/write flag.

b. Functional Description

This function computes and validates the physical address of an I/O area within the User address space. If the address is valid, this function calls the I/O initiation routine to start the I/O operation. In PSUNIX, this function determines if the I/O area crosses a page boundary. If it does cross a boundary, this function calls "contig()" to make the requesting process contiguous.

c. Returned Values

The values returned by this function are not checked.

d. Error Conditions

If an I/O error occurs or the requested operation is not valid, an error condition is signaled by setting a bit in u←uerror in the requesting process's UVECTOR.

APPENDIX C: SYSTEM BENCHMARKS

A. BENCH1

```
chdir /usr/emery
sh old
chdir ken
cc -O -c slp.c
cd ..
cd dmr
ed ipc.c </usr/bench/edcmd >/dev/null
chdir /usr/bench
cc -O rftest.c
bas tower<towerin>/dev/null
od  /usr/sys/conf/m45.s >/dev/null
cp /unix /dev/null
chdir /bin
sum *>/dev/null
wait
chdir /usr/emery/ken
rm slp.o
chdir /usr/bench
rm a.out
```

B. BENCH2

```
chdir /usr/emery
sh old&
chdir ken
cc -O -c slp.c&
cd ..
cd dmr
ed ipc.c </usr/bench/edcmd >/dev/null&
chdir /usr/bench
cc -O rftest.c&
bas tower<towerin>/dev/null&
od  /usr/sys/conf/m45.s >/dev/null&
cp /unix /dev/null&
chdir /bin
sum *>/dev/null&
wait
chdir /usr/emery/ken
rm slp.o
chdir /usr/bench
rm a.out
```

LIST OF REFERENCES

1. Allen, B. E. and Barksdale G. L., Design Considerations for the NPS Signal Processing and Display Laboratory Multiprocessing Operating System, Naval Postgraduate School, 1975.

2. Hawley, J. A. and Meyer W. B., MUNIX, A Multiprocessing Version of UNIX, M. S. Thesis, Naval Postgraduate School, 1975.

3. Marsh, M. T., Memory Management For Paged, Heirarchical Memory in A Multiprocessing Computer System, M. S. Thesis, Naval Postgraduate School, 1975.

4. Joy, R. E., Implementation of an Adaptive Scheduling Algorithm for the MUNIX Operating System, M. S. Thesis, Naval Postgraduate School, 1975.

5. Digital Equipment Corporation, PDP-11/45 Processor Handbook, 1975.

6. Knuth, D. E., The Art of Computer Programming, v. 1, Addison-Wesley, 1968.

7. Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," Communications of the ACM, v. 17, no. 7, p. 365-375, July, 1974.

8. Ritchie, D. M., C Reference Manual, Bell Telephone Laboratories, 1974.

9. Ritchie, D. M., UNIX Assembler Manual, Bell Telephone Laboratories, 1974.

10. Donovan, J. J. and Madnick, S. E., Operating Systems, McGraw-Hill Book Co., 1974.

11. Ritchie, D. M. and Thompson, K., UNIX Programmer's Manual, 6th ed., Bell Telephone Laboratories, 1975.

12. Ritchie, D. M., The UNIX I/O System, Bell Telephone Laboratories, 1974.

13. Denning, P. J., "Virtual Storage," Computing Surveys, v. 2, no. 3, p. 153-189, September, 1970.

14. Randall, B., "A Note on Storage Fragmentation and Program Segmentation," Communications of the ACM, v. 12, no. 7, p. 365-369, July, 1969.

15. Organick, E. I., The MULTICS System, M. I. T. Press, 1972.

16. Digital Equipment Corporation, PDP(11 Peripherals Handbook, 1975.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexanderia, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science. Naval Postgraduate School Monterey, California 93940	1
4.	Asst. Professor Gerald L. Barksdale, Jr. Code 52Ba Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	LTJG Gary M. Raetz, USN, Code 52Rr Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	Capt. Harvey W. Emery, Jr., USMC Code ISM, Headquarters, United States Marine Corps Washington, D. C. 20380	1

Thesis
E43624
c.1

165612

Emery
The development of a partitioned segmented memory manager for the UNIX operating system.