A DEMAND PAGED UNIX SYSTEM FOR THE HARRIS /6 MINICOMPUTER

by

WILLIAM ARTHUR SHANNON

Submitted in partial fulfillment of the requirements

for the Degree of Master of Science

Thesis Advisor: Charles W. Rose

Department of Computer Engineering and Science

CASE WESTERN RESERVE UNIVERSITY

January 7, 1981

CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

William Arthur Shannon

candidate for the M. S.

degree.

Signed: [illegible]
(Chairman)

[illegible]

[illegible]

Date 9-18-80

A DEMAND PAGED UNIX SYSTEM FOR
THE HARRIS /6 MINICOMPUTER

Abstract

by

WILLIAM ARTHUR SHANNON

The UNIX* operating system was ported from the DEC PDP-11 minicomputer to the Harris /6 minicomputer. The /6 is vastly different from the PDP-11, being word addressible with 24 bit words. The problems encountered while porting UNIX to the /6 are discussed along with implemented solutions and suggestions to simplify future porting efforts. In addition, UNIX was extended to support full demand paging using the /6 virtual memory hardware. The implementation of the resulting virtual memory UNIX system, called UNIX/24V, is described.

----

* UNIX is a trademark of Bell Laboratories

To my wife, Karen, for her encouragement and support.

ACKNOWLEDGEMENTS

I would like to thank Samuel Leffler, who implemented the C compiler for the /6. Without his work this project would not have been possible. He was also instrumental in debugging early versions of the system and provided valuable feedback on design decisions.

The efforts of Robert Gingell are greatly appreciated. He provided or helped find explanations for many of the /6 hardware problems and anomalies. In addition, his porting of the Version 7 Shell helped make UNIX/24V a real, usable UNIX system. He was also responsible for initiating this project within ARJCC.

I would like to thank my advisor, Professor Charles Rose, for his help in preparing this document, and Professors George Ernst and Raymond Hookway for serving on my committee.

Finally, I would like to thank the A. R. Jennings Computing Center for sponsoring this project and for providing manpower and equipment.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

CHAPTER I

INTRODUCTION

1.1. Background and Motivation

In April, 1976, the A. R. Jennings Computing Center of Case Western Reserve University purchased six Harris /6 computers. These computers were to be joined together by a DECnet-like network[21], to be developed and built jointly by Harris Computing Systems Division and Case Western Reserve University. The network was called CWRUnet and was to support both Harris and non-Harris computers.

Harris spent 3 years modifying Vulcan to add the facilities necessary for networking. CWRU spent the same 3 years modifying Vulcan to make it usable as a general timesharing system. After this amount of work, it became clear to CWRU that a major rewrite of Vulcan would be necessary before it could achieve the hoped for functionality in a network environment. Harris had no interest in performing this rewrite since many of the problems were avoided by using their new 500 and 800 series systems. However, CWRU was left with /6's which must be made usable. For several reasons (manpower, experience, etc.) it was not deemed feasible for CWRU (ARJCC in particular) to rewrite Vulcan or to write an entirely new operating system for the /6.

Inspired by other´s successes with UNIX [10] [13] [14], ARJCC decided to initiate a project to port Version 7 UNIX to the Harris /6. UNIX was to provide a base from which to build CWRUnos, CWRU´s network operating system. The UNIX project was divided between two graduate students - Sam Leffler, who wrote the C compiler for the /6 [12], and myself, who ported the UNIX operating system to the /6. Although the /6 is based on a very old architecture, a sophisticated paging system has been added to the basic design. The thesis of this work is that the UNIX operating system can be ported to the Harris /6, and that it can be further modified and extended to support demand paging.

1.2. UNIX

The UNIX operating system [20] is a general purpose timesharing system for medium scale minicomputers, typically with 128K - 2M bytes of memory, 5M - 100M bytes of disk storage, and 8 - 32 terminals. It was originally designed for the PDP-11 family of minicomputers by Ken Thompson and Dennis Ritchie of Bell Laboratories and is distributed by Western Electric. UNIX is written almost entirely in the C language [11] [19], a medium to high level systems programming language with much of the flavor of PL/I.

Being written in C has made it possible for UNIX to be ported to several other machines, including the DEC VAX-11/78Ø [13], Interdata 7/32 [14] and 8/32 [1Ø], Zilog Z8ØØØ, and Amdahl 47Ø. Work is also underway to port UNIX to many other machines such as the Motorola 68ØØØ and the BBN C machine. This work concerns the porting of UNIX to the Harris /6.

Most implementations of UNIX are swap-based systems, requiring a process to be entirely resident in memory to run, and removing a process entirely from memory when space is needed. There are a few exceptions. The Bell UNIX system for the VAX performs partial swaps, only removing part of a process from core if possible. However, the entire process must be resident to run. The Amdahl UNIX system is built on top of VM37Ø and so is, in some sense, a paged system, although UNIX itself does not perform any paging. The Berkeley UNIX system for the VAX is a true paging system. It is an extensively modified version of the Bell VAX system that fully supports the VAX paging hardware. The UNIX system for the Harris /6 (called UNIX/24V) described herein is also a true paging system. The /6 virtual memory hardware is fully utilized to provide a true demand paged UNIX system.

1.3. Hardware Configuration

UNIX/24V was first developed on a /6 with 192KW of memory, one model 5200 cartridge disk (10.8MB), one model 6600 magtape drive, a console interface connected to a DECwriter II, one DMA Communications Processor (DMACP, a 6800-based intelligent terminal multiplexer), and a model 4125 Dataproducts line printer. It was later moved to a system with 208KW of memory, one model 5550 high capacity storage module disk (300MB), one model 6600 magtape driver, a console interface with DECwriter II, four DMACP´s, and a MUX with 10 terminal ports. Only UBC channels were used for the disk and tape interfaces. The console interface was slightly modified to allow connection to a DECwriter instead of a TEC terminal, which Vulcan requires (see Appendix B).

1.4. Synopsis

Chapter II describes the /6 architecture - its registers, I/O and interrupt structure, etc. The use of /6 registers by the C compiler is also described.

Chapter III describes the UNIX virtual machine model - the virtual machine expected by the UNIX kernel itself. The UNIX view of processes, memory management, and processor modes is described.

Chapter IV describes how the UNIX virtual machine was mapped to the /6. The actual porting of UNIX is described, including the changes made to data structures and procedures.

Chapter V deals with the modifications made to UNIX to support demand paging. The data structures and algorithms introduced and changed will be described in detail.

Chapter VI contains conclusions, performance observations, and suggestions for further research.

CHAPTER II

THE /6 ARCHITECTURE

2.1. Overview

The /6 is a 24 bit per word, word addressable machine. It is based on the Datacraft 6024 architecture. The 6024 originally had a 15 bit address space and was later expanded to 16 bits and then 18 bits. Because of this, the addressing structure is neither clean nor consistent. The newer 500 and 800 series machines allow a 20 bit address space.

2.2. Register Set

The /6 contains 8 programmer-accessible registers and 2 pseudo registers. The registers are not general purpose; each register is intended for a different function. The following paragraphs describe the registers [6]. The use of the registers in code generated by the C compiler is also briefly described; for more information see [12].

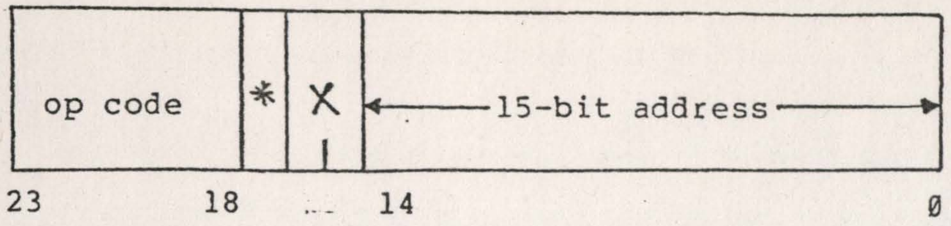
A Register The 24-bit A register is the main arithmetic register. It contains complete arithmetic and shift capabilities and is used as the input/output communication register. In C code this register is used

	as the general arithmetic and shift register.
E Register	The E register is a 24-bit general register. It is also used as an extension of the A register for additional shifting and arithmetic capability.
D Register	The D (Double) register is a 47-bit pseudo-register consisting of the A and E registers. It provides double-precision capability. The A register contains the 23 least significant bits and the E register contains the 24 most significant bits. In C code this register is used to manipulate long integers.
I, J, and K Registers	The I, J, and K registers are 24-bit index registers. The I register must hold the byte pointer when replacing bytes in memory. The J register must hold the byte pointer when extracting bytes from memory. The J register is also used for subroutine linkage. The K register is used for the frame pointer in C code.
B Register	The B (Byte) register is a pseudo-register consisting of bits Ø-7 of the A register. It provides byte manipulation capability.
C Register	The C (Condition) register is a 4-bit register that stores and displays the results of specific operations. The following conditions are indicated: Overflow, Negative, Zero, and Positive.
H Register	The H register is a single bit register associated with the Bit Processor feature.
V Register	The 18-bit V register is provided with the Bit Processor feature. This register is used to store a base address of an effective memory location containing the bit to be manipulated.

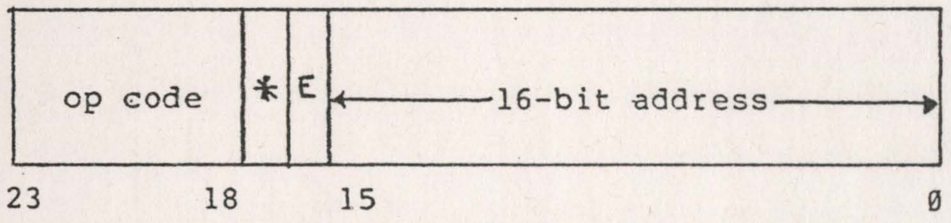
2.3. Instruction Set and Addressing

The /6 is essentially a one address machine. Each instruction can specify at most one memory address. Instructions exist to move data from register to register and between registers and memory. Although the format of instructions appears quite regular, not all instructions can be applied to all combinations of operands (see [12] for more details).

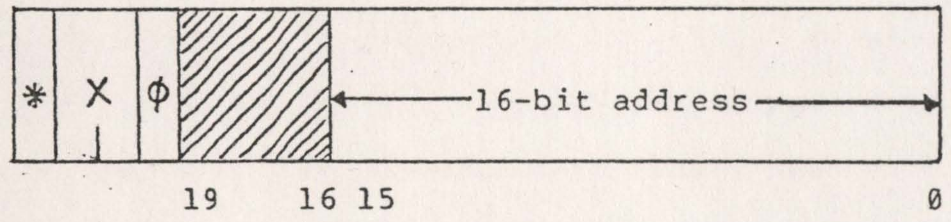
Most memory reference instructions are able to specify a 15-bit memory address, an optional index register for indexing, and optional indirection through the resulting effective address (Figure 2.1a). Indirection (optionally with indexing) can continue to any level. Since the address field of the instruction is only 15 bits wide, only 32K words can be directly addressed. However, the Program Counter is 16 bits wide. The first two 32KW sections of an address space are referred to as Map Ø and Map 1, determined by the high bit of the PC. All direct references using the instruction format described above can only address within the current 32KW map; that is, the high bit of the PC is or´ed into the effective address for all direct references. To access any address in the lower 64K words of memory, without regard to which map the program is executing in, one level of indirection must be used. The instruction specifies indirection through the effective address, which contains a Direct Address Constant (DAC) which can be used to address any word in



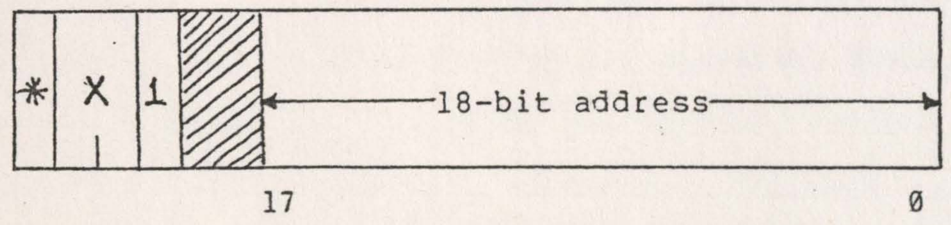
Direct Addressing



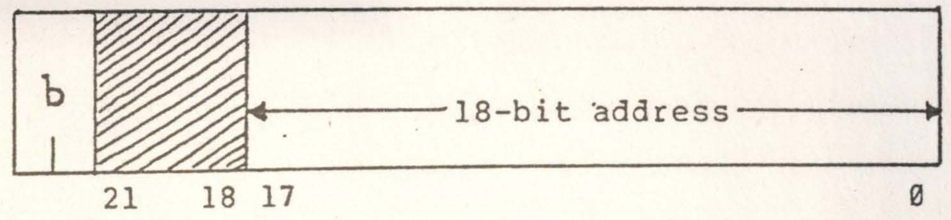
Long Branch Instructions



Standard Indirect Format (DAC)



Long Address Format (LAC)



Byte Address Constant (BAC)

Figure 2.1 - Instruction and Addressing Formats

the first 64K words of memory (Figure 2.1b).

Although up to 64KW may be used for instructions, the total address space of the /6 is 256KW. Only data may appear above 64KW. (Note that not even instructions executed by the execute memory (EXM) instruction may reside above 64KW. This has important implications later.) To address any word above 64KW a Long Address Constant (LAC) must be used (Figure 2.1c). Note that bit 20 must be be set to a one for the address constant to be recognized as a LAC. If bit 20 is not set, the effective address is truncated to 16 bits. This causes problems with pointers in C. It would be quite natural when referencing through a pointer to merely indirect through the location where the pointer is stored. This is not possible, however, since bit 20 will not be set in the pointer. If bit 20 were set in the pointer, address arithemtic would become quite complex, since bit 20 would have to be masked off. Again, see [12] for more details.

The addressing mechanism described above suffices to address any word-aligned data item. However, to address characters, a different address format is required (Figure 2.1d). The low 18 bits specify a word address and the high 2 bits specify a byte offset. The byte offset may have the value 1, 2, or 3 specifying the first, second, or third byte of the word. Bytes are numbered from the left. Special instructions exist to increment a byte pointer to point to the next byte, handling byte offset incrementing, and wraparound to the next word. See [12] for more discussion of the problems caused by this byte addressing technique.

2.4. I/O System

The following discussion of channels and interrupts greatly simplifies what is actually possible in the /6. Features that are not used by UNIX or that are best ignored are not discussed here. For complete details see the appropriate Harris documentation [6][7][8].

2.4.1. Channels and Units

I/O devices on the /6 are addressed by a channel number and a unit on that channel. Programmed I/O of words or bytes is possible to units that support this mode of data transfer -- most terminal and line printer interfaces. DMA transfers are controlled on a per channel basis on channels that are equipped for DMA transfers. The channel's responsibility is to control DMA transfers and to pass control and status information between the CPU and the individual device controllers attached to the channel. The channel has very little knowledge of the device controller attached to it, which is quite different from devices on the PDP-11. A PDP-11 device controller integrates the functions of DMA control and device control into one unit. Since these two functions are separated into two pieces of hardware on the /6, special programming considerations are necessary. For example, when the device indicates an error condition that prevents data transfer and the channel is still "conditioned" for the data transfer, the data transfer must be aborted in the channel before another command can be issued.

2.4.2. I/O Instructions

There are several instructions available for communicating with I/O devices. However, not all instructions are recognized by all channels and units. The channel number and unit number are coded as part of the instruction word. Therefore, each I/O instruction can apply to only one channel and unit. This makes it difficult to write a reentrant device driver for a group of devices all of which are the same except for their channel and unit numbers. All I/O instructions use the A register to transmit or receive data or status. The I/O instructions are described in the following paragraphs.

OCW	The Output Command Word instruction transfers a command word to the specified channel/unit combination. The command word specifies the operation the peripheral device is to perform. It may also be used to enable or disable device interrupts.
ISW	The Input Status Word instruction retrieves the current status of a device. The low 8 bits of the status are set by the device and the high 3 bits are set by the channel. The channel status bits indicate, among other things, whether or not DMA is active.
IDW	The Input Data Word instruction requests a specific channel/unit combination for a data word. The data word is often an 8 bit data byte from a character oriented peripheral device, such as a terminal. Some devices also use it to return an extended status word.
ODW	The Output Data Word instruction transfers a single word to the specified channel/unit combination. This is most often used to transfer single bytes to character oriented devices.
OAW	The Output Address Word instruction is used to initialize a DMA-type channel for DMA activity. The A register contains the address of a parameter block, which has the format shown in Figure 2.1. The first word of the parameter block contains the word count for the DMA transfer. The second word contains the address from/to which the data is to be transferred. For data chaining, the word count and buffer address are repeated as many times as necessary, with the chain bit set appropriately. For command chaining, the new command word precedes the word count for each chained operation. Note that the chain list must be contiguous in memory. After the channel is initialized with an OAW, an OCW is used to start the transfer.
IAW	The Input Address Word instruction returns the next address to be written to or read from for a DMA operation.
IPW	The Input Parameter Word instruction returns the address of the parameter block controlling the current DMA operation.

2.5. Priority Interrupt System

The /6 has two separate interrupt groups, Group Ø and Group 1. Group Ø is reserved for internal CPU functions and is composed of up to 8 executive trap levels. Group 1 is reserved for external interrupts and may have a maximum of 24 levels.

The 8 executive traps correspond to specific CPU features, many of which are optional, and are therefore permanently assigned to specific levels. Level Ø is the highest priority trap, and level 7 is the lowest. The executive traps and corresponding interrupt levels are as follows:

Ø	Power Down
1	Power Up
2	Program Restrict
3	Instruction Trap
4	Stall Alarm
5	Interval Timer
6	SAU Overflow/Underflow
7	Address Trap

All Group Ø traps are higher priority than Group 1 traps.

Each peripheral device is given one interrupt level in Group 1. Interrupt levels and signaling are completely separate from the device channel/unit addressing described above. Level 23 is the lowest level and level Ø is the highest level. When a particular interrupt level is active, all lower levels are prevented from interrupting.

Each interrupt level has associated with it a memory location that may be thought of as the interrupt vector location. The correspondence between interrupt group and level and memory address is shown in Figure 2.2. When an interrupt occurs, an Execute Memory (EXM) instruction with the appropriate memory address as operand is simulated. The instruction stored in the interrupt vector will almost always be a Branch and Save Long (BSL) instruction, which causes the PC and condition codes to be saved and the interrupt routine to be entered. The interrupt routine must save the registers. To return from the interrupt, a Branch and Reset interrupt Long (BRL) instruction is used. The BRL instruction loads the condition codes and PC that were saved (in the first location of the interrupt routine) by the BSL instruction.

Several registers are used to control the external device (Group 1) interrupt system (Figure 2.3). The first register is the Arm/Disarm (A/D) register. It is the "first level of defense" for incoming interrupts. If a particular interrupt level is not armed and that interrupt is triggered, it will

Address (octal)	Interrupt
60	power down
61	power up
62	virtual memory violation
63	instruction trap
64	stall alarm
65	interval timer
66	SAU overflow/underflow
67	address trap
70	Group 1 Level 0 - hard parity error
71	Group 1 Level 1 - soft parity error
72	Group 1 Level 2
73	Group 1 Level 3
74	Group 1 Level 4 - console terminal
75	Group 1 Level 5
76	Group 1 Level 6 - magtape
77	Group 1 Level 7 - disk
100	Group 1 Level 8
101	Group 1 Level 9 - DMACP #1
102	Group 1 Level 10 - DMACP #2
103	Group 1 Level 11 - DMACP #3
104	Group 1 Level 12
105	Group 1 Level 13
106	Group 1 Level 14
107	Group 1 Level 15
110	Group 1 Level 16
111	Group 1 Level 17
112	Group 1 Level 18
113	Group 1 Level 19
114	Group 1 Level 20
115	Group 1 Level 21 - line clock
116	Group 1 Level 22 - second clock
117	Group 1 Level 23 - reschedule

Figure 2.2 - Interrupt Addresses

be lost. Interrupt levels for devices that exist in the system are usually armed at initialization time and never disarmed.

The next register is the Request register. If an interrupt level is armed and the interrupt is triggered, the Request register will be set. The Request register may be set under program control to cause a software interrupt.

The Enable/Inhibit (E/I) register controls further processing of an interrupt request. If the interrupt request bit is set, the level is enabled, and no higher priority interrupt is active, the interrupt will be processed as described above. If the level is not enabled, the request will be saved and will cause the interrupt to become active if the level is later enabled, assuming no higher priority interrupt is active.

Once an interrupt request "passes through" the A/D and E/I registers, it may become active if no higher (or same) priority interrupt is active. When an interrupt becomes active, the Active register is set.

When an interrupt is active, the corresponding level may be disabled. This places the interrupt in a "permissive" state. When this occurs, lower priority

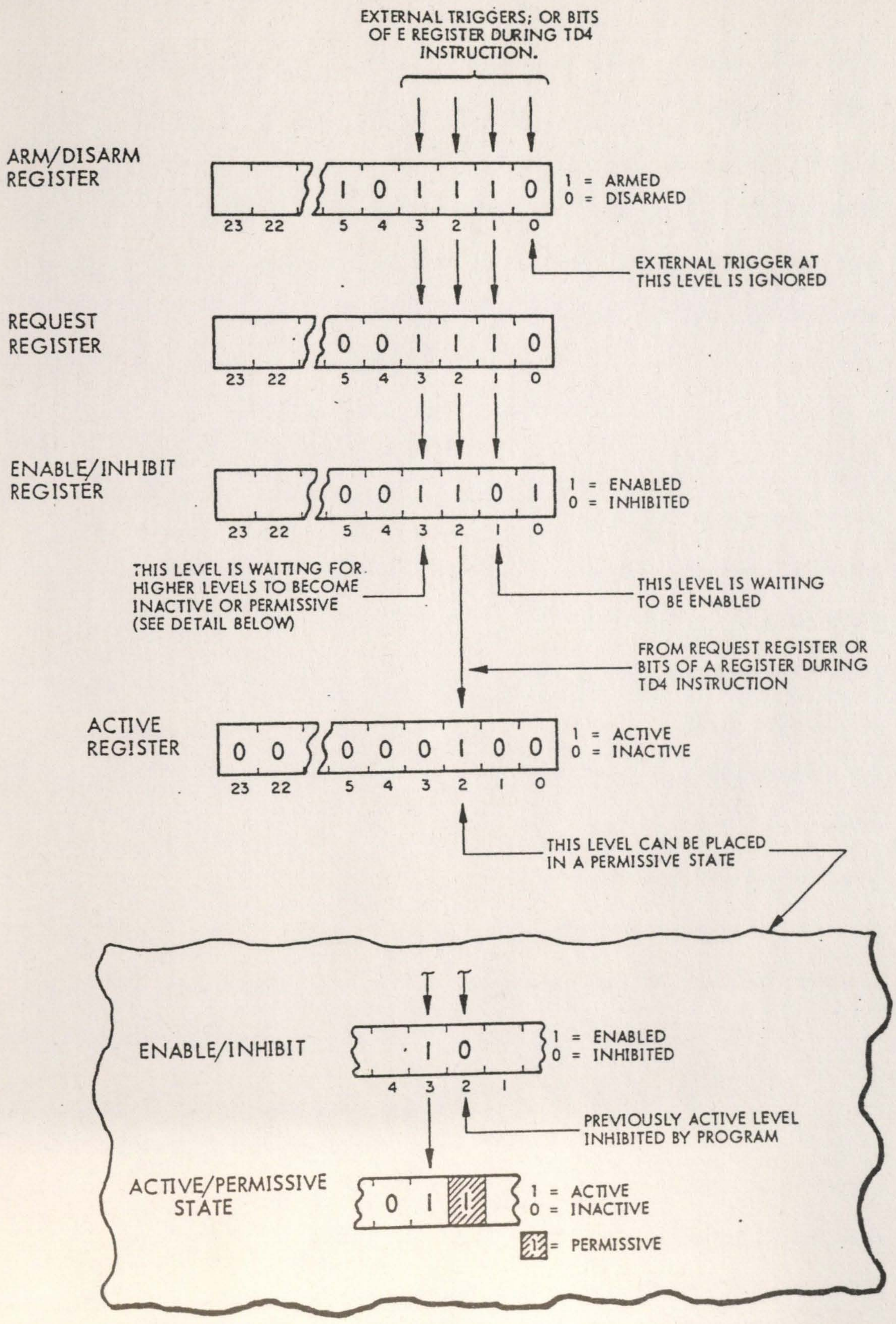

EXTERNAL TRIGGERS; OR BITS OF E REGISTER DURING TD4 INSTRUCTION.
ARM/DISARM REGISTER
1 0 1 1 1 0
23 22 5 4 3 2 1 0
1 = ARMED
0 = DISARMED
EXTERNAL TRIGGER AT THIS LEVEL IS IGNORED
REQUEST REGISTER
0 0 1 1 1 0
23 22 5 4 3 2 1 0
ENABLE/INHIBIT REGISTER
0 0 1 1 0 1
23 22 5 4 3 2 1 0
1 = ENABLED
0 = INHIBITED
THIS LEVEL IS WAITING FOR HIGHER LEVELS TO BECOME INACTIVE OR PERMISSIVE (SEE DETAIL BELOW)
THIS LEVEL IS WAITING TO BE ENABLED
FROM REQUEST REGISTER OR BITS OF A REGISTER DURING TD4 INSTRUCTION
ACTIVE REGISTER
0 0 0 0 0 1 0 0
23 22 5 4 3 2 1 0
1 = ACTIVE
0 = INACTIVE
THIS LEVEL CAN BE PLACED IN A PERMISSIVE STATE
ENABLE/INHIBIT
1 0
4 3 2 1
1 = ENABLED
0 = INHIBITED
PREVIOUSLY ACTIVE LEVEL INHIBITED BY PROGRAM
ACTIVE/PERMISSIVE STATE
0 1 1
1 = ACTIVE
0 = INACTIVE
▨ = PERMISSIVE

Figure 2.3 - Interrupt Registers

interrupts may interrupt without the need for the interrupt in the permissive state to be reset. The permissive interrupt may later be enabled, at which point it will become active again. The BRL instruction, which is used to return from an interrupt service routine, will reset the highest priority active (not permissive) interrupt.

2.6. Virtual Memory System

The /6 provides a virtual memory system that supports demand paging. The virtual memory system allows execution of programs that are only partially loaded into memory, thus allowing programs larger than the physical memory size to be run. Each process executing on the /6 is divided into a collection of 1024 word pages which need not be contiguous in main memory. Each process may be composed of no more than 256 pages. Pages may be protected with read only or read/write attributes, allowing pages to be shared between processes.

2.6.1. Modes of Operation

The /6 operates in two modes -- monitor and user. In monitor mode, the paging logic is disabled, and all addressing is to physical memory. In user mode, the paging logic is enabled, and all addresses are translated by the virtual memory system from logical addresses to physical addresses. The virtual memory system may only be manipulated when the CPU is in monitor mode. Any virtual memory system instructions executed in user mode are treated as illegal instructions.

2.6.2. Virtual Memory System Registers

The virtual memory system contains several registers that control its operation. These registers are described below.

VARs The virtual memory system contains 4096 Virtual Address Registers. The VARs map logical addresses to physical addresses. Each VAR is 10 bits, the bottom 8 bits contain a physical page number. The top 2 bits contain access mode information. Each process is allocated a contiguous set of VARs, defined by the Virtual Base Register and Virtual Limit Register for the process. The VARs thus defined correspond to the logical pages of the process. The VARs may be thought of as the page table for the process.

VBR The Virtual Base Register is a 12 bit register that contains the number of the first VAR assigned to the currently executing process. The VBR must be context-switched between processes.

VLR The Virtual Limit Register defines the last VAR used by the currently executing process. It is a 10 bit register. The bottom 8 bits contain the number of VARs used by the current process, minus 1. Bit 8 is the ROM suppression bit; see the discussion of the ROM instruction below. Bit 9, when set, allows the current process to execute a set of instructions that would otherwise be illegal. This set of instructions includes the I/O and priority interrupt instructions. The VLR must also be context-switched.

VUR There are 256 of these 1 bit Virtual Usage Registers. Each VUR corresponds to a physical memory page. Each time a physical memory page is accessed, the corresponding VUR is set.

VNR There are also 256 Virtual Not-modified Registers, each 1 bit. When a physical memory page is written in to, the corresponding VNR is set.

VUB The 8 bit Virtual Usage Base register controls access to the VURs and VNRs. The contents of the VUB define which VUR or VNR will be accessed by the Query virtual Usage Register (QUR) and Query virtual Not-modified Register (QNR) instructions.

VSR The 12 bit Virtual Source Register defines which VAR is read for the Transfer 2 virtual address Registers to Double (TRD) instruction.

VDR The 12 bit Virtual Destination Register serves a similar purpose as the VSR for the Transfer A to 1 virtual address Register (TAR) and Transfer Double to 2 virtual address Registers (TDR) instructions.

VPR The 12 bit Virtual demand Page Register provides information about page faults. The low 4 bits contains the reason for the page fault and the high 8 bits contain the logical page number in the current process that caused the page fault.

2.6.3. Basic Address Translation

In the /6 virtual memory system, physical and logical memory is divided into 1024-word pages. A mapping scheme is applied to all memory references. The VBR, VLR, and VARs work together to perform the mapping. Figure 2.4 illustrates the mapping technique. The high bits (bits 10 - 17) of a memory reference specify a logical page in the user´s address space. This page number is compared with the VLR to check for a page-out-of-range violation. The page number is

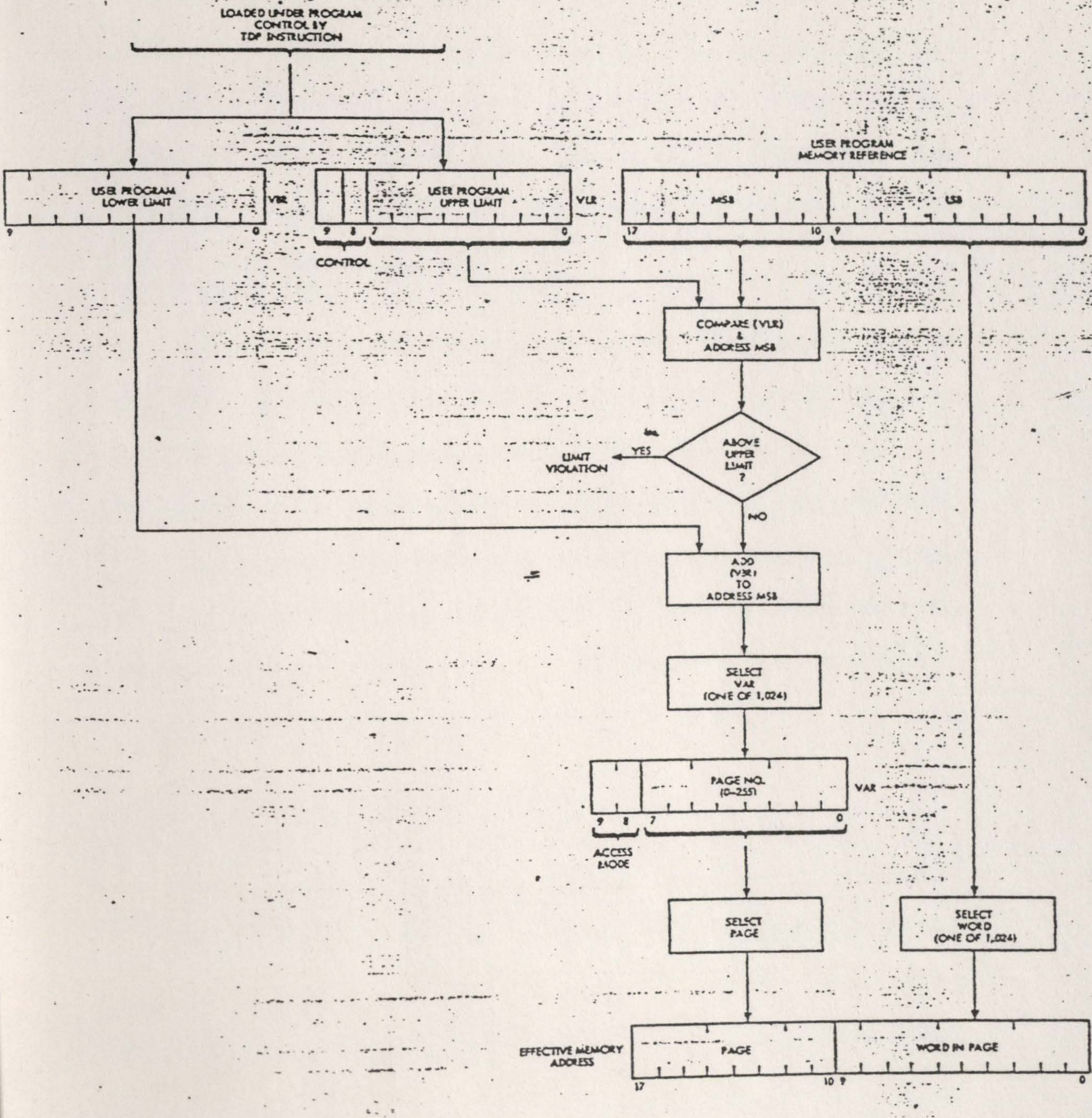


Figure 2–4. Basic Address Translation, Virtual Memory User Mode

added to the VBR to obtain the number of a VAR. The access mode bits of the VAR are compared with the intended access type (read/write) to determine if the access is allowed. A demand page violation (page not resident in memory) may be signalled at this point. Otherwise, the physical page number in the VAR replaces the logical page number in the memory reference and the instruction is performed.

It can be seen from the above discussion that each user process will need a contiguous set of VARs to map its pages. The particular set of VARs used is specified by a base-bounds mechanism, the VBR and VLR. Note that since the VARs are a limited resource, it may not be possible to allocate a set of VARs to each process all the time. This problem, and its solution, is discussed in more detail in Chapter IV.

2.6.4. Virtual Memory Violations

The virtual memory system recognizes several violation conditions and generates an interrupt (Group Ø, Level 2) so that the operating system may take corrective action. When this (or any) interrupt occurs, the CPU is placed in monitor mode before executing the hardware generated EXM through the interrupt vector. The VPR may be examined to determine the nature of the violation.

Four different virtual memory violations are possible. The first is a limit register violation. This occurs when an access is made to a page number greater than the value specified in the VLR. The second and third types are access violations. They occur when a write attempt is made to a non-writable page and when an execute attempt is made to a non-executable page. The last violation type is a demand page violation. A demand page violation occurs when the VAR for the accessed page specifies that the page is not resident in memory. In all cases, the user's PC can be backed up and the instruction retried after corrective action has been taken. The VPR specifies how much to back up the PC. The corrective action may consist of, for instance, loading a page into memory for a demand page violation, or making a page writable for a write violation. The only instruction that may not be recovered from properly after a page fault is a Transfer Memory to Registers (TMR) instruction that is indexed, since the index register may be loaded before the page fault occurs.

2.6.5. Virtual Memory Instructions

Several instructions are provided to manipulate the virtual memory system. Most of them transfer data to or from the A or D register or test the state of virtual memory system registers. These instructions will be described briefly below.

TDS	The Transfer Double to Source and destination registers instruction loads the VDR with the contents of the A register and the VSR with the contents of the E register.
TSD	The Transfer Source and destination registers to Double instruction loads the E register with the contents of the VSR and the A register with the contents of the VDR.
TAR	The Transfer A to 1 virtual address Register instruction loads the VAR specified by the VDR with the contents of the A register. The VDR is incremented by one.
TDR	The Transfer Double to 2 virtual address Registers instruction loads the VAR specified by the VDR with the contents of the E register. The VDR is then incremented by one. The VAR now specified by the VDR is loaded with the contents of the A register. The VDR is again incremented by one.
TRD	The Transfer 2 virtual address Registers to Double instruction loads the E register with the contents of the VAR specified by the VSR. The VSR is then incremented by one. The A register is then loaded with the contents of the VAR now specified by the VSR. The VSR is again incremented.
TDP	The Transfer Double to Paging limit registers instruction loads the VBR with the contents of the A register and the VLR with the contents of the E register.
TPD	The Transfer Paging limit registers to Double instruction loads the A register with the contents of the VLR and the E register with the contents of

the VBR.

TUD The Transfer Usage base register and demand page register to Double instruction loads the A register with the contents of the VPR and the E register with the contents of the VUB. Note that bits 4 - 11 of the VPR replace bits 16 - 23 of the A register and bits Ø - 3 of the VPR replace bits Ø - 3 of the A register.

TEU The Transfer E to virtual Usage base register instruction loads the VUB with the contents of the E register.

QUR The Query Usage Register instruction sets the condition register to "zero" or "not-zero" if the contents of the VUR specified by the VUB is Ø or 1, respectively. The specified VUR is cleared and the VUB is incremented by one.

QNR The Query Not-modified Register instruction sets the condition register to "zero" or "not-zero" if the contents of the VNR specified by the VUB is Ø or 1, respectively. The specified VNR is cleared and the VUB is incremented by one.

ROM The Release Operand Mode instruction will cause the effective memory address of the following instruction to be translated by the virtual memory system. The ROM instruction has no effect if the ROM suppression bit in the VLR is set.

RUM The Release User Mode instruction causes the CPU to enter user mode. The RUM does not take effect until a branch instruction is executed. Therefore, in practice, the instruction following the RUM should always be an unconditional branch.

CHAPTER III

THE UNIX VIRTUAL MACHINE

This chapter discusses the UNIX virtual machine model, that is, the type of machine that is well suited to the UNIX kernel. The type of machine facilities required for a reasonable implementation of UNIX will be described. Facilities required by almost all other multi-user operating systems (such as an interrupt driven I/O system, some sort of user/system protection, etc.) will not be discussed. The UNIX virtual machine, as seen from the user´s point of view, is described in [2Ø], [18], and [1Ø]. The kernel´s virtual machine, and how it is used to implement the user´s virtual machine, will be described here.

3.1. Processes

UNIX provides user programs with an address space consisting of up to three logical segments. The text segment starts at the beginning of the user´s virtual address space, contains only instructions and constant data, and is write protected. A single copy of the text segment is shared among all its users. The text segment may be empty if the user´s program is not reentrant. In this case, the instructions would reside in the data segment. The data segment starts at the next hardware protection boundary after the text segment, is writable, and is private to each process. It may be expanded by explicit system calls. The stack segment starts somewhere after the data segment and may grow either up or down, depending on what is most natural for the particular implementation. The stack segment grows automatically. On the PDP-11, VAX, and /6, the stack grows down. On the Interdata 8/32, the stack grows up.

Each process also has associated with it a system stack. The system stack is used when the system is executing procedures on the user´s behalf, such as servicing system calls. The system stack is located in the system´s per-process context block, generally called the `u vector´ or `user structure´.

The u vector contains any per-process information that is not needed when the process is swapped out of memory. Information about a process that must always be available is stored in the proc structure, which is always resident. The u vector contains the following types of information:

(i) State information, including saved registers, disposition of signals, label variables, etc. Does not include scheduling state information, such as process state (running, blocked, etc.) or priority.

(ii) Identity information, identifying the user that owns the process.

(iii) File information, describing open files and the current directory.

(iv) System call information, including arguments and return values.

(v) Mapping information, describing the size of each of the logical segments and how it should be arranged in memory.

(vi) Accounting information, such as execution times.

3.2. Memory Management

In most implementations of UNIX, including the PDP-11 and Interdata versions, the three logical segments described above are grouped into two segments for memory management purposes. The text segment is allocated contiguously in memory as one piece. It is also swapped as a single piece. The data and stack segments are allocated together in one piece of memory. They are packed as closely together as is possible, given the granularity of the memory management hardware. On the PDP-11, for instance, the data segment resides at the beginning of the allocated piece of memory, and the stack segment resides at the end. The physical hole between them will be smaller than the logical hole between them. Therefore, to expand the data segment or stack, a new piece of memory must be allocated, the data segment copied to the beginning of it, the stack segment copied to the end of it, and the new space in the middle added to the data or stack segment, depending on which one is expanding. Since there may not be enough contiguous memory (and since UNIX does no memory compaction), it may be necessary to swap the combined data and stack segment out to disk and bring it back in a larger memory area, when available. Also, UNIX makes no attempt to allocate incremental pieces at either end of the combined data and stack memory segment to minimize the amount of copying necessary.

Although the above scheme may require large amounts of data to be moved around, it greatly simplifies the problem of swapping segments in and out of memory, since all segments are contiguous in memory and on disk. Given more sophisticated memory management and I/O systems, many of the problems of the PDP-11 scheme may be avoided. In UNIX/32V on the VAX, memory is allocated a page at a time, and the pages need not be contiguous. All pages of the process must be resident in memory for the process to execute since UNIX/32V does not support demand paging. Swap space is allocated contiguously, but only a part of a process needs to be swapped out, if only a small amount of memory is needed. The partially swapped process may then be reloaded with much less I/O activity than if it had been fully swapped. In VM UNIX on the VAX, memory is allocated a page at a time and demand paging is supported. Swap space is allocated in multi-page chunks but the chunks need not be contiguous with each other. In UNIX/24V on the /6, both memory and swap space is allocated a page at a time and demand paging is supported so that the process need not be entirely resident in memory to execute.

As can be seen from the above discussion, a wide variety of memory management schemes are possible in UNIX. It is hard to state precisely the minimum requirements of the memory management hardware, but it seems that a single base and limit register (or even dual registers, if it is desirable to write-protect the text segment) are marginal, because of the difficulty of providing independently growable data and stack segments. Almost any sort of segmentation or paging system seems to be quite adequate.

3.3. Mapping

It should be obvious from the above discussion that some sort of mapping must be applied to all user memory references to provide the proper (safe) virtual machine for each user. However, because of the way the kernel does context switching, the kernel must also run in a mapped mode. The u vector is context switched by changing the kernel´s mapping registers so that a new u vector is mapped into the same address in the kernel´s logical address space. This method of context switching has several advantages. The u vector can be directly addressed at all times because it is at a fixed location in the address space, as opposed to being referenced indirectly through a pointer. Also, since the system stack is in the u vector, it is necessary that the u vector always have the same logical address. Otherwise, any addresses stored on the stack that point to other locations on the stack (most importantly, the stack frame linkage) would no longer point to the proper place.

It is most desirable to have at least two sets of mapping registers, one for the current user process and one for the kernel. One set of mapping registers can be managed, but they will have to be reloaded on every system call and interrupt. It is almost necessary that mapping not be associated with running in user mode, as demonstrated in the above discussion. On the /6, mapping and user mode are tied together, which caused several problems that are discussed in the next chapter.

3.4. I/O

UNIX does not require anything particularly unusual in terms of I/O capabilities. Priority interrupts, DMA, and programmed I/O are assumed to be available, but almost anything could be made to work. Although existing PDP-11 device drivers assume the PDP-11's form of memory-mapped I/O addressing, other I/O schemes can be made to work just as well if the proper primitives are chosen. The next chapter discusses this for the particular case of the /6.

CHAPTER IV

THE /6 UNIX SYSTEM

Now that the /6 architecture and the UNIX virtual machine have been described independently, this chapter will show how the UNIX virtual machine was modelled with the /6. In many respects, this was a process of making the /6 look like a PDP-11. There were also cases where UNIX had to be changed because there was no convenient way to model some constructs on the /6. The logistics of the porting process will first be described, followed by a description of the implementation of the UNIX virtual machine. The solutions to problems alluded to in previous chapters will also be described. Finally, the swap-based /6 UNIX system will be described. The paged /6 UNIX system is described in the next chapter.

4.1. The Port

The hardware to which UNIX was ported was described in Chapter I. This hardware was reserved for UNIX development. Additionally, there was a large /6 system running Vulcan, the standard Harris operating system, that was also available for the UNIX porting project.

Before any attempt was made to move pieces of UNIX to the /6, some experimentation was done with small standalone programs. Most of this early work depended heavily on having workable facilities on Vulcan. An old C compiler, developed by the University of Wisconsin, was used to experiment with standalone C code on the /6. A set of standalone startup and bootstrap programs were written that allow programs compiled by the Wisconsin C compiler to run on a bare machine without Vulcan. These standalone programs were written to magtape in a form suitable for the /6 ROM bootstrap, so that they could be booted on a bare machine.

By using these standalone facilities, it was possible to experiment with I/O devices and gain some insight into their proper operation. Most of the Harris manuals were either vague, ambiguous, or conflicted about how to program most of the I/O devices. Simple standalone programs were constructed to "experiment" with the I/O devices and determine the proper way to program them.

The development of the /6 UNIX system was intimately tied to the development of the /6 C compiler by S. J. Leffler [12]. The C compiler started as a cross compiler on the PDP-11/45, producing symbolic assembler code. As modules of the UNIX kernel were rewritten for the /6, they were compiled on the PDP-11 to check them for compile-time errors. The diagnostics of the portable compiler were very helpful here.

When enough of the kernel modules were converted, the assembler code was moved to Vulcan, assembled, and linked. The standalone modules written for the Wisconsin C compiler were rewritten to be compatible with the linkage conventions of the portable C compiler. This method allowed many of the modules of the UNIX kernel to be tested and debugged.

At this point it was necessary to have a UNIX file system on the disk so that the file system modules of UNIX could be debugged. The make-file-system program (mkfs) was converted to run on Vulcan under the Wisconsin C compiler. File systems could be built in an `unblocked´ file on Vulcan. The file was then copied to a disk pack by a privileged Vulcan program. The disk pack was then moved to the UNIX development machine for further testing.

Soon after it was possible to build file systems, the portable C compiler was ported to Vulcan. Programs could then be compiled and run on Vulcan; cross compiling was no longer necessary. A library of UNIX system call interfaces was written and small test programs using UNIX system calls were built. These test programs were put on the UNIX file system by mkfs (under Vulcan) in place of the /etc/init program. The /etc/init program is the first program run when UNIX is started. This method allowed most of the UNIX system call and file system code to be debugged.

In parallel with the previous task, many of the UNIX user programs were being converted to the /6. A simple `init´ program and shell were created in order to provide an environment in which to test many of the other UNIX user programs. The real init program was converted and installed once the necessary system calls were tested and working. This process prompted several redesigns of some of the system call interfaces, in particular the signal system call interface.

Several ARJCC staff members helped port many of the UNIX user programs, such as the editor and the Bourne Shell. It is the large number of small UNIX tools that make UNIX such a usable system, and the large number of programs converted at this point helped make the /6 UNIX system a real UNIX system.

Once the UNIX environment was seen to be reasonably stable and functional, Work was begun on many of the programs necessary to support C programming on the /6 UNIX system. These programs included an assembler, a loader, the make program, the yacc compiler-compiler, and the Standard I/O Library that is used by almost all UNIX programs. When these programs were operational on UNIX, it was possible to move the C compiler itself from Vulcan to UNIX. This provided a fairly complete program development environment on UNIX; the most important item missing was a debugger. Several additions to the C compiler (described in [12]) provided a program tracing facility that helped the debugging effort.

With the C compiler resident on UNIX, the only thing remaining to be done was to move the kernel code to UNIX and build a new UNIX system on UNIX itself. In the process of doing this, several assembler bugs were discovered and quickly fixed. It then became possible to build new UNIX systems and new C compilers on UNIX, no longer necessitating the development of UNIX programs on Vulcan. At this point UNIX was self-supporting.

The UNIX system developed by the preceeding method was a swap-based system. It used the /6 virtual memory hardware to simulate the PDP-11 memory management scheme. The next phase of development consisted of two tasks performed in parallel. The first task was to write and test device drivers for many of the devices on the /6. The initial UNIX system contained drivers for only the disk, magtape, and console terminal. Drivers for the DMACP terminal multiplexer, Dataproducts line printer, and 300MB disk were written and tested. The second task was the design of the paging system, which is discussed in Chapter V.

4.2. Data Structures

Due to the word size of the /6, many of the fundamental data structure of UNIX had to be changed. Although the sizes of most of the data structures are well-parameterized in the UNIX kernel, other implementations of UNIX have tended to keep them the same as the PDP-11 version of UNIX. This was usually possible because the word size of the other machines was generally a multiple of 2 bytes. The /6 word size is 3 bytes, which is incompatible with the PDP-11 sized data structures.

Although some of the data structures could have remained the same size, some inefficiency or waste would have resulted. Alignment problems were particularly important in data structures that are stored on disk. In general, most data structures were extended so that they were larger than the corresponding PDP-11 data structure. This tended to reduce the number of portability problems, at least in porting from the PDP-11 to the /6. Any program that takes advantage of the larger /6 data structures could be difficult to port back to the PDP-11 or other UNIX systems.

The data structures of concern here are those that reside on disk. Data structures residing only in the kernel were easily changed, either by changing a defined constant specifying the size of the data structure, or by changing the type definitions of the elements that make up the data structure. These changes can be easily seen by reading the system source code. The other data structures are described here.

4.2.1. Disk Blocks

The first and most important "fundamental constant" was the size of a disk block. On the PDP-11 and VAX, a disk block is 512 bytes, which is the size directly supported by the hardware. The disk hardware on the /6 supports a disk block (sector) size of 112 words (336 bytes). There were two concerns with this size of disk block. First, when storing a page (1Ø24 words) on disk, some space is wasted. Looking forward to the paging system, it was thought to be desirable to align pages of executable programs on disk sector boundaries to simplify moving them between memory and disk. The second concern was that many of the data structures that are stored in disk blocks (such as directories and inodes) assume that a power-of-2 of them will fill a disk block. It was felt that changing this would have far-reaching implications.

To resolve the above problems, it was decided to use a pseudo disk block size of 512 words. This would allow a page to be stored in two disk blocks. Data structures would also fit easily. The size of 512 came about as a compromise of several considerations. The pseudo disk block was to be implemented by using several contiguous disk sectors, with each new block starting at the beginning of the next sector. Therefore, the amount of waste at the end of the last sector of a block was important. Either 512 or 1Ø24 word blocks offered an acceptably small waste. The amount of waste is as follows:

block	sectors	waste
512	5	48 words, 8.6%
1024	10	96 words, 8.6%

The percentage of waste is the same in either case. Blocks of 1024 words would be particularly convenient when manipulating pages in the paged version of the system, as well as when swapping programs in the swapped version. However, 1024 words was felt to be much too large; it is six·times the size of the PDP-11 blocks. This would reduce the number of blocks that could be kept in the UNIX kernel's buffer cache, thus reducing the effectiveness of the cache. 512 word blocks seemed to offer a reasonable compromise among size, waste, and convenience.

4.2.2. Inodes

An inode describes a file; its owner, location on disk, etc. The locations of disk blocks that comprise the file are specified by a list of disk addresses. All but the last three of these addresses are disk addresses of blocks that contain the contents of the file. The last three disk addresses are pointers to blocks containing more pointers to blocks that actually contain the file information. The three pointers specify single, double, and triple indirect blocks. In the PDP-11 and VAX versions of UNIX, this list of disk addresses is a list of 13 24-bit addresses packed into 4Ø bytes. Since neither the PDP-11 nor the VAX can directly operate on 24-bit data items, a packing and unpacking operation is required when inodes are moved between memory and disk. The total size of a PDP-11 inode is 64 bytes, allowing 8 inodes to fit in a single disk block.

On the /6, the list of disk addresses is a simple (unpacked) array of 2Ø pointers. The total size of an inode is 32 words, allowing 16 inodes to fit in a single disk block. The number of disk address pointers was adjusted to make the total size of an inode a power of 2; the size of the other information in the inode was fixed and could not be adjusted. The structure of a /6 inode is as follows:

```
struct dinode
{
    unsigned short di_mode;   /* mode and type of file */
    short          di_link;  /* number of links to file */
    short          di_uid;   /* owner's user id */
    short          di_gid;   /* owner's group id */
    off_t          di_size;  /* number of bytes in file */
    daddr_t        di_addr[2Ø];  /* disk block addresses */
    time_t         di_atime; /* time last accesses */
    time_t         di_mtime; /* time last modified */
    time_t         di_ctime; /* time created */
};
```

The list of 2Ø disk addresses allows 17 direct, 1 indirect, 1 double indirect, and 1 triple indirect pointer. This allows 17 + 512 + 512*512 + 512*512*512 = 134480401 blocks (206561895936 bytes, approximately 200GB) to be addressed. However, since the size of the file is stored in a variable of type `off_t´, which is defined to be a long integer, a file as large as 140737488355328 bytes (approximately 140000GB) can be specified. The actual maximum file size allowed is the smaller of these two, about 200GB. Even this is larger than the maximum size of a UNIX file system volume, 25769803776 bytes (approximately 25GB), which is the real limit because disk addresses are stored in a 24-bit word.

4.2.3. The Super-Block

The super-block is the second disk block on a file system volume, the first being reserved for bootstraps. It specifies the size of the disk, the size of the list of inodes (the i-list), and serves as the head of the free list of unused disk blocks. There are two arrays of information in the super-block. The first is an array of free blocks, the head of the free list. The second is an array of i-numbers (indexes into the i-list) of inodes that are unallocated. This is used only as a cache of unallocated i-numbers; the real allocation information is a part of each inode. All other parts of the super-block are essentially fixed in size. The sizes of these two arrays were increased so that the super-block would completely fill a disk block. The sizes are as follows:

array	parameter	PDP-11 size	/6 size
free list	NICFREE	50	250
i-numbers	NICINOD	100	246

The sizes of these arrays were chosen somewhat arbitrarily. Some instrumentation could be performed to determine if these relative sizes are appropriate.

4.2.4. Directories

A directory entry has the following structure:

```
struct direct
{
    ino_t d_ino;              /* the i-number */
    char  d_name[DIRSIZ];     /* the file name */
};
```

On the PDP-11 (and VAX), DIRSIZ is 14, making the total size of a directory entry 16 bytes. 32 directory entries would fit in a single disk block. On the /6, the structure is the same and DIRSIZ is 21. The total size of a directory entry is 8 words, allowing 64 directory entries to fit in a single disk block.

The file name size of 21 characters was chosen so that it was at least as long as the PDP-11 size (to ease porting of user programs), and so that a power-of-2 of directory entries would fit in a disk block. The only problem here is that if any /6 UNIX programs were to depend on file names being 21 characters long (or anything more than 14 characters), portability back to other UNIX systems could prove difficult. This is a problem for the author of /6 UNIX programs; if he is never told of the longer file names, there should be no problem.

4.2.5. Other Data Structures

Few other data structures caused problems as difficult as the ones described in the previous section. One other data structure that did cause quite a few problems was the clist. The clist (character-list) is a linked list of structures containing characters that are input from or output to tty devices (other uses are possible; this is the most common one). While the size of clist elements did not cause any great problem, the addressing of them did. The machine-independent algorithms for manipulating the clist were machine-independent only on machines that were byte addressable with a power-of-2 bytes per word. The clist routines assumed that addresses of clist elements could be shifted and masked in certain ways, which did not work at all on the /6. These routines were rewritten to be more machine-independent.

4.3. Kernel Mode

As described in Chapter II, the /6 operates in two modes, monitor mode and user mode. The virtual memory hardware is enabled in user mode and disabled in monitor mode. As was seen in Chapter III, the UNIX kernel must operate with mapping enabled. To overcome this problem, kernel mode was defined. Kernel mode is a variant of user mode, the difference being that the privileged instruction bit in the VLR is set in kernel mode and reset in user mode. Although kernel mode is a mapped mode, the VARs for kernel mode are set up to map logical addresses to the same physical addresses. Mapping is enabled, but no "mapping" is actually performed except for the mapping of the u vector. All UNIX kernel software determines the state of the machine (kernel or user) by the state of the privileged instruction bit in the VLR. The state of the virtual memory system (enabled or disabled) is not considered. Note that the combination of the virtual memory state and privileged instruction bit defines four `modes´, only three of which are ever used. The privileged instruction bit is never reset when the CPU is in monitor mode.

Although almost all of the kernel can run in kernel mode, not all operations can be performed in this mode. Since interrupts put the CPU in monitor mode, kernel mode must be established before the interrupt routine is called. Also, the return from the interrupt routine can only be done from monitor mode. In addition, it is not possible to execute any of the virtual memory instructions in kernel mode, since they would cause an illegal instruction trap. Therefore, to perform any operation that must use virtual memory instructions, a special machine language routine is used. Each such machine language routine first enters monitor mode using the method described later. The routine then performs the needed function and reenters kernel mode.

This switching between monitor and kernel mode is not straight-forward. To switch from monitor to kernel mode it is first necessary to load the VBR and VLR with values that map the kernel address space. The RUM instruction is then used to enter mapped kernel mode. To switch from kernel mode to monitor mode is somewhat more complex. The /6 has only two mechanisms that cause it to enter monitor mode. The first is the BLU instruction, which is the standard system call instruction used in Vulcan. As described in the section on system calls (Section 4.4), BLU instructions were reserved. The second method is an interrupt. The easiest interrupt to generate from software is an illegal instruction interrupt. Therefore, the method to enter monitor mode is as follows. First, the global variable `nofault´ is loaded with the address where it is desired to enter monitor mode, usually a few instructions ahead. Second, an illegal instruction is executed, usually a TDP instruction. The illegal instruction interrupt handler checks the value of nofault. If non-zero, it returns there in monitor mode. Otherwise, a real illegal instruction has occurred and normal interrupt processing continues. This mechanism is used extensively (and only) in the machine language assist.

The following code fragment is typical of the method used to switch between kernel and monitor mode.

```
_func:
        bli csv
        tmi !2,k        / save argument in register
        tma _nofault    / save old value
        tam *!stack     /   on stack
        toa 1f          / where to return to
        tam _nofault    /   in monitor mode
        tpd             / cause illegal inst trap
1:
        ...
        < perform fuction here >
        ...
        rum             / reenter kernel mode
        buc .+1         /   at next instruction
        tme *!stack     / restore old value
        tem _nofault    /   of nofault
        bul cret        / return
```

Although this scheme works and is, in fact, necessary, it may generate a somewhat significant amount of overhead when switching between kernel and monitor mode. Measurement of the number and frequency of these switches would be interesting, but has not been attempted.

4.4. System Calls

System calls are the means by which user programs may request the system to perform tasks on their behalf. System calls in the /6 UNIX system are implemented as an illegal instruction. A Transfer Double to Limit register (TDL) instruction was chosen as the system call instruction because it is illegal in both user and monitor mode, and the low order bits of the instruction could be used to encode the system call number. This instruction is available as the `sys´ opcode in the UNIX assembler, as.

The /6 has an instruction, the Branch and Link Unrestricted (BLU) instruction, that is used by Vulcan as the system call instruction. It enters monitor mode and branches to a specified location in low core, saving the return address in the J register. There were two reasons for not using this as the UNIX system call instruction. First, it destroys the J register, which would have to be saved before every system call, thus complicating the system call interface code. Second, it was felt to be desirable to be able to trap these calls from a user program so that Vulcan programs could be simulated. Therefore, BLU instructions are not used as system call instructions; instead they generate a signal to the offending process.

When the user executes the sys instruction, an illegal instruction trap is generated, and the kernel trap routine is called. The trap routine determines what instruction caused the trap. If it was a sys (TDL) instruction, arguments are collected and the appropriate system call routine is called. Otherwise, an illegal instruction signal is sent to the process.

Arguments to system calls are passed via a parameter block. The address of the parameter block is loaded into the I register before executing the sys instruction. This is completely different from the method used on the PDP-11. Some arguments to some system calls are loaded in registers, while others are stored in-line, after the sys instruction. Therefore, to keep the text segment pure, it is often necessary to store the system call and arguments in the data segment and perform an indirect system call. Also, the arguments that were pushed on the stack before calling the system call interface must be moved to registers or to the parameter block after the sys instruction. On the /6, it is almost never necessary to do an indirect system call, and parameters rarely need to be moved around. In nearly every case, the contents of the parameter block correspond exactly to the arguments of the C-callable system call procedures described in section 2 of the UNIX Programmer´s Manual [22]. This makes most of the system call interface routines (that allow C programs to execute system calls) to be very simple. The prototypical system call interface routine is as follows:

```
syscall = number
_syscall:
        tmi !stack    / get pointer to param block
        sys syscall   / do the system call
        bno .+2       / skip if no error
        bul cerror    /   else return error
        bjl Ø         / return, result in A or D
```

This sequence of instructions is probably sufficient to implement 95% of all system call interfaces. One notable exception is the signal system call.

In order to properly interface with C functions that are to be used to trap signals, a small amount of machine language code is needed. Therefore, instead of trapping directly to the C function, the signal system call interface arranges that it be called instead of the C function. The C function that must be called eventually is stored in a table. The small interface routine that is called when the trap occurs dispatches through this table to call the C function. On return, it executes the return-from-signal system call, described in the next section.

4.5. Signals

The implementation of signals in the /6 UNIX system differs significantly from the PDP-11 implementation. The difference is primarily in the signal trapping mechanism. The PDP-11 supports a stack directly in hardware. The natural form of interrupt routine entry and exit uses the stack. On the /6, the stack is simulated; there is no direct hardware support. Interrupt routine entry and exit is inherently non-stack oriented. It is not desirable to model software interrupts (signals) after the hardware interrupt mechanism of the /6, since recursive interrupts are not allowed by this method.

Signal trap routine entry and exit is handled extensively by the kernel. Before calling a signal trap routine, the kernel pushes the current state of the user process on the user's stack. This state includes all the registers, the Program Counter, and the condition codes. This implies that the kernel has access to the user's stack pointer. As described in [12], the user's stack pointer is implemented as a regular memory location. Since this memory location changes with each user program, the kernel must know the address of the stack pointer for each user. This is accomplished by forcing the user program to tell the kernel where its stack pointer is. For C programs, the standard C startup code issues a special signal system call that contains the address of the stack pointer as a parameter. Another method considered was to have the loader put the address of the stack pointer in the a.out header of the executable program. This was rejected because it is not necessary for all programs to have a stack. For instance, non-C programs may choose different linkage conventions and may not need a stack.

To return from an interrupt routine, the state pushed on the stack by the kernel must be popped off and restored. A new system call was added to do this -- the return-from-signal system call. This system call will restore the user´s registers from the values stored on the stack. The PC and condition codes are also restored from the stack. Any of the values stored on the stack may have been changed by the signal trap routine. In particular, if the PC is changed, the program will return to a different point.

4.6. VAR Allocation

Each user process on /6 UNIX must have a contiguous set of VARs to map its pages. To map the maximum size process, 256 VARs are necessary. These must be allocated out of the pool of 4Ø96 VARs available on the machine. The first 256 VARs are permanently allocated to the kernel, leaving enough to map 15 maximum size processes. It was not felt that it would be sufficient to map only 15 processes at once. The overhead of loading VARs is considerable and should be avoided if at all possible.

The first step towards a solution of this problem involved adding some complexity to the allocation scheme. Since very few processes really need 256 VARs, it is preferable to allocate each process only as many as it would need. This implies that the stack could not start at the same fixed place (the end of memory) for every process. The loader was modified to select a variable stack size based on the size of the program being linked. The stack size is placed in the previously unused word of the a.out header. The kernel uses this information to decide how many VARs to allocate for the program. It is possible to override the default stack size when linking the program if it is known that the program will need an exceptionally large or small stack.

The second optimization was to allocate VARs only for processes that were loaded in memory. When a process is swapped out, its VARs are freed. Before swapping the process back in, a new set of VARs is allocated for it. This reduced the number of VARs used by processes that could not run because they were swapped out.

As mentioned above, the overhead of loading the VARs for a process everytime it is run is significant. Therefore, it was desirable to minimize the number of times the VARs must be reloaded. In the PDP-11 UNIX system, prototypes of the memory management registers for the process are stored in the process′ u vector. Every time the process is run, these prototype memory management registers are biased to reflect the process′ location in memory and loaded into the real memory management registers. In the /6 UNIX system, it was not advantageous to keep prototype VARs in the u vector. Instead, copies of the actual VARs are kept there. Since the hardware VARs are loaded from the u vector by a machine language routine, this greatly simplified the operation of this routine. However, unlike the PDP-11, the /6 VARs need only be reloaded when the real memory address of the process changes, instead of every time it is run.

A flag in the proc structure for each process is used to indicate that the hardware VARs must be reloaded from the copies in the u vector. This flag is set whenever the VARs in the u vector are changed -- for instance, when the stack or data segment grows. This flag is checked before running the process and, if set, the hardware VARs are reloaded. There are also times when the VARs are explicitly reloaded. This is usually done when the software VARs have been changed and the kernel then needs to access the user´s address space.

4.7. I/O and Interrupts

4.7.1. I/O Instruction Primitives

On the /6, I/O can not be performed directly from C as it can on the PDP-11 with its memory mapped I/O. Therefore, it was necessary to write small machine language routines that performed the I/O instructions. The problem was: how primitive should these routines be? A routine could be written for each I/O instruction that merely executed the instruction with appropriate data. These routines would be very short, only a few instructions. A similar approach could be taken, but each routine could perform retries if the channel was busy. These routines would be somewhat longer, maybe 1Ø to 2Ø instructions. Lastly, a very high level approach could be taken: having a routine, for instance, that would start a DMA operation. Such a routine would take several parameters and execute several different I/O instructions.

Since the last approach was so different from the PDP-11, it was not clear what the primitives should be.

Therefore, it was not considered any further. It is necessary to further explain the difference between the first two approaches. When an I/O instruction is issued, it is possible that the channel may not be ready to accept the instruction. However, if the instruction is retried several times, the channel may soon become ready. The channel indicates its readiness by setting the \`zero´ condition code bit. This bit may be tested to determine if a retry is necessary. In the first approach described above, it would be necessary to do these retries from the C program. In the second approach, these retries would be done in the machine language assist.

The second approach is the one that was adopted. It implements primitives very close to those of the PDP-11, thus allowing many of the PDP-11 device drivers to serve as a basis for /6 device drivers. The primitives implemented retry each I/O instruction 2Ø times before failing. If the channel has not responded after 2Ø tries, it is probably broken. A value of -1 is returned as the result of the function call to indicate that the operation could not be performed. Note that -1 may also (but rarely) be a valid return value from some of the input instructions. Few of the existing device drivers check this returned value, assuming that if the device is no longer functional, it will be evident from other status indications. A more fault tolerant system may wish to check these return values.

As was seen in Chapter II, different I/O instructions must exist for each channel and unit that must be addressed. Therefore, the I/O procedures could not directly execute I/O intructions, or a different I/O procedure would be needed for each device. The solution involved building an array of I/O instructions for each device (actually, for each channel/unit). This array contains all of the possible I/O instructions, with the appropriate channel and unit for the device. The array is initialized by the `devinit´ procedure, which is called by each device´s open routine. The I/O procedures described above are passed the address of this I/O instruction array. The procedures then use an Execute Memory (EXM) instruction to execute the appropriate I/O instruction. The devinit procedure also initializes the interrupt vector for the device.

4.7.2. Priority Levels

The most difficult primitives to model on the /6 were the set-priority-level primitives of the PDP-11.

The /6 has no equivalent mechanism for directly manipulating the priority level of the interrupt system. Instead, the Enable/Inhibit register must be manipulated when it is necessary to prevent a class of devices from interrupting. The use of the PDP-11 interrupt levels is as shown in the following table:

level	devices
7	no device interrupts
6	clock
5	disks, magtapes, multiplexers
4	terminals, paper tape
3	not used
2	not used
1	used by clock routine
Ø	base priority

Note that the clock is the highest priority device, followed by DMA devices and simple terminal devices. Priority level 1 is used only by the clock routine.

The interrupt structure of the /6 is quite different. A typical configuration is shown below:

level	device
Ø	hard parity error
1	soft parity error
4	console terminal
6	magtape
7	disk
9	DMACP
21	clock

Note that the clock is the lowest priority device and the console terminal is the highest priority device.

This is exactly the opposite of the PDP-11.

There are procedures named spl#, where # is a priority level, that are used to set the processor priority level. Each of these routines returns the previous priority level. A procedure called splx is used to set the priority level from a previously saved value. On the PDP-11, these routines are very simple, using the spl instruction. On the /6, these routines had to manipulate the E/I register in unorthodox ways.

Although these routines are fairly general in description, they are used in quite specific ways. The most common use is to raise the processor priority while manipulating certain data structures so as to provide mutual exclusion to the data structures. The priority is then lowered after manipulating them. The spl4 and spl5 procedures are used by device drivers so that they are not interrupted while manipulating private data bases. The spl6 and spl7 procedures are used when manipulating global data bases such as the clist and buffer cache.

To model the spl procedures on the /6 it was necessary to group the device interrupts together to simulate the priority levels of the PDP-11. There are three groups. The first group contains only the hard parity error interrupt and is always enabled. The second group contains only the clock. The third group contains all other devices. The spl4 and spl5 routines (which are the same) inhibit interrupts for all devices. The spl6 and spl7 routines (which are also both the same) inhibit interrupts for both the clock and all devices. This allows the normal uses of the spl procedures to function compatibly with the PDP-11 versions.

4.7.3. Clock Interrupt Handler

The only unusual use of the spl procedures on the PDP-11 is in the clock interrupt routine. It uses these routines to lower the current priority level. Before executing timeout procedures, the priority is lowered to level 5. Before performing the once per second housekeeping, the priority is lowered to level 1. The clock interrupt routine checks that the previous priority level was zero before lowering it to 1. This prevents the clock from executing this section of code when it has interrupted another interrupt routine, or when it is already executing it.

The /6 clock interrupt routine is quite different. Since the clock is the lowest priority device, it must raise the processor priority before executing the timeout procedures. Also, an entirely different mechanism is used to perform the once per second housekeeping. Every second, if the processor was at base priority, a software interrupt is triggered on level 22. This interrupt is handled by the second clock routine which performs the once per second housekeeping. Having the second clock execute at a priority level lower than the main clock achieves the same effect as lowering the priority to level 1 in the PDP-11, but is much simpler and safer on the /6.

Another difference between the PDP-11 clock and the /6 clock is that the /6 clock interrupts with a frequency of 120 HZ, as opposed to 60 HZ on the PDP-11. This caused only minor problems in the kernel since the clock frequency is an easily changed `define´d constant. Many user programs assumed that the clock frequency was 60 HZ. The `time´ command was the most obvious offender. Timeouts caused the only problem in the kernel. Most routines that requested a timeout assumed that the clock frequency was close to 60 HZ. When the timeouts are used to time delays for terminals, quite different results are obtained if clock frequency is twice as fast. To correct this problem without changing all the procedures that requested a timeout, the timeout procedure was changed to scale its argument from 60 HZ to the actual clock frequency.

4.7.4. Device Interrupts

All device interrupts are handled by a common routine and then dispatched to the individual device drivers. Each device interrupt is first handled by a small interface routine. There is an interface routine for every device. The interface routine is responsible for saving the current processor state in a static data area reserved for the device, and for restoring the state before returning from the interrupt. After the state is saved, the interface routine enters kernel mode (it was running in monitor mode) and calls the common device interrupt routine. The interface routine passes several parameters to the common routine (named `call´), such as the device interrupt handler to call and a parameter to pass to it. The common routine pushes the argument to the device interrupt routine on the stack and then calls the routine.

The most important function of the common routine is performed when the device interrupt routine returns. The runrun scheduling flag is set when it is necessary to reschedule the processor. Before returning from the interrupt, the runrun flag is examined. If set, and the interrupt would return to user mode, a level 23 interrupt is requested by software. This interrupt is interpreted by the `trap´ routine, and the scheduler is called.

The next function of the common routine is to clear the idle flag. This will cause the scheduler to re-examine the process table; if it had been waiting in the idle loop. The common routine then enters monitor mode and returns to the interface routine that called it. The interface routine will restore the registers and return from the interrupt.

4.7.5. Processor Traps

Processor traps are handled in a manner similar to device interrupts. A common routine (`trap´ in the machine language assist) handles all processor traps, calling the C procedure `trap´ with an argument specifying the trap type. Small interface routines exist as for devices. The interface routines are slightly different -- after saving the registers and entering kernel mode they move the saved registers onto the stack and then call the machine language trap procedure. They also reset the interrupt they are servicing so that the processor is at base priority. The trap procedure never returns to the interface routine.

The common trap procedure is similar to the common device interrupt procedure. It moves arguments specified by the interface routine onto the stack and then calls the C `trap´ procedure. The first argument to the C trap procedure indicates the type of trap. The trap types are shown in the following table.

number	trap type
Ø	power down
1	power up
2	virtual memory fault
3	illegal instruction
4	stall alarm
5	interval timer
6	SAU fault
7	address trap
8	parity error
12	reschedule
14	BLU
15	unknown trap or interrupt

The machine language trap procedure performs the return from interrupt function itself. All external interrupts are held during the return sequence. The registers and PC to be restored are moved from the stack to a static data area. A BRL instruction can not be used to restore the execution state because the processor trap interrupt was reset by the interface routine. Therefore, a complex sequence of instructions, including instruction modification, is needed to properly restore the previous state.

4.7.6. Device Drivers

Most of the device drivers in the /6 UNIX system were based on similar drivers in the PDP-11 system. The basic structure and logic of the PDP-11 drivers were used in the /6 drivers, thus, greatly simplifying the task of writing /6 device drivers. Those device drivers that differ significantly from their PDP-11 counterparts deserve discussion here.

The /6 disk device driver was based on the PDP-11 RK05 device driver. It was first used to control the Harris 10.8 MB disk, which is quite similar to the RK05 in size and function. The device driver was later generalized to handle both the 10.8 MB disk and the 300 MB disk. The new driver is table driven and easily expanded to handle other disks. This was possible because all of the Harris disks are very similar in programming, requiring only small device specific routines.

There is one feature worth noting about the /6 disk driver. Since the disk driver controls many disks, the disk type is selected by the low 3 bits of the major device number. These bits are used as an index into the table of disk descriptions. Therefore, the major device numbers for the blocked and raw versions of a disk must agree in the low 3 bits. Generally, the major device number of the raw device is 16 more than the major device number of the block device.

The only other device driver that is at all unusual is the DMACP device driver. The DMACP is a Motorola 68ØØ-based intelligent terminal multiplexer. The interface between the DMACP and the /6 is quite sophisticated and general. Before it can be used, the DMACP memory must be loaded with the microcode that supports the terminal multiplexing function. The specific code used for UNIX is the same as that developed by Gingell for Vulcan [4]. Although this DMACP code is more than sufficient for Vulcan, it lacks some of the functionality that UNIX would like. It does not support either baud rate changes or use of the modem control signals. These are both software limitations; removing them is only "a simple matter of programming."

The DMACP microcode is loaded by a user program from a file. A memory special device is used to access the DMACP memory for loading. The DMACP microcode was developed using a cross-assembler on Vulcan; no such cross assembler currently runs on UNIX/24V.

4.8. System Initialization

This section provides a brief description of the functions of the machine language system initialization procedure. The system initialization procedure performs three major functions. First, it initializes the first 256 VARs for the kernel (Figure 4.1). The first VAR is initialized as read/write since many data items are stored in the first page of memory. The following VARs, up to the end of the text segment, are initialized as read only. From the end of the text segment to the end of the kernel, VARs are initialized to read/write. The VARs from the end of the kernel up to the last VAR used by the kernel are initialized to no access (demand page). The last VAR used by the kernel (255) is initialized to the scheduler´s u vector, which starts at the first page after the kernel data. The kernel´s VBR and VLR are also initialized.

The second function is to clear the bss segment of the kernel. This sets all otherwise uninitialized variables to zero and clears the scheduler´s u vector.

The last major initialization function is to determine how much physical memory is present in the machine. This is done by storing a -1 in the first word of every page and then reading it back. If a -1

```
page                              access

  Ø  |-------------------|
     |   vectors/data    |   RW
  1  |-------------------|
     |                   |
     |                   |
     |                   |
     |       text        |   RO
     |                   |
     |                   |
     |                   |
  t  |-------------------|
     |                   |
     |                   |
     |                   |
     |       data        |   RW
     |                   |
     |                   |
     |                   |
  s  |-------------------|
     |/  /  /  /  /  /   |
     |  /  /  /  /  /  / |
     | /  /  /  /  /  /  |
     |/  /  /  /  /  /   |
     |  /  / hole  /  /  |   none
     | /  /  /  /  /  /  |
     |/  /  /  /  /  /   |
     |  /  /  /  /  /  / |
     | /  /  /  /  /  /  |
255  |-------------------|
     |     u vector      |   RW
     |-------------------|

     t = bytes_to_clicks(&etext)

     s = bytes_to_clicks(&end)
```

Figure 4.1 - Kernel Logical Address Space

is not read back, that page does not exist. Note that no interrupt or trap occurs when accessing physical memory that does not exist.

After performing the above initialization tasks, the system initialization code calls the C main procedure. Main continues the initialization task, initializing some data structures, as in the PDP-11, and performing some machine dependent initializations. The machine-dependent initializations consist of clearing and freeing all unused physical memory, opening the console terminal, and enabling the parity and reschedule interrupts.

4.9. Swapping

As mentioned previously, two versions of the /6 UNIX system were constructed. The first version was a swap-based system, which was used to bootstrap the second version which supports demand paging. This section describes the parts of the swapping system that are not present in the paging system.

The swapping system was modelled after the PDP-11 system. Most of the same data structures and algorithms are used. Both swap space and memory allocation are handled the same as in the PDP-11 system. In fact, the differences can be isolated to two areas: the movement of data between swap space and memory, and the setting up of user virtual memory registers.

The `swap´ procedure is used to move program images between swap space and memory. On the PDP-11, an entire program image can be moved with one I/O operation. Although this is also possible on the /6, several factors make it more difficult. Since the /6 UNIX disk blocks are really pseudo disk blocks that do not completely fill an integral number of sectors, it is not possible to write more than one block at a time. For example, if an attempt was made to write two blocks by specifying the starting disk address of the first block and specifying a word count of 1024 (2 blocks), the beginning of the second block would be written at the end of the last sector comprising the first block. This part of the sector is not normally used and would not be readable if the blocks were read one at a time.

Fortunately, the /6 supports chained I/O operations. This allows I/O operations with several different addresses and word counts to be specified in one operation. Using the example above, the write operation for the first block would be specified, followed by the write operation for the second block. The chained I/O operations are stored in a contiguous area of memory. The device channel automatically retrieves the next I/O command in the list if the previous command specified chaining. Therefore, to move an entire program image, a chain list can be built specifying the I/O operation for every block of the image.

Chain lists are built by the `bchain´ procedure. The chain list is built in an array that is private to every disk device. The size of this array limits the number of I/O operations that can be chained together. At present, such arrays are large enough to chain together operations for up to 64 blocks. This is a trade-off between space (size of the array) and efficiency (number of I/O operations needed to move an image). Therefore, the swap procedure must divide the image into 64-block pieces and issue an I/O request for each one. The disk driver will (by calling bchain) break the request into a chain of single block requests.

The other major difference between the swapping system and the paging system is in the handling of the user virtual memory prototype registers. In the swapping system, the VAR prototypes are stored in the process´ u vector, as described previously. In the paging system, the VAR prototypes are not directly available. Instead, the u vector contains pointers to other data structures from which the VARs are created and the loaded. The next chapter contains the details of this scheme.

4.10. General Portability Comments

There are several assumptions implicit in the design of the UNIX kernel that affected its portability to the /6. The problems centered around the addressability of memory and the use of pointers.

The first problem encountered was the manipulation of the user's PC by the kernel. The kernel assumed that all memory was inherently byte addressible and, therefore, that the PC must be of type "pointer to character" (char *). On the PDP-11, this causes no problems, even though the PC never points to a byte that is not word aligned. However, on the /6, the PC is a word pointer, which is quite different from a byte pointer in representation. When the kernel needed to increment the user's PC (for instance, to skip an instruction) it would add the number of bytes per word to the PC. Since the /6 PC is not a byte pointer, this type of operation did not work correctly. The solution was to change all uses of the user's PC to manipulate it as a word pointer instead of a byte pointer. A more general solution would be to parameterize the type (word, long word, or byte pointer) of the PC and manipulate it appropriately.

A similar problem occurred when referencing user memory, for instance, to gather system call parameters. Several procedures exist to transfer bytes and words between user and kernel memory. The most often used ones are: fubyte and fuword (fetch user byte or word), and subyte and suword (set user byte or word). It is clear that a byte address should be provided to fubyte and subyte. However, byte addresses are also passed to fuword and suword. This would be necessary if non-word aligned words were to be manipulated. This does not occur on either the PDP-11 or the /6. However, all address parameters to fuword and suword are converted to byte addresses; this conversion is free on the PDP-11 but not on the /6. This caused no real problems on the /6, just unnecessary overhead. Since it may be desirable to perform such operations on other machines (such as the VAX), the use of user addresses should be parameterized in a manner similar to that suggested for the PC.

Another problem caused by the varying representation of addresses on the /6 occurred when unions of pointers were used. The `buf' data structure contains an element that is a pointer to a buffer of data from a block on a disk. Depending on the contents of the buffer, it was desirable to reference the buffer using different types of pointers. Therefore, the pointer to the data buffer is really a union of pointers of various different types. Since all pointers on the PDP-11 have the same representation, this causes no problems and satisfies the type constraints of C. However, on the /6, if the address is stored in one member of the union as a byte address, and then referenced by another member of the union as a word address, it will have the wrong format. The solution was to store the address as a byte address and insert type conversions whenever it was necessary to reference it as a word address.

4.11. Features Not Supported

Those features described in the UNIX Programmer´s Manual that are not supported in UNIX/24V are discussed in this section. Some of the problems expected to be encountered when implementing them are also discussed.

4.11.1. Accounting

The `acct´ system call is currently not implemented. The only problem one would expect to encounter is with the pseudo floating point representation of times. It may be desirable or necessary to rewrite the `compress´ procedure.

4.11.2. Lock and Phys System Calls

Neither the `lock´ nor `phys´ system call is implemented. The lock system call, as implemented on the PDP-11, will probably not function as desired in the paged /6 UNIX system. If it is desired to lock in core every page the process is currently using or demand pages in, extensive changes may be required.

The phys system call should be straightforward to implement. Since only three of the four possible flag values in the user´s page table are used, the fourth could be used to indicate a mapping to physical memory. A special case in the `sureg´ procedure would recognize the new flag value and load the VAR appropriately.

4.11.3. Process Profiling

Process profiling has been implemented in the swapping system and appears to work correctly. However, several problems arise in the paging system. The buffer into which the profiling information is accumulated must be locked into core. The reason for this is that the profiling is done by the clock interrupt routine, at which point demand page interrupts can not be processed correctly. Therefore, the fix allowing profiling to work under the paging system is to have the profile system call lock all pages of the profile buffer into core. Of course, the pages must be unlocked at some time in the future.

4.11.4. Process Tracing

No attempt has been made to implement any of the features supported by the ptrace system call. Several of the sub-functions that involve reading and writing the traced process′ address space and u vector are probably straightforward to implement. The real problem occurs with any of the tracing functions. It is not clear how to implement breakpoints on the /6. The /6 does support an address trap option, which would be very helpful for debugging. However, none of the machines on which UNIX/24V was developed had this option. Also, using the address trap option would allow only one breakpoint at a time. Much work is required in this area.

4.11.5. Raw I/O

At one point, raw magtape I/O was supported by the swapping system. Currently, no raw I/O to magtape or disk is supported. The difficulty lies in constructing the proper chain list for the I/O operation. In the paging system, the pages involved in the I/O operation must also be locked in core. The chain list can become very complex, involving both data chaining and command chaining. Most importantly, since the chain list must be built based on the page table of the process requesting the I/O operation, it is not clear how to pass the chain list from the high level I/O routines to the low level disk drivers, or how to have the low level drivers build the chain list. The I/O system will probably have to be extended in some way for this to work well.

4.11.6. Other

Because of the amount of code involved, neither multiplexed files nor the packet driver are supported. Although it is claimed that both of these subsystems are portable, it is not clear how difficult it will be to port them to the /6.

4.12. Summary

This chapter has presented the changes necessary to port UNIX and the modelling of UNIX primitives to facilitate the implementation of UNIX on the /6. Changes to data structures were generally straightforward. Modelling of I/O primitives was only slightly more difficult. Memory management using the

/6 virtual memory hardware was somewhat more difficult, but provided few major problems until demand paging was added. The most difficult problems were the creation of kernel mode and the modelling of the PDP-11 interrupt system with the /6 priority interrupt hardware.

CHAPTER V

PAGING

5.1. Overview

The paged /6 UNIX system fully supports the demand paging facilities of the /6 hardware. A 256K word virtual address space is available to each process, divided into 256 pages of 1024 words each. The system is driven by a primitive balance set scheduler and uses a working set [3] page replacement policy.

Several data structures are used to manage the user´s virtual memory -- page tables, the virtual memory map, and the core map shown in Figure 5.1 and 5.2. These data structures control the allocation and use of swap space and physical memory, on which the virtual memory concept is built.

As described in Section 2.6, the /6 hardware support for paging centers around a set of 4096 Virtual Address Registers (VARs). The VARs are thus a limited resource used to store `page tables´ for some subset of all the processes in the system. A contiguous set of VARs must be allocated for use by a process before it

```
page table      file system         swap space

|--------|      |---------|         |--------|
|        |      |         |         |        |
|        |----->|         |         |        |
|        |      |         |         |        |
|        |      |         |         |        |
|        |      |---------|     --->|        |
|        |                      |   |        |
|        |                      |   |        |
|        |                      |   |--------|
|        |   virtual memory map |
|        |                      |    core map    memory
|        |      |---------|     |
|        |      |         |     |   |--------|  |----|
|        |----->|         |------   |        |  |    |
|        |      |         |         |        |  |    |
|        |      |         |         |        |  |    |
|        |----->|         |-------->|        |-->|    |
|        |      |         |         |        |  |    |
|        |      |         |         |        |  |    |
|--------|      |         |         |        |  |    |
                |         |         |--------|  |----|
                |         |
                |         |
                |---------|
```

Figure 5.1 - Paging Data Structures

can be allowed to execute.

5.2. Virtual Memory

Each page of a user process is allocated from a system-wide pool of virtual pages, the size of which is limited by the amount of swap space allocated on secondary storage. Virtual memory is controlled by an array of structures, the virtual memory map, with one structure per page of virtual memory. The index of each structure in the array corresponds to the location of the virtual page in swap space. The (structures describing the) pages of virtual memory are linked together to form a list of free virtual memory.

5.2.1. Page Tables

A page table exists in the u vector of each user process. It is large enough to describe 256 pages, the maximum size of a process. Each page table entry may describe one of three types of pages; the type is specified by the flag field of the page table entry as defined in Figure 5.2.

A flag value of PT_FILE indicates that the page, when first referenced, is to be read in from the file containing the process′ image (page i, Figure 5.3). In this case, two additional fields specify the disk

```
/*
 * the virtual memory map
 */
struct vmmap {
    unsigned    vm_rpage:8;             /* real page number */
    unsigned    vm_refcnt:10;           /* reference count */
    unsigned    vm_loaded:1;            /* loaded flag */
    unsigned    vm_cow:1;               /* copy on write */
    unsigned    vm_text:1;              /* text page */
    union {
        unsigned      vm_bsp;      /* bs procs using page */
        struct vmmap    *vm_next;  /* free list */
    } vm_un;
};

/*
 * page table entry
 */
#define PT_FILE  0     /* fill from file on demand */
#define PT_ZERO  1     /* zero fill on demand */
#define PT_ALLC  2     /* page is allocated in v. m. */

typedef union {
    struct {     /* used for file fill */
        daddr_t pte_blk1;  /* first block of page */
        daddr_t pte_blk2;  /* second block of page */
    };
    struct {     /* used otherwise */
        unsigned pte_flags:2;  /* PT_xxxx flags above */
        unsigned           :22; /* overlap with pte_blk1 */
        struct vmmap *pte_vmp; /* pointer to virtual page */
    };
} pte;

/*
 * the core map
 */
struct coremap {
    unsigned cm_dirty:1;   /* has been written */
    unsigned cm_wanted:1;  /* someone waiting for page */
    unsigned cm_locked:1;  /* locked in I/O */
    unsigned cm_intrans:1; /* in transit */
    unsigned cm_alloc:1;   /* page is allocated */
    union {
        struct vmmap   *cm_vmp;  /* ptr to virtual copy */
        struct coremap *cm_next; /* if free */
    } cm_un;
    unsigned cm_used;     /* bs procs that have ref'ed */
    unsigned cm_wsp;      /* bal set procs in w.s. of */
};
```

Figure 5.2 - Data Structure Declarations

addresses of the two halves of the page. These addresses were converted from file-relative addresses to absolute disk addresses when the process was initiated. (The loader has arranged for the pages to start on block boundaries; the \`-z´ flag to the loader invokes this option.) Because of the overlap between the flag field and the disk block field, only 22 bits are available to specify the first disk block address of a page. This limits the maximum size of a file system volume to 6442450944 bytes (approximately 6.5GB), which is far larger than any disk drive currently available on the market.

The overlap between flag and address fields was used to reduce the size of the page table so that it would not consume too much of the u vector. There are three methods for removing this restriction, were it felt to be desirable.

1. The conversion from file-relative addresses to absolute disk addresses could be done at demand page time, instead of at process initiation time. In this case no addresses would need to be stored; the file offset could be deduced from the page number.

2. The size of the page table could be increased. In this case, one must be careful that enough space is left in the u vector for the kernel stack.

3. The block size could be changed from 512 words to 1Ø24 words. Only one disk address would then be needed to locate the page. This option will be discussed later.

```
                            block     executable file

                              Ø     |--------------------|
                                    |    a.out header    |
                              1     |--------------------|
                                    |                    |
                              2     |       page Ø       |
                                    |                    |
                              3     |--------------------|
                                    |                    |
                                    ~                    ~

page table entry for page i         ~                    ~
                                    |                    |
|-----------------------|    2i+1   |--------------------|
|ØØ|        2i+1        |--------->|                    |
|-----------------------|    2i+2   |                    |
|           2i+2        |--------->|       page i       |
|-----------------------|    2i+3   |--------------------|
23 21                   Ø           |                    |
                                    ~                    ~


page table entry for page j

|-----------------------|
|Ø1| /  /  /  /  /  /  / |
|-----------------------|
|  /  /  /  /  /  /  /  /|
|-----------------------|


page table entry for page k             virtual memory map

|-----------------------|           |--------------------|
|1Ø| /  /  /  /  /  /  / |    ---->|     |       | | |  |
|-----------------------|    |      |--------------------|
|                       |-----      |                    |
|-----------------------|           |--------------------|
```

Figure 5.3 - Page Table Examples

A flag value of PT_ZERO indicates that, when the page is referenced, a virtual and physical page will be allocated and filled with zeroes (page j, Figure 5.3). This mechanism is used for bss data pages and for stack pages that are allocated dynamically.

The third flag value, PT_ALLC, indicates that a virtual page has already been allocated for this logical page of the process (page k, Figure 5.3). Another field of the page table entry points to the structure describing the virtual page.

To allow sharing of virtual memory pages, several page table elements can point to the same virtual memory map. The reference count field specifies how many page tables are currently pointing to the particular virtual memory map. This count is used when deallocating a page to determine if there are still any users of the page or if it should be returned to the free list. In the current implementation, the only situation in which the reference count will be greater than one occurs when a data page is shared between two or more processes in a copy-on-write mode.

5.2.2. Copy on Write

Copy-on-write allows a data page to be shared between processes in a read-only mode. If any process attempts to write into the data page, a private copy of the page is made and the process continues as shown in Figure 5.4. This allows sharing of pages that would otherwise be private. As implemented in UNIX/24V, pages are only shared in a copy-on-write mode between a parent and child process. This is essentially the same scheme used by Tenex [1]. Copy-on-write is implemented on the /6 by setting the VAR for the page to indicate that the page is read-only. The copy-on-write flag in the virtual memory map for the page is used to distinguish copy-on-write pages from normal data pages. If a write violation occurs for a copy-on-write page, a new virtual and physical page is allocated, the original page is copied, and the process is allowed to continue with the new page, which is now writable. This mechanism greatly simplifies creating a copy of a process as requested by the fork system call. Instead of actually copying the process, all data pages of the process are marked copy-on-write.

5.2.3. Shared Text

Text segments may also be shared between processes. The page table for text segments is not kept in the process′ u vector. Instead, it is kept in a page table that is part of the `text′ data structure. The text structure describes all shared text programs

```
page table
for user 1
                   virtual memory map   core map     memory
|--------|
|        |      |--------|         |--------|   |--------|
|        |--===>|        |-------->|        |-->|        |
|        |  |   |        |         |        |   |        |
|--------|  |   |        |         |        |   |        |
            |   |        |         |        |   |        |
            |   |        |         |        |   |        |
            |   |        |         |        |   |        |
page table  |   |        |         |        |   |        |
for user 2  |   |        |         |        |   |        |
            |   |        |         |        |   |        |
|--------|  |   |        |         |        |   |        |
|        |  |   |        |         |--------|   |--------|
|        |---   |        |
|        |      |        |
|--------|      |--------|
```

Before write attempt into copy-on-write page

```
page table
for user 1
                   virtual memory map   core map     memory
|--------|
|        |      |--------|         |--------|   |--------|
|        |----->|        |-------->|        |-->|        |
|        |      |        |         |        |   |        |
|--------|      |        |         |        |   |        |
                |        |         |        |   |        |
                |        |         |        |   |        |
page table      |        |         |        |   |        |
for user 2      |        |         |        |   |        |
                |        |    ---->|        |-->|        |
|--------|      |        |    |    |        |   |        |
|        |      |        |    |    |--------|   |--------|
|        |----->|        |-----
|        |      |        |
|--------|      |--------|
```

After write attempt into copy-on-write page

Figure 5.4 - Copy-on-write

that are currently in use (or "sticky"). For each shared text segment, a count is kept of the total number of processes and the number of loaded (not swapped out) processes using the segment. The reference count for virtual pages that are part of a shared text segment is one, since only one page table points to the virtual page. The only other difference between text virtual pages and data virtual pages is that the text flag will be set in the virtual memory map for the page so that the page may not be written, even using the copy-on-write mechanism.

5.3. Physical Memory

A /6 CPU may have a maximum of 256 pages of real memory. This real memory is described by a table of structures, the core map, similar to that used to describe virtual memory (Figure 5.2). Those physical memory pages that exist in the machine, and are not used by the kernel, are available for allocation. Free memory pages are linked together in a free list. In addition, a flag in the structure for each physical page indicates whether that page is free or allocated. This flag is used when considering which virtual pages should be removed from physical memory, since only allocated pages need be considered.

One of the most important functions of the core map structure is to accumulate memory usage information. Several fields are used to maintain usage and modification information for pages. Since page usage and modification information is stored by the virtual memory hardware in special hardware registers, it must be copied to the software-maintained structure periodically. This updating function is performed by the `pgstats´ procedure, which is called from three places in the kernel. Pgstats is called before working set (Section 5.5) statistics are updated (by `wscalc´), since the working set algorithm uses the page usage information. It is also called when, during allocation of a memory page, it is determined that a page must be unloaded, since the decision as to which page will be unloaded is based on the values of the usage and modified flags for each page. Finally, pgstats is called from the scheduler (`swtch´) before switching to a new process. This is necessary because the raw page usage information is translated into per process page usage statistics by pgstats and any recent references to pages should be attributed to the current process, not the next process.

Several other fields of the core map structure are used when moving pages between swap space and physical memory. The locked flag is set whenever a page is active in an I/O operation (paging, or, eventually, raw I/O), and for any pages which must be kept in core, such as processes′ u vectors (and eventually profile buffers). The in-transit flag indicates that a paging operation is in progress for this page. The wanted flag is set when a process discovers that a page it needs is being demand paged into memory. To best illustrate the use of these flags, the procedure for loading and unloading a page is described below.

Loading a Page

1. If loaded and in-transit flags are set, set wanted flag and wait for page.
2. If loaded flag not set but real page number is non-zero, page is being unloaded; set loaded flag and continue.
3. Otherwise, allocate a physical page and lock it.
4. Link coremap and vmmap together.
5. Set loaded flag in vmmap.
6. Set in-transit flag.
7. Do the I/O.
8. Clear in-transit flag.
9. Clear locked flag.
10. If the wanted flag is set, clear it and wakeup anyone waiting.

Unload a Page

1. If locked or in-transit, can′t unload.
2. Clear loaded flag.
3. Set in-transit flag.
4. Do the I/O.
5. Clear in-transit flag.
6. If loaded flag is set, page has been reclaimed; take no further action.

7. Free physical page.
8. Clear real page number in vmmap.

5.4. Balance Set

In the swap-based UNIX system, only a small number of processes could co-exist in memory, with other processes residing in swap space. In the paged UNIX system, there is also a limited number of processes that may co-exist in memory. The limit is determined not by the total sizes of the processes, as in the swap-based system, but by the working set sizes of the processes. The set of processes whose working sets may co-exist in memory is called the balance set. Note that in both the swapping and paging systems only those processes that are actually using physical memory have VARs allocated to them. The concept of a balance set is used both in Multics [15] and Tenex [1].

The balance set concept was added to UNIX to simplify several data management problems. For this reason, its use as a scheduling aid is somewhat secondary and has not been fully developed. The balance set proved most useful in the management of page tables of resident processes and in the collection of working set statistics. Since page tables, and in particular sharing of pages, are managed by software, it was often necessary to inform several processes that the status of a page had changed so that it could reload its hardware VARs. The hardware VARs may be considered a cache of page tables. If the software-maintained page tables are changed, the copies in the cache (VARs) must be invalidated and updated before being used again. This occurs most often with shared pages, but also occurs when one process forces a page used by another process to be unloaded.

When the status of a page is changed by a process, other users of the page must be notified of the change. Notification of a change in page status causes the process to reload its hardware virtual memory registers with the new information in the software data structures. Difficulty occurs when attempting to locate all users of a page. The most obvious solution is to link together all the processes that are using the page. To do this, a separate linked list would be needed for each page and each process would need 256 links to other processes using the same pages. These links would most likely be added to the page table in the u vector, making it dangerously large (enough space may not be left for the kernel stack, forcing the size of the u vector to increase to 2 pages). Also, management of these links could become quite complex; they would have to be updated every time the process is swapped in since they could not be changed when it is swapped out. This solution appears to be overly complex and expensive in terms of space and time.

The solution chosen was to limit the number of processes that may need to be notified when a page is changed. This small number of processes is called the balance set, and consists of all the processes that are "swapped in" (i.e., the process′ u vector is in core). The balance set is limited to 24 processes so that each process in the balance set may be represented by a bit in a word. A word in the virtual memory map for each page contains a set bit for every balance set process that is using the page. A separate table, 24 elements long, contains a pointer to the proc structure for each process in the balance set. This makes location of all processes using a virtual page straightforward. Note that only balance set processes need be notified about changes in page status, as they alone have VARs allocated that may need to be changed.

Usage statistics for physical pages are maintained for each balance set process. A bit is set in a word in the core map structure when a page is referenced by a balance set process. Another word indicates which processes' working sets the page is a member of.

The swapper has been modified to remove processes from the balance set when requested. The core allocation routine sets a flag (bsout) when it discovers that a process must be removed from the balance set. It then awakens the swapper and waits for it to signal completion of its job. When awakened, the swapper first checks the bsout flag to determine if a process must be removed from the balance set. If so, it selects a process to remove, using the same criteria used in the swapping system. It then wakes up anyone waiting for a balance set removal.

5.5. Working Set

The working set page replacement algorithm used in UNIX/24V is based on that proposed for Tenex by Radelja [17]. This algorithm was designed to properly maintain working sets when shared pages are involved.

The working set algorithm is invoked once every τ units of process virtual time. Virtual time is measured using the interval timer option of the /6. The timer is started when a new process image is initiated by the exec system call. The timer runs only when the process is in user mode and is context switched between processes. The timer interrupts every τ time units (currently .5 seconds), at which time it is reinitialized and the working set algorithm is performed.

The following is a description of the working set algorithm. When a physical page is allocated for a process, as a result of a demand page or any other cause, it is marked as being in the working set of the process by setting the appropriate bit in the working set word for the physical page. At every working set timer interrupt, the following action is taken: if the page was referenced by the process, it is added to the working set of the process; otherwise, it is removed from the process' working set. In either case, the usage bit is cleared. No pages are removed from memory at this time.

When a physical memory page is needed, and the free list is empty, a search is made through the core map of all existing pages. Pages that have neither been referenced nor are in the working set of any process are available for replacement. Pages that have not been modified are chosen first. If the page has been modified, it is written out to swap space. No attempt is made to further discriminate between pages that satisfy the above constraints. If no such page can be found, a process is removed from the balance set and swapped out.

Removing a process from the balance set involves several housekeeping chores. First, all pages used only by the process are unloaded. Swapping out a process guarantees that at least one page of physical memory will be freed; that one page is the process' u vector. If all other pages of the process are shared with other processes, they will not be unloaded. Another housekeeping chore involves removing all references to the balance set process being swapped out. The appropriate usage and working set bits must be cleared in the virtual memory map and core map structures for all pages the process has used.

Note that when the process is swapped out, all information about which pages were in its working set is lost. The only working set information kept is the size of the working set. The working set size is computed by the sureg procedure, which loads the hardware VARs from the information in the page table. Sureg counts the number of memory resident pages the process is using and stores this number as the process' working set size. All scheduling decisions that were previously based on the total size of the process now use the working set size.

The working set size of a process is very important when considering whether or not to swap in the process. If there are not enough free memory pages to contain the working set and the u vector of the process, it is not swapped in. Note that shared pages, and most importantly shared text segments, are not accounted for. It is possible that there would be more than enough pages for the process if it used many shared pages that were already in memory. This is one area of the scheduler that requires further research.

CHAPTER VI

CONCLUSIONS AND FURTHER WORK

6.1. Conclusions

The work described herein has shown that it is possible to port the UNIX operating system to the Harris /6 minicomputer, which is vastly different from any other computer to which UNIX has been ported. The UNIX operating system, and most of the UNIX environment, has proved to be extremely portable. In addition, UNIX proved to be an excellent base on which to build a demand paged operating system. The demand paging extensions to UNIX were easily added with very little effect on other unrelated portions of the UNIX kernel. The resulting system combines the functionality of UNIX with the advantages of a virtual memory environment, providing an exceptionally powerful and usable environment.

The UNIX/24V environment, at both the command level and programming level, is quite compatible with other implementations of UNIX. However, due to the architecture of the /6 and the resulting implementation of C, the programmer is "encouraged" to program in a more strict and portable style. This has both advantages and disadvantages. The advantages of portability should be obvious at this point. The disadvantages are more difficult to specify and usually consist of certain beliefs or prejudices of the programmer. Many programmers believe that it is necessary to know what kind of machine they are programming on, so that they can write more efficient code. This code they believe to be efficient is often highly machine dependent. In many cases, such code can be replaced by equally efficient code that is also portable. However, there are situations in which it is desirable to know something of the underlying system. For instance, see the many papers on programming in a virtual memory environment [2] [5]. Such programming considerations often have an impact only on efficiency without jeopardizing portability.

The current implementation is by no means complete. The performance of the system is poor. A "seat of the pants" comparison indicates that performance, as indicated by the time required for the system to complete a task with no other load on the system, is worse on the /6 than on a smaller PDP-11/34 system. Also, the performance difference between the swapped and paged versions of the system is not significant. Timing tests with the C compiler showed a decrease of less than 10% in real time with a similar increase in system time when compiling the same small program on the paged system instead of the swapped system.

During an early stage of the system development (before the paged system) some performance monitoring of the kernel was performed in an attempt to pinpoint the slowness in the system. The I/O statistics feature was installed to verify that the disk was performing properly. All data indicated that disk transfer rates and seek times were as expected. The interval timer was also used to monitor the kernel by sampling the PC periodically. It was found that most of the kernel′s time was spent in three routines; idle, csv, and cret; as was expected. In addition, the routine bcopy was seen to occupy a large amount of the kernel′s time. Bcopy is used to copy bytes, usually between kernel buffers. Bcopy was changed to copy double words instead of bytes, if possible. This reduced the time spent in bcopy, but did not significantly improve system performance. It seemed that no one area of the system was at fault, making performance analysis of the system much more difficult.

6.2. Current Limitations

In addition to the unsupported features listed in Section 4.1Ø, there are several limitations of the current implementation of the system. The most important limitation is that processes can not be larger than 64K words. This limitation is really a limitation of the C compiler. The C compiler does not always generate correct code for references through pointers that point above 64K, see [12] for complete details. Because of this, the kernel support for processes larger than 64K words was never completed or tested. The only known problem area is the initialization of the user´s stack pointer; it should be initialized in LAC format instead of DAC format. This could be done either by the kernel or by the C runtime startup procedure (crtØ.s).

Another limitation is that the kernel contains no support for floating point. Although space is reserved in the u vector to save floating point registers and status, the calls to procedures savefp and restfp have been ifdef´ed out. These procedures must be added to the machine language assist, mch.s. In theory, adding these routines should be straightforward. The only anticipated problem is that it is possible, when restoring the floating point condition codes, to lock up the SAU and eventually the entire /6. This problem is documented in the Vulcan source code and is concerned with the ordering of instructions to restore floating point condition codes and enable SAU interrupts.

6.3. Further Work

This section presents some areas for further research that may help improve the performance of the system. Performance improvements fall into two catagories: those that will improve the overall performance of the system, and those that will improve the performance of the system under heavy load.

As was mentioned in Section 6.1, no particular part of the system was found to be consuming an inordinate amount of the kernel´s time. The reason for this is not clear. To some extent, the C compiler may be at fault. From examination of the code produced by the C compiler, an estimated 1Ø-2Ø% improvement is possible by adding a simple peephole optimizer. Also, several operations such as character pointer arithmetic and character string indexing that were cheap on the PDP-11 are quite expensive on the /6. Since characters are often used to contain flags for kernel data structures, this may be one source of unneeded overhead. Changing such flags from characters to integers would result in a slight increase in data structure sizes but may also decrease code size and execution time.

As was mentioned previously, there may be some justification for reevaluating the choice of block size. Analysis may indicate that performance improvements would result from using a block size equal to the hardware supported sector size. The impact of this change may be quite extensive and should be carefully investigated. On the other hand, several functional improvements may be realized by increasing the block size from half a page to a whole page. This would allow the implementation of extensions to the file system services that would permit a process to map a page of a file directly into its address space, as is done in Tenex.

The area requiring the most work is the part of the system added to support demand paging. The paging system should be instrumented to determine how to adjust the parameters of the working set algorithm. The possibility of keeping track of the working set when the process is swapped out, so that it may be preloaded when the process is swapped in, should be investigated. The performance of the modified scheduler working as a balance set scheduler should be measured. A complete rewrite of the scheduler may be necessary so that the balance set concept is handled more naturally and effectively. In particular, the swap-in decision could be based on more accurate information about shared pages.

REFERENCES

[1] Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX: a Paged Time Sharing System for the PDP-1Ø," CACM, Vol. 15 (March 1972), pp. 135 - 143.

[2] Brawn, B. S., and F. G. Gustavson, "Program Behavior in a Paging Environment," AFIPS Conference Proceedings, Fall Joint Computer Conference, Vol. 33 (1968), pp. 1Ø19 - 1Ø32.

[3] Denning, P. J., "Working Sets Past and Present," IEEE Transactions on Software Engineering, Vol. 6, No. 1 (January 198Ø), pp. 64 - 84.

[4] Gingell, R. A., "CWRU/Dmacp Control," internal memorandum, A. R. Jennings Computing Center, Case Western Reserve University, Cleveland, Ohio, February 198Ø.

[5] Guertin, R. L., "Programming in a Paging Environment," Datamation, Vol. 18, No. 2 (February 1972), pp. 48 - 55.

[6] Harris Corporation, "Reference Manual - Slash 6 Digital Computer," Sept. 1976.

[7] Harris Corporation, "Series 6ØØ Peripheral Device Programming Considerations," July 1975.

[8] Harris Corporation, "Universal Block Controller (UBC) Input/Output Channel," Sept. 1978.

[9] Harris Corporation, "Series 84ØØ Direct Memory Access Communications Processor (DMACP)," August 1978.

[1Ø] Johnson, S. C., and D. M. Ritchie, "Portability of C Programs and the UNIX System," Bell System Technical Journal, Vol. 57, No. 6, Part 2 (July - August 1978), pp. 2Ø21 - 2Ø48.

[11] Kernighan, B. W., and D. M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[12] Leffler, S. J., "An Implementation of the C Programming Language for the Harris /6 Minicomputer," Case Western Reserve University, Cleveland, Ohio, forthcoming Master´s Thesis.

[13] London, T. B., and J. F. Reiser, "A UNIX Operating System for the DEC VAX-11/780 Computer," Bell Laboratories Technical Memorandum 78-1353-4, July 7, 1978.

[14] Miller, R., "UNIX - A Portable Operating System?," Australian Universities Computing Science Seminar, February, 1978.

[15] Organick, E. I., <u>The Multics System: An Examination of Its Structure</u>, MIT Press, Cambridge, Massachusetts, 1972.

[16] Parmelee, R. D., T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts," IBM Syst. J., No. 2, 1972, pp. 99 - 130.

[17] Radelja, M. A., "A Performance Study of the TENEX Operating System´s Pager Module," Case Western Reserve University, Cleveland, Ohio, Phd. Thesis, Aug. 1976.

[18] Ritchie, D. M., "A Retrospective," Bell System Technical Journal, Vol. 57, No. 6, Part 2 (July - August 1978), pp. 1947 - 1970.

[19] Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C Programming Language," Bell System Technical Journal, Vol. 57, No. 6, Part 2 (July - August 1978), pp. 1991 - 2020.

[20] Ritchie, D. M., and K. Thompson, "The UNIX Time-Sharing System," Bell System Technical Journal, Vol. 57, No. 6, Part 2 (July - August 1978), pp. 1905 - 1930.

[21] Rose, C. W., "CWRUnet: The Case Western Reserve University Distributed Minicomputer Network," Case Western Reserve University, Department of Computing and Information Sciences, 1975.

[22] UNIX Programmer´s Manual, Vol. 1 & 2, Seventh Edition, January 1979.

APPENDIX A

SYSTEM INSTALLATION

UNIX/24V currently runs on the Wickenden TSA system. The system is booted from tape with the file system residing on the 300MB disk. The 300MB disk pack is divided into three logical areas for use by UNIX; see the disk device driver for the details of this division. The first area is used for the root file system, the second area is used for swap space, and the third area is a mounted file system containing all the system sources and is mounted on directory /usr. These three areas cover only half of the 300MB disk; the last area is nowhere near full.

The directory structure on the /usr file system is identical to that on the standard Version 7 distribution tape. The sources for the system kernel are in /usr/sys; the makefile in /usr/sys/conf will rebuild the system. To make a new boot tape the command file \`wrtape´ is used. After rebuilding the system, mount a tape on the tape drive and type "wrtape unix". This will cause a new boot tape to be written containing the new system. Once debugged, the new system should be copied to /unix so that programs that depend on the system namelist will work properly.

To build and move a UNIX/24V system to another /6 is somewhat more difficult. Compared to the PDP-11, there seems to be much less standardization on the channel/unit numbers and interrupt levels for devices on the /6. There are two main problems encountered when moving to another /6. The first problem is moving the file system. If the destination system has a 300MB disk drive, the disk pack could be moved directly. Most likely this will not be the case. The method used in the past has been to use a subset of the standalone support programs to move an image copy of the root file system. The program bcopy (in /usr/src/cmd/standalone) is a standalone program to copy a file system image from tape to disk. Bcopy must be linked with the appropriate disk driver and tape driver containing the proper channel/unit numbers for the destination system. The program maketape is used to write bcopy out to tape. The root file system image can be copied to tape after bcopy, or copied to another tape. Following is an example command sequence to accomplish this:

```
<build new bcopy>
% maketape mtboot bcopy
% dd if=/dev/rp0 of=/dev/mt0 count=4845
4845+0 blocks in
4845+0 blocks out

<mount tape and scratch disk on destination machine>
<use ROM bootstrap to boot from tape>
Input: mt(0,1)
Output: rp(0,0)
Count: 4845
<this will copy the root file system to the new disk>
```

The next problem involves building a UNIX kernel for the destination machine. Device definitions must be added to or removed from l.s and c.c in /usr/sys/conf as appropriate. Channel/unit numbers and interrupt levels may also need to be changed. Some opportunity exists for changing these as the system is booted. Before booting, raise sense switch 1; this will cause the system to halt after being loaded into memory. The device definition blocks in l.s can be patched in core with the new channel/unit numbers and interrupt levels, and the system can be restarted at location 040. During this process it will be convenient to have a copy of the namelist for the system being booted, as obtained from `nm´.

While running in single user mode, the /usr file system can be restored from a standard dump or tar tape. Only four devices are needed for single user operation: the console terminal, the disk containing the root file system, a magtape drive, and the line clock. Once the /usr file system is restored, a new system can be configured with device drivers for any other devices available on the system.

There are several things that could be done to simplify the task of moving UNIX to another /6.

1. A standard boot program (as on the PDP-11) that knows about all devices and file systems, would simplify booting UNIX and standalone programs.

2. A `mkconf´ program for the /6 would simplify the task of preparing configurations from machine descriptions.

3. Standalone versions of mkfs and restor would allow the preparation of a standard release tape, as for PDP-11 UNIX.

4. The process of patching configuration tables when the system is booted could be simplified by adding a configuration dialog module to the system. This module would be invoked as a sense switch option when the system is booted and would allow the user to specify the channel/unit numbers and interrupt levels for all devices known to the system. After the configuration tables are modified, the system would come up as normal. This module could be overlayed (somehow) or removed from a production system.

APPENDIX B

CONSOLE TERMINAL MODIFICATION

Harris packages all /6 systems with a TEC model 455 terminal for use as the console terminal. The TEC terminal is a block-transmit, upper-case only terminal that is interfaced via a twisted pair differential interface. Since UNIX prefers character-transmit, lower-case terminals, it was desirable to replace the TEC. None of the terminals available to us were equipped with differential-type interfaces. However, the TEC terminals were equipped with a special interface circuit that converted the differential signals into TTL level signals for use by the terminal. In addition, the DECwriter II that was to be used for the console could be interfaced via TTL level signals. Therefore, the solution was to use the differential interface coming out of the /6, convert it to TTL using the board in the TEC terminal, and connect it to the DECwriter. The wiring between the TTL conversion board connector and the DECwriter connector is as follows:

```
TEC                 DECwriter
J1                    J4
---                 ---------
 9 ------------->  2
10 <-------------  4
 7 <------------>  1 gnd
```

In addition to the above change, it is also necessary to change the baud rate of the /6 interface from 9600 baud to 300 baud for use with the DECwriter. This change is easily accomplished by changing DIP switches on the interface board; consult the appropriate Harris documentation.

This digital copy was produced by the Case Western Reserve University Archives in 2020.

Original documents from the University Archives were scanned at 300 ppi in black and white or grayscale or color. Blank pages were not scanned. The images were OCR'd using Adobe Acrobat X.

Please send questions or comments to
University Archives
Case Western Reserve University
archives@case.edu
216-368-3320

Warning Concerning Copyright Restrictions
The Copyright Law of the United States governs the making of photocopies or other reproductions of copyrighted material. Under certain conditions specified in the Law, libraries and archives are authorized to furnish a photocopy of other reproduction. One of these specified conditions is that the photocopy or other reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a user makes a request for, or later uses, a photocopy or other reproduction for purposes in excess of "Fair Use," that user may be liable for copyright infringement