MUNIX, A MULTIPROCESSING VERSION OF UNIX

John Alfred Hawley

MUNIX, A MULTIPROCESSING VERSION OF UNIX

by

John Alfred Hawley, III

and

Walter de Brito Meyer

June 1975

Thesis Advisor: B. E. Allen

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MUNIX, A Multiprocessing Version of UNIX		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1975 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John Alfred Hawley, III and Walter de Brito Meyer		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1975 13. NUMBER OF PAGES 58
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

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

MUNIX
UNIX

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

Reported herein is the modification of a monoprocessing operating system, UNIX, designed to support an interactive multiprogramming environment on the PDP 11/40, 11/45 and 11/50 computers, to a multiprocessing operating system utilizing two PDP 11/50 processors and an expanded set of peripheral devices. The evolution of change from UNIX to the more responsive and efficient modified version called MUNIX is documented here to provide a basis for continuing work on the system and as a guid

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (cont.)

that might prove helpful to others engaged in similar projects. MUNIX, while retaining all the qualities of UNIX, provides increased system efficiency and improved response to users. Although the project is considered a success the full potential of MUNIX is far from being realized. Towards this end and several suggestions for improvement are made in the closing chapter.

MUNIX, A MULTIPROCESSING VERSION OF UNIX

by

John Alfred Hawley, III
Lieutenant Commander, United States Navy
B.S., University of Michigan, 1964

Walter de Brito Meyer
Lieutenant Commander, Brazilian Navy
B.S., U.S. Naval Postgraduate School, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1975

ABSTRACT

Reported herein is the modification of a monoprccessing operating system, UNIX, designed to support an interactive multiprogramming environment on the PDP 11/40, 11/45 and 11/50 computers, to a multiprocessing operating system utilizing two PDP 11/50 processors and an expanded set of peripheral devices. The evolution of change from UNIX to the more responsive and efficient modified version called MUNIX is documented here to provide a basis for continuing work on the system and as a guide that might prove helpful to others engaged in similar projects. MUNIX, while retaining all the qualities of UNIX, provides increased system efficiency and improved response to users. Although the project is considered a success the full potential of MUNIX is far from being realized. Towards this end several suggesticns for improvement are made in the closing chapter.

TABLE OF CONTENTS

I. INTRODUCTION

For the purpose of student and faculty research at the Naval Postgraduate School in the Computer Science and Signal Processing disciplines, computer hardware and software was purchased in the fall of 1974. The purchase consisted of two PDP 11/50 processors manufactured by Digital Equipment Corporation, a CSP 30 processor made by Computer Signal Processors Incorporated, discs, terminals, graphics display devices, various types of primary memory, and other peripherals (See Fig 1).

The ultimate goal of this relatively large system was to integrate all the components specifically to support signal processing research. When the entire system is not devoted to the real time realm of signal processing, it will be used as a tool by students and faculty involved in computer science related activities such as software engineering research. An agreement with Bell Laboratories provided the software to operate this hardware, in the form of UNIX, a monoprocessing/multiprogramming/time-shared operating system. However, UNIX, as delivered, did not have the complete set of capabilities necessary to fully utilize all system resources, nor did it provide all of the required environment for the envisioned research. Therefore, to accomplish the established goals its set of capabilities had to be expanded considerably. This expansion consisted, in part, of providing a real time environment for processes demanding it, giving the memory manager the ability to manage memories of different physical types and varying access times, in combination, as well as memory private to

and shared by the processors (memory that can be accessed by more than one processor in a multiprocessing enviroment is considered shared and memory accessable by only one processor is considered private in this context). In addition, and most importantly for the current paper, this expansion added to the system the ability to utilize both PDP 11/50 processors together in a multiprocessing mode or alone, as desired.

Changing UNIX to a multiprocessing operating system is the topic of this thesis. The alteration was approached with several goals in mind:

* changes made, of necessity, would have to be compatible with other system improvements such as the aforementioned real time and memory management changes;

* efficient operation was the goal; consequently, cpu overhead would have to be kept to a minimum;

* related to the above item is response time to the user which had to remain as short as possible while at the same time increasing the total throughput of the system;

* the resulting system would have to be flexible; in slightly different terms, it could not be configuration dependent; as an example, when one of the cpu's becomes unavailable, the same system should run in a monoprocessing mode with as little effect as possible on the user.

The process of accomplishing these goals and producing the desired multiprocessing system was approached in a systematic way and can be outlined as follows:

* investigate the proposed hardware configuration in order to determine how its properties might impose limitations on the new system or how they might be taken advantage of;

* document UNIX so that it could be studied and understood and become intimately familiar with it;

* identify critical areas of UNIX where changes and additions to the code would be necessary in order to accomplish the task at hand;

* make those changes and additions defined in the previous step;

* try the new system and test it until satisfied that it operates as desired;

* begin testing the modified system to determine its efficiency and where improvements might be made; this step is a perpetual one that will continue through the life of the system as new hardware is added, new applications are attempted and as advances in the science of computers dictate.

II. GENERAL MULTIPROCESSING CONSIDERATIONS

A. GENERALITIES

In order to enhance throughput, reliability, computing power and allow parallelism, additional processors can be added to a computer system. When an operating system utilizes the capabilities of more than one processor, it is considered a multiprocessing system. There are various techniques, differing in the degree of process scheduling sophistication, that can be utilized when connecting and operating a multiprocessing system.

B. MULTIPROCESSING METHODS

Multiple processors can be connected and operated as separate systems with each processor having its own dedicated resources. Job scheduling for this separate systems approach is accomplished by manually assigning a job to a chosen system. Considerable configuration flexibility exists in this type of multiprocessor set up since assignment of the available memories and I-O devices can be made in most any combination desired ranging from giving one processor all the resources, in order to service a large demanding job, to dividing the resources equally between the processors. Of course the full power of a system's resources cannot be utilized when they are divided. Only in

a system where each processor has access to all possible resources can full advantage be taken of system attributes.

Another way to multiprocess is sometimes referred to as the master/slave method. In this scheme one processor is assigned the task of process scheduling and control. This master processor then instructs its slaves, the other processors in the system, in what to do by assigning them processes to run and starting them up. When unusual situations are encountered the subordinate processors must stop and wait for the leader to tell them what to do. In this type of set up, if the master processor is busy when a slave needs instructions, the slave must wait causing a loss of precious computing time. Moreover, if the master processor is down or not available for some reason, the entire system is unusable, at least until a new master can be brought up.

The most popular and efficient method used in multiprocessing is one that treats all processors as equals. A more strict definition of a multiprocessing system according to Enslow [1] would include only this type of system. Enslow's definition is:

* "A multiprocessor computer is a system containing two or more processor units of approximately comparable capabilities
  - all having access to shared common memory,
  - all having common access to at least a portion of the IO devices,
  - all being controlled by one operating system that provides interaction between the processor and the programs they are executing at the job task step, data set, data element and hardware levels".

In this type of system, processor scheduling is

logically decentralized and each processor runs almost as if the other processor(s) were not there. One of the advantages gained with this approach to multiprocessing is versatility; such a system should run regardless of the number of processors assigned (up to the maximum that the system is designed for). This kind of homogeneous processor scheduling was chosen, as a goal, for the modified version of UNIX called MUNIX.

Since every processor in such a multiprocessing system is treated equally each must be able to perform all system operations. It follows that each processor must have the ability to access all system tables, databases and other resources. Because of this processor freedom, communication and coordination between processors is essential.

C. PROCESS SYNCHRONIZATION

A multiprocessing system will only be successful if it solves all the problems of properly synchronizing the processes it executes. Process synchronization depends on non-interference between processors and the passing of important information between them. Problems known in the literature as the "race condition" and as the "deadly embrace" or "deadlock" will bring an operating system to its knees if good interprocessor communications and coordination are not achieved.

1. Race Condition

Reference 2 defines a race condition as a condition that ..."occurs when the scheduling of two processes is so critical that the various orders of scheduling them result

in different computations".

In a multiprocessing operating system it is imperative that processors do not access or modify system resources and databases at the same time if race conditions are to be avoided; for if such events were allowed one processor could be altering a critical table at the same time another is accessing it. As an over simplified example, suppose one processor has searched the memory management tables for enough memory for a particular process and finds it. In addition, suppose a second processor is searching the same table for available memory that it needs. If the second processor chooses the memory it wants before the first has updated the table, to reflect the memory it has already allocated for its process, two processes may be allocated the same physical memory space; this will surely cause havoc when these processes are executed. Reference 2 contains several detailed examples of race conditions.

Though there are several ways to avoid a race condition, only one method will be discussed here. Dijkstra [3] defined a way to lock and unlock critical system resources so that only one processor at a time can get at them. Although his P and V operators are quite general and useful, this discussion will be limited to how they can be used to insure that only one processor at a time accesses critical system resources.

The following algorithms are illustrative of the P and V operations, where S is a semaphore (flag) assigned to the particular resource which is to be locked:

* THE P OPERATOR
  - S = S - 1
  - if S < 0 then wait else access to resource is allowed.

* THE V OPERATOR

  - S = S + 1
  - if S <= 0 (less than or equal to zero) send a signal.

The wait instruction, in this case, will cause a processor to wait until it receives a signal indicating that another processor is finished with this particular resource and it is free to access it. If there are more than two processors provisions for queuing them up while waiting for access to system resources would be needed. To each resource that requires a locking device a distinct semaphore must be assigned and initialized to the proper value; in the case where only one processor at a time is allowed access to a resource, the initial value will always be one (1).

A P operator is normally executed just prior to a processor entering a critical area of system code or a system resource; as stated above this causes the processor to wait if another is presently executing this section of code or accessing the same resource. When the access is complete, execution of a V operator signals any processor that might be waiting to get at the code or the resource and increments the semaphore to indicate this processor is finished with the critical code or resource.

D. DEADLOCK

The problem of deadlock or deadly embrace occurs when two tasks, both in active execution, each requests a resource that is assigned to the other and neither can proceed without the requested resource or give up the resource assigned to it. Deadlock is a problem found not

only in multiprocessing systems; it is a problem that must be investigated and solved for any multiple active program environment. The multiprocessing deadlock problem, however, is ususally more complex and requires a greater effort to solve.

III. HARDWARE CHARACTERISTICS

This chapter will be limited to only those hardware factors that directly and significantly affected the modification of UNIX. For a more detailed discussion of the PDP 11/50 hardware see references 7 and 8.

When an operating system is composed it is usually written to control a specific set of hardware. Because of this, an operating system's characteristics are greatly determined by that hardware. UNIX was written specifically for hardware based on the PDP 11/40, 45, 50 series processors and many of its properties are the result of this hardware's influence. The multiprocessing version of UNIX also derives many of its properties from the configuration of its hardware. The following is a listing of the hardware set up associated with each of the two processors to be operated by the multiprocessing system:

* Processor B:
    - 16K of MOS private memory (450 nsec. access time)
    - 32K of core shared memory (750 nsec. access time)
    - 16K of CSP shared memory (900 nsec. access time)
    - 1 DEC LA30-C terminal
    - 1 card reader (600 cards per minute)
    - 1 paper tape reader/punch
    - 1 printer/plotter (500 lines per minute)
    - 1 DEC DH11-AC communications multiplexor connected to (up to 16) remote terminals
    - 1 impact printer
    - 3 magnetic tape machines

- 2 DECpack disk cartridges
- 1 UNIBUS.

* Processor A:
  - 16K MOS private memory
  - 32K core shared memory
  - 16K CSP core shared memory
  - 1 disk controller
  - 3 CDC 9762 disks
  - 1 DEC LA30-C terminal
  - 1 Vector General 303I 3D vector display terminal
  - 1 Ramtek raster scan display unit
  - 1 UNIBUS.

Fig 1 presents a more detailed illustration of the proposed configuration.

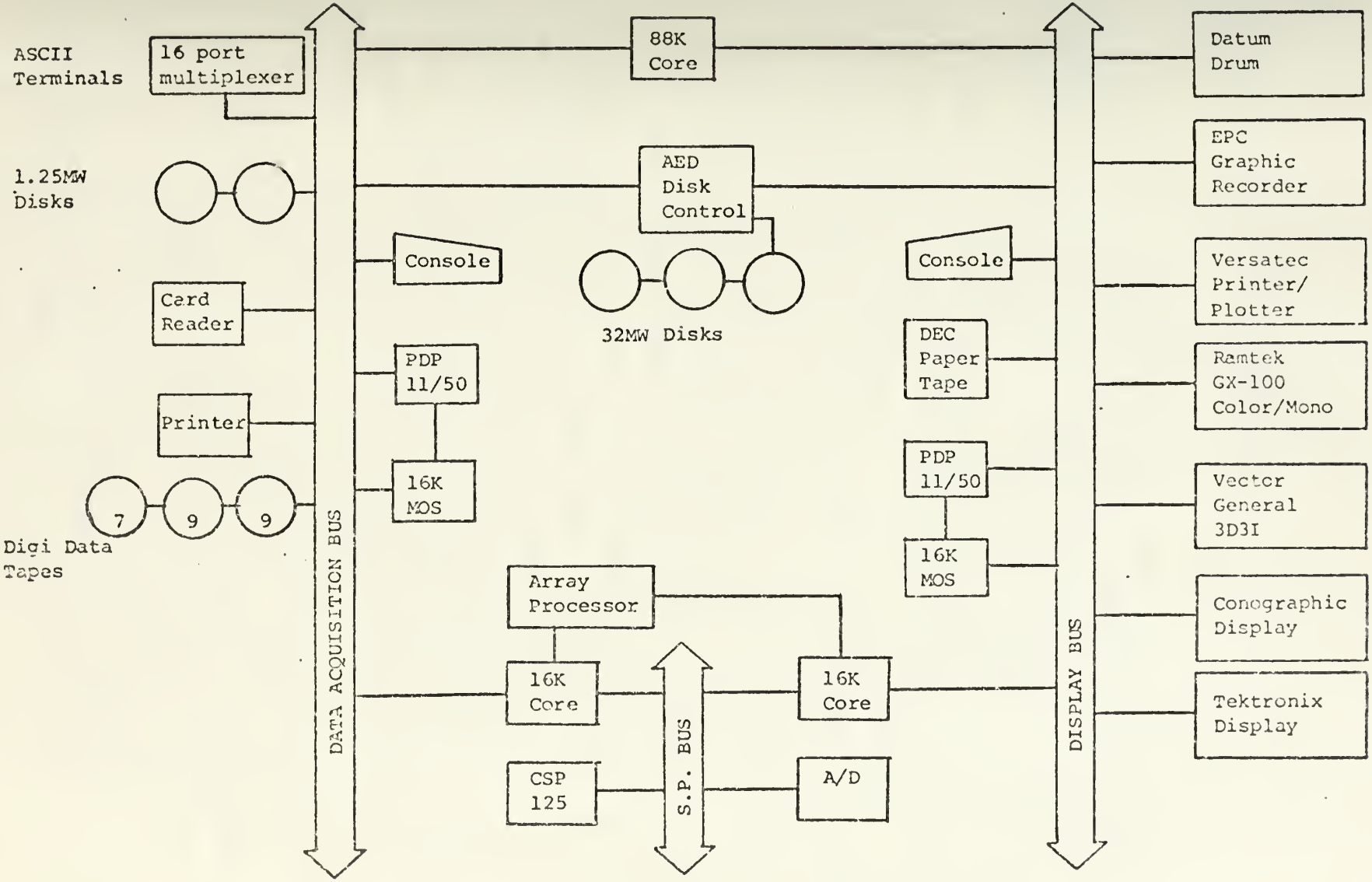


FIG. 1 - HARDWARE CONFIGURATION

A. THE PDP 11/50 PROCESSOR

The PDP 11/50 processor has been optimized to operate in a multiprogramming environment by giving it two sets of general registers and allowing it to run in any one of three separate modes: Kernel, Supervisor or User. In the Kernel mode the processor has complete control of the machine whereas in either Supervisor or User mode it is not allowed to execute certain instructions and can be denied access to the system's peripherals. It is an interrupt driven processor which automatically uses a stack when interrupted, a subroutine call is made, or trap instructions are executed. It uses a Processor Status Word (hereafter referred to as PS) in which information about its current status is kept. The information contained in the PS consists of: the current mode, the previous mode, the general register set in use, the priority level (there are eight priority levels used to mask out interrupts of equal or subordinate priorities), and several condition codes set by the results of certain instructions. The general registers are split into two sets (0 and 1) containing six registers each: registers r0 through r5. There are three registers referred to as r6 or sp which are used to hold the stack pointer for each of the three possible modes of operation, and one r7 or PC, which is the processor's program control register; it is used to keep the address of the next instruction to be fetched for execution. Two options that enhance and expand the capabilities of the PDP 11/50 are the Memory Management Unit to be described later, and the Floating Point Processor which is not important to this thesis. Both of these options were selected for the PDP 11/50's purchased.

B. DEDICATED RESOURCES

1. UNIBUS

The UNIBUS is a high speed bus that provides asynchronous and bidirectional communications between system devices. It allows the devices to send, receive and exchange data independently without involving the processor. Each processor has dedicated to it one UNIBUS and is consequently connected to only those devices attached to that UNIBUS. Therefore, except for the shared memory which is dual ported and connected to both processors through their own UNIBUS, there are no resources accessable to both processors. Because of this, there are severe limitations on the multiprocessing system operating on this hardware configuration. The result is that the two processors cannot be equal, as desired, in the modified UNIX. Since UNIX is a time shared system, the bulk of its work would be initiated at the remote terminals associated with processor B, with occasional inputs from the card reader and tape decks. Therefore, because processor A cannot get to those devices dedicated to B, it cannot share the heavy I/O loads expected in a time shared system, and B would be performing virtually all of the normal I/O.

Moreover UNIX places a swap file, comprised of processes active in the system but not desired in main memory, on a disk; and this disk is connected to B's UNIBUS. There is a lot of activity associated with this swap file, especially when the system is experiencing heavy utilization and there are a lot of active processes. Suffice it to say, during normal operation processor B would be doing all but very unusual I/O. Moreover, since the swap device is on the B side and processor A's private memory is on the other side

of our configuration, there is no direct way that a process can be loaded from the swap disk into that processor's private memory. What it all comes down to is: processor A would have to be a very junior partner in the new system. Its job would consist of picking a process to run and running it until it tried to do some I/O, which probably would not take very long, then give it to processor B to run.

One solution to the access problem is the future purchase of a hardware device (essentially a switch) that allows each processor to act as though it owns both UNIBUS's making possible access of all resources by either processor. If this purchase is made the problem of deadly embrace will have to be investigated.

A future hardware addition, not listed in the proposed configuration, is an RP11 disk pack that is dual ported. Because of the two ports, both processors can be hooked up to it through their own UNIBUS. Although this does not bring processor "A" to full equality with "B", it would allow for much more equitable sharing of the system load if this RP disk is used to support the swap file. Allowing access to the swap file from either processor would let B concentrate on the I/O initiated at the terminals while A could do much of the swapping of processes in and out of memory. In addition a direct path to either processor's private memory would be assured.

C. DEADLY EMBRACE

An advantage gained by each of the processors being unable to access the resources of the other is that the problem of deadly embrace is not exaggerated. UNIX, except

for some extreme cases where processes can cause a deadlock, obviates this problem and since one processor will never be waiting for its counterpart to finish with a resource (except main memory which cannot cause a deadly embrace) the problem is not changed in the MUNIX operating system.

D. INTERRUPTS

System interaction on the PDP 11/50 is accomplished through the use of the UNIBUS to make data transfers and to interrupt program execution and branch the processor to a service routine. The interrupts can be initiated by either software or hardware. In addition, the hardware provides the capability of sending an interrupt from one processor to another. To support efficient coding of the P and V operators used in locking a processor out of a critical resource while the other is accessing it, such interrupt capability is imperative. More will be said later when the actual method of coding these operators is discussed.

E. MEMORY MANAGEMENT

The PDP 11/50 used in the proposed hardware configuration has associated with it a Memory Management Unit which facilitates memory management and protection in a multiprogramming environment. With this unit attached the processor can take advantage of the three possible modes of operation to enhance its multiprogramming capabilities. These modes are:

* Kernel mode, in which the program has complete control of the machine;

* Supervisor mode, where the processor is inhibited from executing certain instructions and can be denied access to the system peripherals;

* User mode, where the restrictions are the same as for Supervisor mode.

With each of the modes is associated a set of 32 sixteen bit registers. Each set is subdivided into two groups of 16 registers; one is used for addressing all instruction fetches, index words, immediate operands and absolute addresses, and the other is used for all other addressing. Finally the sixteen registers of each group are divided into eight Page Address Registers used in converting a virtual address to a physical address, and eight Page Descriptor Registers used to indicate page length and access control.

When the Memory Management Unit is turned on, each address is interpreted as a virtual address containing information to be translated into an eighteen bit physical address. Utilization of the virtual addressing technique provided by the Memory Management Unit expands the maximum addressable memory space from 64K bytes, using a sixteen bit physical address without the unit, to 256K bytes. The construction of the physical address from the virtual address is quite straight forward. The sixteen bits of the virtual address are divided into two parts: the three most significant bits are used to determine which page register to get the physical page address from, and the other thirteen are used for the displacement in that page.

An executive program (an operating system) would usually be run in Kernel mode so that it can manipulate the entire usable address space, access all the peripherals, and execute the full range of instructions provided on the PDP 11/50. In order to execute a user process (a user process is considered to be any process other than the

operating system) the executive program need only load it into main memory, fill in the User Page Address Registers and the associated Page Descriptor Registers, set the PS to user mode and the PC to the first instruction of the user process.

IV. UNIX, THE OPERATING SYSTEM

UNIX is a general purpose, multi user, monoprocessing, interactive operating system written by Dennis M. Ritchie and Ken Thompson, among others, of Bell Laboratories for the Digital Equipment Corporation PDP 11/40, PDP 11/45, and PDP 11/50 computers. It offers many features seldom found even in larger operating systems, including:

* compatible file, device and interprocess I/O;
* the ability to initiate asynchronous processes;
* a hierarchical file system incorporating demountable volumes.

See reference 4 for an interesting introduction to these features and an overview of the system as a whole.

This chapter will present, and describe in the necessary detail, those properties and features which are important to the modification of UNIX into a multiprocessing system.

A. TRANSLATION AND DOCUMENTATION

UNIX, as received, was not documented; this lack of documentation makes almost any change to the system rather painful because of the accurate translation that is needed prior to attempting a modification. Consequently, before trying to make UNIX a multiprocessing system, a very large portion of the original had to be translated, documented,

and studied. Intimate knowledge of the system data bases, process scheduling algorithm, and processor flow through the system were required before any critical decisions on the approach to the problem at hand could be made.

Most of UNIX is written in a high level language called C, and a small portion is written in the assembler language of the PDP 11/50. C is a language which, at this writing, is only implemented under the UNIX operating system (see reference 5); in order to translate UNIX, a comprehensive knowledge of C had to be developed taking much time and effort. However, if the bulk of UNIX had not been written in a high level language, the translation effort, without documentation, most probably would have been doomed to failure. As it was, the translation and documentation phase of the project took a considerable amount of time, three months, which was to be expected considering the complexity of an operating system such as UNIX. References 4 and 6, although not detailed, give an overview of UNIX and were helpful during the translation and the documentation of the system.

The next several sections are devoted to a detailed description of those areas of UNIX that had to be understood in order to make the system into the multiprocessing system desired. These important areas were only a part of the total translation effort since much more of the system had to be deciphered in order to pick out and identify these areas. Only those areas of code critical to understanding the changes made in the production of the multiprocessing operating system are necessary to this discussion.

B. PROCESSES

A process, in the context of this paper, is defined as a computation that may be done concurrently with other computations. An active process in UNIX is a process that has been created by the system call "fork" and has not been terminated by the system. The function fork will be described is section D of this chapter. There are two exceptions to this definition: the first is the process known as process 0 which is essentially the system itself (it is the only process run in Kernel mode) and the second is the process referred to as "init", or process 1, which is a special process set up to perform some critical system tasks. Neither are created by a "fork" call and both are active throughout the life of the system (ie. as long as it is running). An image is the current state of a process with regard to all its registers, its I/O status, its immediate residence (whether in main memory or in the swap file), its stack and stack pointer, its PC and PS, and all other of its attributes needed by a processor to execute it correctly. This image is kept by UNIX in several system data bases which are updated for each change in a process' status. Other information necessary to the scheduling of processes, management of memory, and the management of devices is also kept in various system tables. Several of the more important data structures are presented in the following sections.

1. Process Table

One of the most important system data bases utilized by UNIX is the process table; most of the information kept in this table is used for scheduling decisions regarding the active processes. There are up to fifty active processes allowed in the system at any one time. Each of the process blocks in this table has fourteen data elements that may be filled in as needed to maintain the updated status of its

process. When a process is created it is assigned a process block; when it is deactivated by the system it relinquishes it. The elements of the process block that are important to this paper are p_stat, p_flag, p_addr, and p_wchan.

a. p_stat

The scheduling status of a process is kept in p_stat where:

* SSLEEP means that the process has been put to sleep and is not available to run;
* SWAIT means that the process is awaiting completion of I/O;
* SRUN means the process is ready to be executed;
* SIDL means the process is active but is not in any other status;
* SZOMB means that the system has terminated this process but still has to clear it out.

b. p_flag

The status of the process with regard to the memory manager is kept here; it is also used for scheduling purposes.

* SLOAD indicates process is in main memory;
* SSYS indicates process is a special system process;
* SLOCK indicates process should not be swapped out of main memory;
* SSWAP indicates process is resident in the swap file.

c. p_addr

The address, whether in main memory or in the swap file, of the beginning of a process' address space. It is not the address of the process itself but that of a data segment that is attached to the beginning of each process and carried with it throughout its existance. This data segment will be referred to as the "u vector" and will be discussed below.

d. p_wchan

This is the element used to contain the address of a channel that this process has been put to sleep on. It is not always a channel in the true sense of the word, such as an I/O channel. It is used to indicate the reason for stopping the execution of a process so that something more important can be done. The process will not be rescheduled until a wake up is performed on the channel it is sleeping on. Both "sleep" and "wake up" are system functions.

2. The U VECTOR

Each process has attached to its address space a 1024 byte segment that will be referred to as the u vector. It is attached at the beginning of a process' code, as depicted in Fig 2, and contains most of the per-process data kept by the system.

Some of the more important elements of this user vector are the stack, the stack pointer, memory management information, general register information (mostly kept on the stack) and a pointer to the process block belonging to this process. As stated above, each process is given a user vector, including the system (process 0). Upon initialization of the system, the Kernel Data Space Address

Register 6, KDSA6, is loaded with the address of the 1024 byte area to be used as the u vector for process 0. The user vector structure is then overlayed beginning at the u vector's relative address zero. This user vector contains all the process information not contained in the stack including the stack pointer (in u.u_rsav[0]) when the process is not being run. Fig 2 illustrates the overlay and a few of the user vector elements. These elements are explained in later sections of the paper. This overlay of the user vector structure takes up 238 bytes of the 1024 bytes in the u vector. The remaining space is saved for the Kernel stack which is used while this process is running or the system is performing in behalf of it. The Kernel stack pointer is then set to the u vector's relative address 1023. Each time a process is chosen to run, a similar procedure takes place: namely filling KDSA6 with p_addr from the process table, and the stack pointer from the user vector which was just made available. All other information needed to run the process is retrieved from either the stack or the user vector. This procedure is normally carried out by the system function "swtch", which will be presented in the section on important system functions.

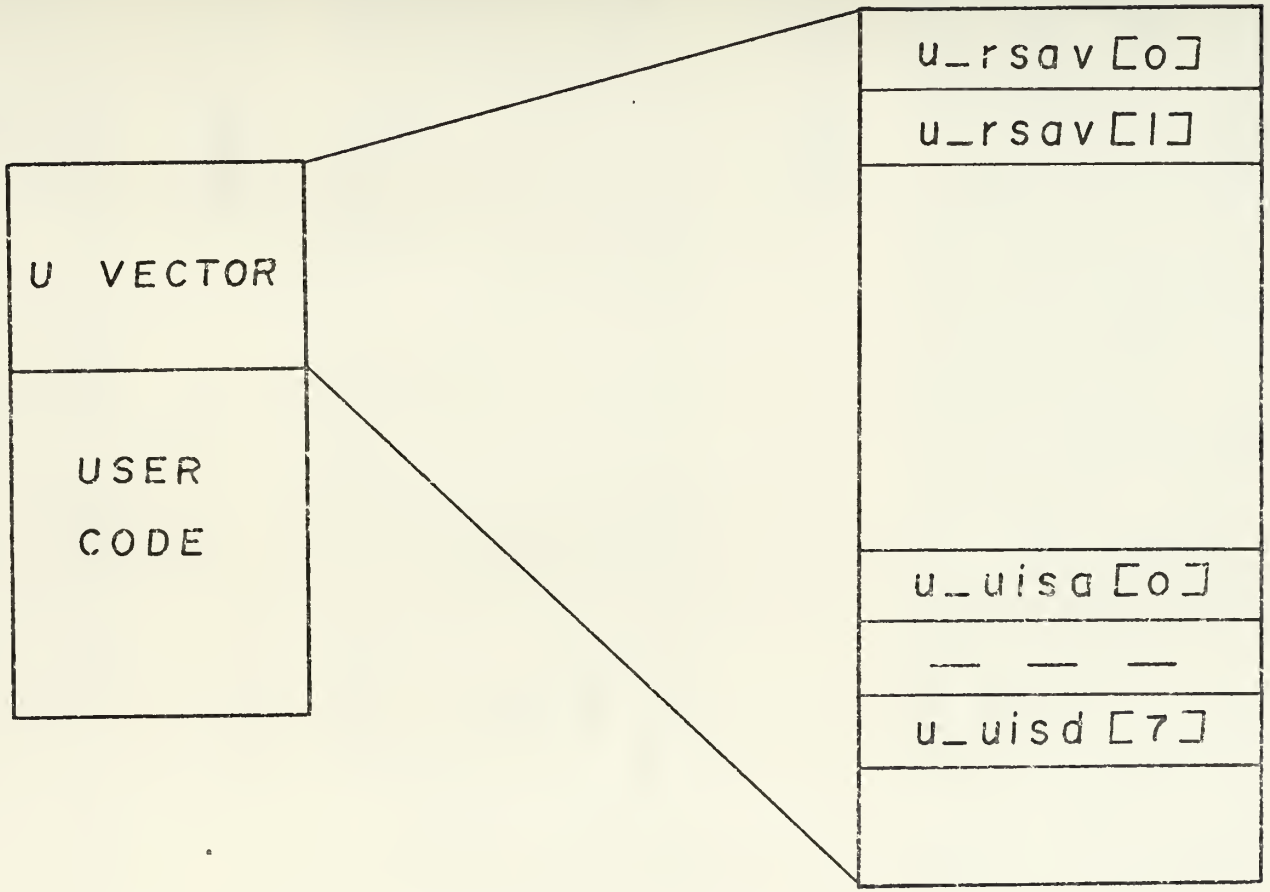


FIG. 2 - PROCESS IMAGE and U VECTOR

C. SYSTEM DATA BASES

An operating system requires much information about the status of its memory, its devices, its file system and the processes it is manipulating. UNIX has several data bases which it uses to save current information about the system. The process table and the u vector are just two of these. Other data bases of interest in this paper are those used to keep track of memory allocation and special buffer areas.

1. Coremap

The memory manager keeps the current status of main memory in the table called coremap. If memory is needed the function "malloc" is used to search coremap for available, free, memory; it then assigns it, if found, to the requester and changes its status to allocated. When the system is finished with a section of memory it invokes the function "mfree" which sets this section of memory to free in coremap, coalescing free memory if possible.

2. Swapmap

The memory manager uses swapmap in exactly the same way as coremap. The same system functions are used and the only difference is that the table keeps the memory status of the swap file rather than that of main memory.

3. Cfreelist

This data table contains status information concerning the

buffers used by character devices (eg. a remote terminal). This resource is manipulated by the system routines "getc" and "putc".

4. Bfreelist

Analogous to cfreelist, this data base is used to keep track of the allocation and the availability of the buffers used for the system block devices (eg. a disk). The system functions "getblk" and "brelse" update and use bfreelist.

D. SYSTEM FUNCTIONS

In the C language programs, routines and subroutines are all referred to as functions and they are all constructed using the same format. The UNIX functions that were important to its alteration into a multiprocessing operating system will be presented and described as necessary in this section. They deal, for the most part, with swapping processes in and out of main memory, scheduling processes to run, blocking processes, changing status of processes, and actually setting a process up to be executed or terminating the execution of a process.

1. "mch.s"

When bringing up UNIX, the system is loaded and execution is begun at physical address zero; the lower address space of the system is written in assembly language and has been given the name "mch.s". During the start up "mch.s" loads the Kernel memory management registers with

the proper page addresses (including KDSA6 with the address of the system's u vector as discussed in the u vector section), turns on the Memory Management Unit, loads the stack pointer for process 0 and branches to the C language routine called "main".

"Mch.s" also provides the code which, after an interrupt, sets up the process stack and branches the processor to the proper interrupt handler in addition to many other useful system functions.

2. "main"

The function "main" does some initialization of system data, turns on the clock in order to start clock interrupts, creates the "init" process (this is process 1; its purpose is to activate the remote terminals and ready the system for interactive use), fills in the process block for process 0 and branches to the function known as "sched".

3. "sched"

The "sched" function can be considered to be process 0. It is an infinite loop that is executed, is put to sleep, awakened, picked out of the list of processes, readied to run and executed again. It decides which processes are elegible for swapping from the swap file into main memory, swaps them in if there is room or makes room for them if it can find a process or processes to send to the swap file. When "sched" runs out of processes to bring into core it puts itself to sleep on the channel named "runin"; when it has processes elegible for swapping in but cannot find or make room for them, it puts itself to sleep on the channel called "runout". Process 0 is awakened and

put on the ready list periodically by the functions "clock" and "sleep" if sleeping on the runin channel and by the system routines "psignal", "xswap" and "wakeup" if sleeping on runout. Process 0 maintains the highest system priority so that it will be selected in "swtch" and executed first when awakened from dormancy.

4. "sleep"

"Sleep" is the system function that changes a process' status from ready (p_stat = SRUN) to that of SSLEEP or SWAIT depending upon its requested priority (SSLEEP for the high priority and SWAIT for the lower priorities). The process that is blocked by the "sleep" call is the process that was running when the sequence ending in this call began. It puts the processes to sleep on the channel and with the pricrity requested in the arguments of the function call. Subsequent to blocking a process in this manner "sleep" calls "swtch" to get a new process to be run by the system.

5. "swtch"

This function is invoked in several places throughout UNIX to accomplish the following:

* save the status of the current process with a call to "savu";

* search the process table for the highest priority ready process that is in main memory (its p_flag SLOAD bit is on); if there are no ready processes, the processor will idle until started again by an interrupt and the return from that interrupt; it will again search the process table for an elegible process; this cycle will continue until a

process is found; except under heavy system loading most of the processor's time is spent at idle and looking for a ready process in the "swtch" loop;

* start the new process up by returning to the routine that called "swtch" when this process was last running; this routine will then return to the process via a return from interrupt instruction or a normal function return, such as would be made for process 0 which simply returns from its last "sleep" call.

A switch from the current process is made each time a process is blocked by "sleep", when a clock interrupt occurs with the processor executing in User mode and the running process has used its time allowance, when a process has had an excessive number of interrupts handled by the function "trap", and when a process attempts to do something that causes it to be swapped out of core.

6. "savu"

The assembly language routine "savu", found in "mch.s", saves the contents of the stackpointer and of register five of the current process (register five is saved because it holds crucial information about the last subroutine call). They are usually stored in the first two elements of the user vector, in u.u_rsav[0] and u.u_rsav[1]. Prior to the "savu" call the other essential process status elements are saved by the various system activities that will not be explained here.

7. "retu"

Also an assembly language routine found in "mch.s", "retu" is called with p_addr as its argument. KDSA6 is

loaded with p_addr and the first two elements of the user vector (saved by "savu") are retrieved and loaded into the stackpointer and register five.

8. "sureg"

After "retu" only the User memory management registers need be loaded for the new process to be ready for execution. The function "sureg" accomplishes this using the user vector elements u_uisa[0] through u_uisa[7] and u_uisd[0] through u_uisd[7], loaded with the proper values by a call to the system function "estabur" at the process' creation, to load the memory management registers.

9. "wakeup"

This function causes each of the processes with the status of SSLEEP or SWAIT on the channel specified in the function argument to have their status changed to SRUN. If any of the processes awakened is not resident in main memory and process 0 is sleeping on runout, it is awakened. When "swtch" is next called process 0 will be scheduled and will attempt to find space for the newly awakened processes in order to load them into core from the swap file.

10. "trap"

"Trap" is a function that handles interrupts caused by:

* a bus error;
* an illegal instruction;
* bpt-trace trap;

* iot trap;
* emulator trap;
* a system function call;
* a programmed interrupt;
* a floating point exception;
* memory management violations and memory management traps.

The only interrupts interesting to this paper are those caused by system calls made by user processes (see references 7 and 8 for specific information concerning the other interrupts). Reference 4 contains detailed information on all system calls available in UNIX. In this case an interrupt is generated by the call and it is translated into a normal system function call by "trap". The system, still using the user's stack, services the call and returns control to the calling process. If the current process has had excessive interrupts handled by this function (more than fifteen), instead of returning to the process causing the interrupt, "swtch" is called to find a more deserving process to run.

11. "clock"

"Clock" is the function that handles the interrupts generated by the clock. It keeps track of and updates several system times that are used to change priorities and to make scheduling decisions. If the processor is in Kernel mode when the interrupt is executed "clock" simply returns to the system when it has updated all the necessary data elements. If in User mode when the interrupt is generated, and if the user process' time increment is used up, "swtch" is called at the completion of "clock".

12. "fork"

The "fork" function is used to create new processes. It calls the system function "newproc" which sets up the new process in the image of the one that invoked "fork". Upon return from "newproc", the old process continues, or "swtch" is called, while the new process is left ready to run. The new process is eventually picked by "swtch", initialization of its image is completed in "fork" and execution is begun.

13. Other Functions

The functions "malloc", "mfree", "getc", "putc", "getblk", and "brelse" have been presented in sufficient detail earlier and will not be repeated in this section.

E. SYSTEM INTERACTION

Once initialization is complete the system settles into a never ending loop based around "sched". It remains in this loop as long as a fatal system error is not encountered or shutdown of the system is not ordered. "Swtch" is also a very important part of this loop since between them they completely control memory and process scheduling for UNIX. Fig 3 is a flowchart that illustrates the basic system interaction and scheduling flow.

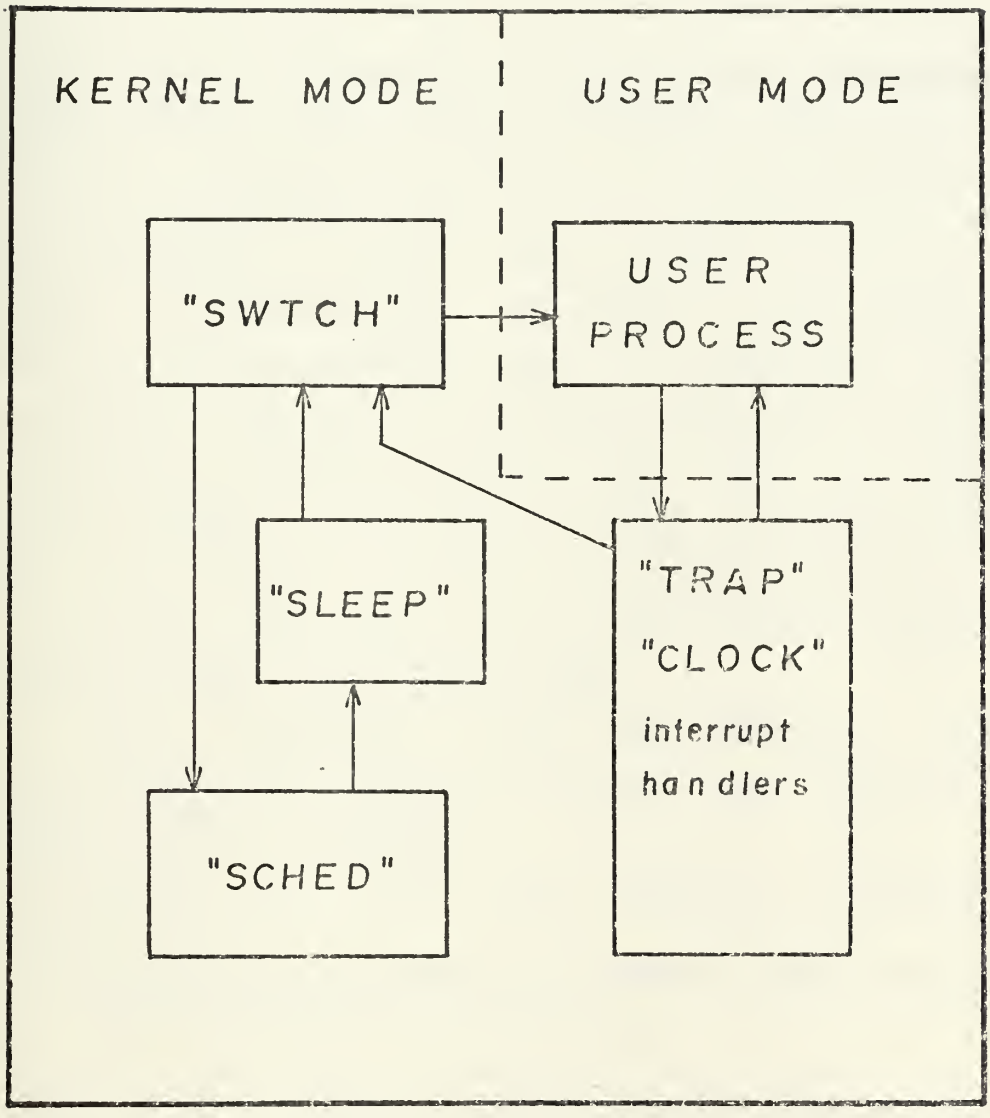


FIG. 3 - SYSTEM INTERACTION

V. MUNIX, THE MULTIPROCESSING UNIX

After the preliminaries of translating, documenting and studying UNIX, the production of MUNIX could begin. As it turned out, scheduling a process to run and starting it up (a task accomplished by the function "swtch") was a straight forward procedure in UNIX, greatly aided by the u vector attached to each process. No change to this procedure was needed in the new system as long as protection mechanisms were set up to ensure that processes could not be picked and run by the two processors at the same time. In fact, the overall problem of producing MUNIX could be stated more simply by breaking it into four separate subproblems:

* solve the problem of starting up both processors and initializing the system data bases;
* since processors could have different dedicated devices ensure that access to devices not connected to a processor is not attempted;
* define and rectify all possible synchronization conflicts;
* ensure that all critical resources were protected so that a race condition could not occur.

This chapter discusses these problems: where they were encountered, and how they were solved.

A. START UP AND INITIALIZATION

In violation of the stated goal of processor equality in MUNIX, it was decided to make processor zero, temporarily, the only processor capable of starting up the system. This was done because the swap disk and the remote terminals were only accessable from this processor (it would have been a difficult task to alter the old system so that either processor could start the system when the swap disk was not available to both processors). Later with a dual ported disk as the swap disk it will be much easier to allow the system start up from either processor. MUNIX, when desired, can still be brought up as a monoprocessing system with either processor.

System initiation remains essentially the same in the new system except that data and data areas for the second processor must be set up. Since both processors would be executing the system at the same time it was necessary to provide a u vector area for processor one. This area is exactly the same as processor zero's and is used when processor one is executing its system process; it is given the name process 2. Process 2 becomes the third permanent system process along with process 0 and "init". It is used by processor one when executing code in "swtch", "sleep" or "sched". It cannot be allowed to run all of "sched" until the dual ported RP disk allows both processors to access the swap file. When it is allowed to run "sched" process 2 will be equivalent to process 0.

"Mch.s" was modified so that, when processor zero starts up the system, it reserves a 1024 byte hole for the second processor's u vector. Processor one executes the same code for start up in "mch.s" and loads its Kernel memory management registers with the same values; the one exception is KDSA6 which it loads with its own u vector address. Both processors then branch out of the assembler code; processor zero to "main" and processor one to a new routine called

"start2".

The function "start2" is a C language routine that:

* initializes the user vector pointer, u.u_procp, to point to the process block assigned to process 2;
* idles the processor until processor zero has completed initialization;
* turns processor one's clock on and branches to "sched".

The function "main", in MUNIX, was expanded by giving it a few more initialization chores. The process block for process 2 is loaded with the proper values and all the system semaphores needed for the P and V locking of resources are set to one. Processor zero continues through "main" and branches tc "sched" with no further changes.

B. DEDICATED DEVICES

It became obvious during the initial coding of the system changes that there was, in several sections of code, a need to know which processor was executing. For instance, if an output to a terminal was about to be made, processor one could not be allowed to make it. In order to stop processor one from attempting such an output there had to be a way to discover that it was going to try. Since user processes would be initiating much of the I/O it could not be predicted before hand which processes a processor could run. The obvious solution was to allow each processor to execute any process until it came to something it could not do and then switch the process over to the other processor. In order to test whether a process could execute certain areas of code, which processor was running had to be

determined; once the processor was known a new system table could be accessed to see if this processor could continue. Making up the table was simple and the determination of which processor was runnning was made easy by a hardware change. The PS of the PDP 11/50 has some bits that are not used and one of these, bit eight, was permanently wired to indicate a zero in processor zero and a one in processor one. All that has to be done subsequent to this change to find out which processor is running a section of code is to check bit eight of the PS. MUNIX utilizes the new system function "whproc" to check this bit; it returns a zero or one as appropriate.

The ability to easily determine which processor is executing some of the code is necessary in many instances throughout MUNIX and is not isolated to cases of device inacessibility. All cases will be mentioned as the modifications to each system function are presented.

C. SOLVING RACE CONDITION PROBLEMS

In order to prevent a race condition each of the system data bases introduced in the last chapter had to be protected from dual access by the processors. In all cases, but one, Dijkstra's P and V operators were the means of securing these system resources. The u vectors were protected by synchronization of the two processors so that they could not simultaneously pick the same process to run. How this coordination was accomplished will be detailed in later sections of this chapter.

1. The P and V Operators

The data resources assigned semaphores and protected by the P and V operators are the process table, runout, bfreelist, cfreelist, coremap, and swapmap. When both processors attempt access to one of these data bases at the same time one is locked out by a Dijkstra type locking mechanism and therefore must wait until the resource is released.

This processor cannot contribute much to the accomplishment of system tasks while idle (it is limited to accepting those interrupts that are not masked out and running the associated interrupt routines). Because of this, the P and V operators were inserted only where absolutely necessary. For an efficient system as little waiting by processors as possible is desirable. Therefore, as little code as possible was put between the locking and unlocking operators.

In addition, it was discovered that the many process table searches conducted throughout UNIX could be divided into three distinct categories. A search of one category can be made simultaneously with one of another without producing a race condition. The first type looks through the table to find an unassigned block to be allocated to a newly created process, the second looks for processes already in the table in order to execute them, swap them in, or swap them out, and the third type looks for processes with certain attributes, depending on the particular search, and is virtually independent from all other searches. The third category needed no lock at all and the first two could be locked with different semaphores. If all process table searches and changes were locked with the same semaphore much more time would be wasted with processors at idle than is lost by MUNIX which takes advantage of the three different and noninterfering accesses to the process table.

How and where the system resources are secured from inappropriate access in MUNIX will be discussed in later sections of this chapter. The following is a list of the semaphores introduced into the system:

* vprocf - assigned to the process table and used for the first type of search introduced above;
* vprocs - assigned to the process table and used for the second type of search introduced above;
* vout - assigned to runout;
* vfreeb - assigned to the bfreelist resource;
* vfreec - assigned to the cfreelist resource;
* vmem - assigned to the coremap resource;
* vswap - assigned to the swapmap resource.

The system functions "poper" and "voper" were written as the MUNIX P and V operators. They follow the same algorithms introduced in chapter two with one addition. The interrupt priority level is set high when "poper" is invoked and reset during "voper" to the original value. This is done so that a processor cannot be interrupted after a P (and before a V) and somehow find its way back to invoke another P. If this were to occur a single processor could cause a deadlock.

D. GENERAL MODIFICATIONS

This section is devoted to the discussion of all the system changes not already presented in the previous sections of this chapter.

1. "sched"

After each processor completes initialization activities it branches to "sched". However, since processor one cannot be allowed an attempt at swapping it cannot be allowed to execute the code in this function. It is branched to the routine because the future aquisition of the dual ported disk will allow both processors to swap. Therefore a test was inserted at the beginning of "sched" to determine whether or not the executing processor can swap. To do this the new function "cntswp" is called and it consults a configuration table in which is indicated the processors having access to the swap file. If the current one has such access it returns a yes, otherwise it returns a no (this configuration table may be changed to reflect hardware configuration changes). If the test returns yes the rest of "sched" is executed; if no, the process (process 2 in the current hardware) is put to sleep on the new channel sloop and "swtch" is called. Because a "wakeup" is never done on sloop this process will never be chosen to run again and the associated processor will forever loop on "swtch" picking available processes to run.

Of the three searches in "sched" the first, which locks through the process table for a process deserving to be swapped in from the swap file, did not require a lock. When the dual ported swap disk is added to the system this search will require a lock and another semaphore can be used. The other two did and are locked on the semaphore vprocs. The last search, because it could select a ready process to swap out of core, was modified so that it wouldn't pick a ready process which was running on the other processor. This was done by providing an SGOING bit for p_flag that was turned on when a process was selected to run. A process with this bit on is not a candidate for swapping out in the MUNIX system.

The final change to "sched" is one that provides

synchronization between processors. Process 0 puts itself to sleep on runout if it has processes to swap in but no place in memory to put them and it is awakened, if asleep on runout, when there is the possibility of making room in core. If not synchronized these "sleep" and "wakeup" calls could be made in such a way that process 0 is put to sleep and never wakes up again. Without process 0 to swap processes in and out the system is deadlocked. Therefore the sleep call and the unblocking of process 0 in "wakeup" are locked on the semaphore vout.

2. "swtch"

Several important changes are found in the MUNIX version of "swtch". Since this is the function that picks the next process to be run by the system, most of the burden of processor synchronization must be carried here. In order to solve the coordination problem a few new process states were introduced. These states are indicated in the process block element p_flag by using more of its bits (p_flag was increased in size from eight bits to sixteen to accomodate the new states needed for MUNIX and other possible modifications to the system). The new states are summarized below.

* Execution - a process that is being executed will have two additional bits indicating its exact state. The SGOING bit shows that it is being executed and the SMD0 or the SMD1 bit is set to indicate which processor is executing it.

* Preference - when a process makes a system call and its processor cannot complete the call the SPREF bit is set in the function "trap".

* Ready - two additional bits have been added to indicate the exact ready state in MUNIX. SRUN and SLOAD must

be on for a process to be ready to be executed but the system also needs to know if a ready process must be run on a particular processor or not. SMD0 and SMD1 are used to show this necessity. If both bits are off it is an indication that either processor can run the process. SANYP is the flag that shows both the SMD0 and SMD1 bits on and its logical negative is used to test if they are both off and the process can be run on any processor.

* System call completion - when a system call has been completed another bit is used to so indicate. This flag, SANY, is used so that this process' processor preference can be reset to either processor in "swtch".

When "swtch" is now entered it turns off the SGOING flag of the process that was last being executed and checks to see if the SPREF flag is on. If it is, "swtch" sets the preference flag (SMD0 or SMD1), and turns off the SPREF flag. If it was not on, "swtch" checks to see if the SANY flag is on. If so, it turns the flag off and changes p_flag to indicate that the process may be run on either processor. If not, no change to p_flag is made. This sequence of events updates the status of the last process to run so that the next time it is picked by "swtch" it will reflect its correct status.

Once the above is accomplished the processor is ready to search for its next process. The search that is conducted is essentially the same as in UNIX except that it is locked on the semaphore vprocs and it checks the p_flag of each process for the SGOING and SMD0 or SMD1 bits. If the SGOING flag is on, the process is not available because the other processor is executing it. If the preference flag for the other processor is on then it will only be run on the other processor.

3. "trap"

In "trap" only two changes were incorporated. This function is invcked every time a process in User mode makes a system call. It is possible that the processor, since it may not be able to access the necessary device, cannot complete the system call. To prevent an access attempt by the wrong processor a test is made prior to the actual system call. This test checks a configuration table to determine whether or not the processor can service the original call. If it cannot a bit (SPREF) is set in p_flag to indicate that the process must be run on the other processor. The process is then put to sleep; it will be awakened after the next clock interrupt and will be executed starting after the "sleep" call in "trap". At the end of "trap" p_flag is set (SANY) to indicate that the current process can run on either processor.

4. "newproc"

In this function one search of the process table is made and it is locked on the semaphore vprocf. This search is of the first type introduced above. One other change delays the setting of p_stat to SRUN, for the process that is being created, until it has been completely readied for execution. Instead p_stat is set to SIDL at the beginning of the function and to SRUN at the very end. This is done so that no attempt to execute the new process is made until it is completely ready.

5. "fork"

"Fork" uses an identical process table search to that of "newproc" and it is therefore locked on the same semaphore, vprocf.

6. "clock"

In this function there are several lines of code that cause critically timed activities to be executed. A carriage return at a terminal is an example of one of those activities. Both processors cannot be allowed to execute this code because the timing would be upset and because, in a case such as mentioned above, the device that the activity was meant to service might not be accessable. Since, at the inception of MUNIX, processor one was unable to reach any of the devices serviced through this code it was branched around when processor one executed "clock".

If MUNIX allowed both processors to update system timing data when "clock" is run the data would not reflect actual system times. To ensure that these system times are accurate only one processor is allowed to keep them. The other processor only updates the timing data associated with the user processes it is running so that it is correctly kept.

A final change to "clock" is a check to see if the other processor has reached a point where it considers the system unstable. When this happens "panic" is usually called; it prints out the reason for the panic and increments a variable from zero to one and halts the processor. The variable indicates to the other processor, when it makes the check in "clock", that the system should be shutdown and it also goes to the function "panic" where it is halted.

VI. MUNIX, AN EVALUATION

In this chapter the existing limitations of MUNIX are briefly presented followed by some thoughts of future improvements to the system. Finally, and in conclusion, a general appraisal of the new system is made.

A. LIMITATIONS

The MUNIX multiprocessing operating system does not provide equality for both of its PDP 11/50 processors. The UNIBUS, dedicated to its own processor and inaccessable to the other, does not allow the system to share its task load equally, as desired. The equality problem will be solved as future hardware additions are made; chapter three introduced two of the possible improvements. The problem is actually one of bringing the second processor, processor one, into a more active role. MUNIX, as written, imposes even more stringent restrictions then required by the hardware limitations. With the exception of main memory and the dual ported swap device, processor one, when in the multiprocessing mode, is not allowed access to any of its peripheral devices. This limitation was imposed to simplify the task of producing a multiprocessor that worked. Now that MUNIX is working one of the first steps to making it better should be the expansion of processor one's capabilities.

B. IMPROVEMENTS

Providing processor one with the ability to utilize its own I/O devices is one way to give it a bigger role in the multiprocessing system. Another way to increase its influence on the system and to improve the efficiency of MUNIX is to schedule processes in a different manner. If the scheduling of processes is made so that I/O bound jobs are given preference on processor zero and compute bound jobs are given preference on processor one the result should be better usage of one's computing powers and faster system throughput. This modification to the system should not be difficult to make as the system now stands. If processor zero is made to pick first from those processes only it can execute, it would not be taking candidate processes away from processor one as often as it does in the present MUNIX. Processor one should inherit a more equal share of the load as a result of being given an improved chance at running those processes not requiring processor zero. Since in the foreseeable future of MUNIX processes that cannot be run on processor zero will be rare and those requiring this processor will abound the suggested modification should improve overall system efficiency.

Another improvement that can be made, but one that must wait for the addition of the dual ported swap disk, is to have processor one do most of the swapping of processes to and from the disk. If this were done processor zero could concentrate on other I/O requirements to the benefit of system efficiency. Again, this improvement is rather easy to implement. After start up and initialization of the system process 0 can be given the status of SIDL and process 2 can be allowed to remain active. Process 2 would then be scheduled as needed to do the swapping chores. Processor zero would have to perform those swaps envolving its private

memory and process 0 would be allowed to run in such cases.

Another thing that could be done to make the system more efficient would be the addition of an interrupt sent from one processor to the other. This interrupt would only be sent when one is at idle and the other has just done something to make a process ready to be executed. This would allow the idled processor to start the process right away instead of waiting for another interrupt to look for its next process.

One more improvement that could enhance system efficiency but one that would aid both UNIX and MUNIX would be an improvement in the way system tables are searched and set up. Every search in UNIX is done by looking at elements of a table one by one until the right one is found. If the tables were ordered so that the right element could be picked from the table with no search or a lower order search time the performance of the system would certainly be upgraded. Of course, an improvement in the speed of retrieving the right element from a table would most likely cost extra memory space and might not be worth the time savings. Some thought should be given to this as a possible modification to the system.

C. PERFORMANCE EVALUATION

MUNIX is already demonstrating improved performance over UNIX. Comprehensive tests have not been possible to date but two timing tests have been made with very promising results. When executing a moderate load of compute bound processes MUNIX demonstrated a twenty per cent improvement in time to completion. The execution of a package of several large and generally I/O bound compilations of C language programs was

estimated to be five per cent faster on the new system. These results are far from conclusive but they do indicate a very promising future for MUNIX.

LIST OF FIGURES

LIST OF REFERENCES

1. Enslow, E, Multiprocessor and Parallel Processing, p. 19-21, John Wiley, 1974.

2. Madnick S. and Donovan, Operating Systems , p. 35-593, Mac Graw-Hill, 1974.

3. Dijkstra, " The Structure of the "THE"-Multiprogramming System", ACM Communications, v. 5, p. 341-346, May 1968.

4. Ritchie D. and Thompson K., The UNIX Programmers Manual, p. I-1 to VIII-1, Bell Telephone Laboratories, 1974.

5. Ritchie, D., C Reference Manual, p. 1-24, Bell Telephcne Laboratories, 1974.

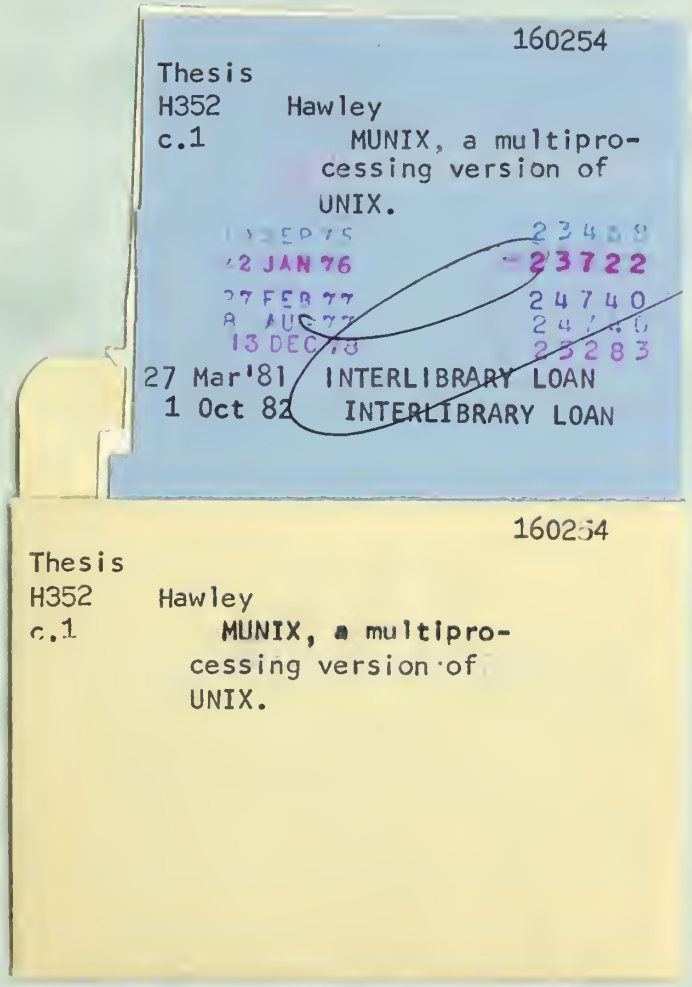
6. Ritchie D. and Thompson K., "The UNIX Time Sharing System", ACM Communications, v. 17, p. 365-375, July 1974.

7. DEC, PDP 11/45 Processor Handbook, p. 1-247, Digital Equipment Corporation, 1974-75.

8. DEC, Peripherals Handbook, p. 1-1/E-7, Digital Equipment Corporation, 1973-74.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 72 Computer Science Group Naval Postgraduate School Monterey, California 93940	1
4.	LT Belton E. Allen, Code 72An Computer Science Group Naval Postgraduate School Monterey, California 93940	1
5.	LTJG Gary M. Raetz, Code 72Rr Computer Science Group Naval Postgraduate School Monterey, California 93940	1
6.	LCDR J. A. Hawley, III, USN 1628 Guthrie St. Virginia Beach, Virginia 23462	1
7.	LCDR Walter E. Meyer, Brazilian Navy SMC 2211 Naval Postgraduate School Monterey, California 93940	1



160254
Thesis
H352 Hawley
c.1 MUNIX, a multipro-
cessing version of
UNIX.