UNIX on the PDP-11/23 and 11/34 Computers

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

During the past few years, the use of mini/micro computers in networks and small laboratory systems has steadily increased. Currently, the UNIX† operating system, used throughout the Bell System, is running primarily on DEC PDP-11/70s. With the advent of inexpensive computers similar in architecture to the PDP-11/70, in particular the PDP-11/23 microcomputer and the PDP-11/34 minicomputer, it became important that UNIX be available for these systems. The author set out, in June of 1978, to move UNIX from the PDP-11/70 to the PDP-11/34. The first version of UNIX on a PDP-11/34 with RL01 disk drives ran in July of 1978.

This paper describes architectural differences between the PDP-11/70 and the PDP-11/34 hardware,[1] their interaction with the UNIX operating system, and the changes the author implemented in that system to make it run on the PDP-11/34, along with some considerations for the future.

2. ARCHITECTURAL DIFFERENCES THAT AFFECT UNIX

There are many architectural differences between the PDP-11/70 and the PDP-11/34. For our purposes, it is important to understand only the differences that affect UNIX. The memory-management (MM) system, the availability of an instruction-backup register, the availability of additional register sets, the program-interrupt request register, and the set-priority-level instruction are all differences that affect the implementation of UNIX.

2.1 Memory Management

The PDP-11 family of computers is based upon a sixteen-bit virtual-address architecture. This architecture is implemented using pairs of MM registers. Each pair is composed of a MM-address register (containing the base physical address for mapping) and a MM-page-descriptor register (containing the length in bytes to be mapped and the direction of page expansion). The virtual address is mapped into a physical address by choosing a MM-address register and adding its contents times 0100 (octal) to the thirteen low-order bits of the virtual address. Thus, a pair of MM registers can map 8K bytes of virtual memory into 8K bytes of physical memory. The MM-address register is chosen by considering the current CPU mode (kernel, user, or supervisor), the type of memory reference (instruction space or data space), and the three high-order bits of the virtual address.

The PDP-11/70 has three CPU modes: *kernel* mode (K-mode), *user* mode (U-mode), and *supervisor* mode (S-mode). Each of these modes allows two types of reference: *instruction* space and *data* space.[2] Each type of reference has eight pairs of MM registers. Thus, the PDP-11/70 has sixteen pairs each of K-mode, U-mode, and S-mode MM registers. These 48 pairs of MM registers enable the PDP-11/70 to access 384K bytes of physical memory.

The PDP-11/34, on the other hand, has only two CPU modes: K-mode and U-mode. Each of these modes allows only one type of reference (instruction space), which has eight pairs of MM registers. Thus, the PDP-11/34 has eight pairs each of K-mode and U-mode MM registers.

---

† UNIX is a trademark of Bell Laboratories.

1. Unless explicitly stated otherwise, all references to the PDP-11/34 can be also applied to the PDP-11/23.
2. Instruction space is used for all instruction fetches, index words, absolute addresses, and immediate operands. Data space is used for all other references.

These 16 pairs of MM registers enable the PDP-11/34 to access 128K bytes of physical memory.

The absence of the data-space reference type on the PDP-11/34 requires all references to be mapped through the instruction space. This reduces the total amount of virtual memory per CPU mode from 128K bytes on the PDP-11/70 to 64K bytes on the PDP-11/34. This is by far the most serious restriction in moving the operating system and user programs from the PDP-11/70 to the PDP-11/34.

In many instances, the operating system moves data from the user to itself and vice versa. This requires the operating system to access physical memory not currently mapped by its K-mode MM registers. To accomplish this, the UNIX operating system must temporarily use MM registers belonging to another CPU mode. In the PDP-11/70, the S-mode is not used by UNIX. Therefore, the PDP-11/70 operating system uses its S-mode MM registers for this temporary addressability. Because the PDP-11/34 lacks a S-mode, the PDP-11/34 operating system must use its U-mode MM registers. This difference requires additional U-mode MM register saving and restoring in the PDP-11/34. This is a disadvantage both in CPU time and in the kernel space taken up by the code that saves these registers.

2.2 Instruction Backup

Temporary variables in a user program are stored in a last-in first-out data structure, which is called a *stack*. A user program in UNIX is run with an initial stack size of 768 bytes, which is expandable in 768 byte increments. The operating system will attempt to increase the stack size when it receives a MM trap from U-mode. After increasing the stack size, the program counter must be backed up and the instruction that caused the MM trap restarted. Unfortunately, some of the addressing modes of the PDP-11 have side effects that affect the general-purpose registers. These addressing modes are auto-increment/decrement of the general-purpose registers, and explicit references through the program counter. Thus, to restart an instruction, these side effects must be undone. On the PDP-11/70, there is a MM register, MMR1, that records any side effects on the general-purpose registers during execution of instructions. This register is used to reset the registers prior to restarting the instruction. The lack of this register for the PDP-11/34 forces a simulation of the source and destination addressing modes of the instruction that caused the MM trap. The lack of this register is very expensive in terms of kernel space taken up by the code that does this simulation.

2.3 Additional Set of General-Purpose Registers

The PDP-11/70 has a set of six additional general-purpose registers. Several critical UNIX routines run with interrupts disabled and utilize this set of registers. Because the PDP-11/34 lacks this set of registers, the PDP-11/34 UNIX must also use its registers for this purpose. This requires additional register saving and restoring, requiring more code in the kernel and more CPU time.

2.4 Program-Interrupt-Request Register

The PDP-11/70 has a program-interrupt-request register. This register is used to detect when the CPU is running at a priority level lower than or equal to a predefined priority level.

The UNIX operating system's algorithm for power-fail recovery depends on reaching a quiescent state[3] before processing the power-fail I/O recovery algorithm. This is accomplished by setting the program-interrupt-request register to interrupt when the CPU reaches priority level 1. The lack of this register in the PDP-11/34 forces its simulation when UNIX returns from interrupts. Because the UNIX operating system processes a great deal of interrupts, even a small amount of

3. Meaning that all interrupt processing started before the power-fail trap must be completed.

additional CPU time per interrupt is very costly.

2.5 Set-Priority-Level Instruction

Each time the UNIX operating system enters and exits critical pieces of kernel code, the CPU priority level is changed. The PDP-11/70 operating system uses the set-priority-level instruction (SPL). The lack of this instruction causes the PDP-11/34 to use a combination of bit-set and bit-clear instructions upon the processor status word (PSW). Because the UNIX operating system changes CPU priority levels a great deal, even a small amount of additional time per priority level change is also very costly.

3. IMPLEMENTATION OF PDP-11/34 UNIX

Moving UNIX from the PDP-11/70 to the PDP-11/34 required the author to write the machine-language assist functions for the PDP-11/34; these functions are written in the assembly language of the target computer to perform the following tasks:

- fault handling;
- memory management;
- speed-critical I/O and arithmetic operations;
- stack frame manipulation;
- hardware priority setting;
- register save and restore;
- machine-interrupt call to C procedure;

The PDP-11/34 machine-language assist functions are written with the same calling conventions as the PDP-11/70 machine-language assist functions and return the same values. This allows the same UNIX C language functions to be used on the PDP-11/70 and the PDP-11/34. In writing the PDP-11/34 machine-language assist functions, the author chose to partition the functions into separate files, as opposed to keeping the traditional *mch.s* file. This organization simplifies the management of the source code.

The module partitions and the algorithms used to handle the architectural differences described in Section 2 above are discussed in this section.

3.1 Module Names

The modules are subdivided by function. All *defines* are in *mch.h*, all the storage declarations are in *end.s*. The modules are listed below alphabetically, with a brief description of their function:

backup.s	attempt to back up an instruction that was only partially executed due to a MM trap from U-mode.
bufio.s	read and write of byte, integers, and longs in physical memory.
clist.s	*put* and *get* functions for the *cblock* structure.
copy.s	read and write large blocks of memory from virtual addresses to physical addresses, copy 64 bytes, clear 64 bytes, read and write from K-mode virtual addresses to U-mode virtual addresses.
csubr.s	save and restore registers that maintain the C stack frame.
cswitch.s	save and restore the user's registers and switch the operating system's idea of who is the currently-running user.
end.s	storage declarations.
fpp.s	save and restore the double-floating-point registers and status word.
math.s	long division, long remainder, minimum, and maximum functions.
mch.h	header file containing constants and definitions.
misc.s	process accounting and set-CPU-priority level.
power.s	save the state of the machine on a loss-of-power interrupt and restore the state of the machine on a resumption-of-power interrupt.

start.s	initialize MM registers, clear storage, and call main.
trap.s	all the fault handlers and the machine interrupt call to C procedure.
userio.s	read and write words and bytes in user's virtual addresses.

3.2 General Algorithms

The detailed description of the algorithms used to provide the functions necessary for the machine-language assist functions is divided into four categories. The algorithms manipulate the MM system, simulate the MMR1 register, simulate the program-interrupt-request register, and simulate the set-priority-level instruction.

3.2.1 Memory Management. In many instances, the operating system needs to access physical memory not currently mapped by the K-mode MM registers. The algorithm used to read and write physical memory utilizes the U-mode MM registers and the "move from/to previous instruction space" (MFPI/MTPI) instruction. To free the U-mode MM registers the old values must be saved on the kernel stack along with the current PSW; then the previous CPU mode (indicated by the PSW) must be set to U-mode. The long physical address is loaded into a pair of general-purpose registers and shifted ten bits to the left. The sixteen high-order bits of the result are the base physical address in *core clicks*[4] for the virtual address. The base address is loaded into a U-mode MM-address register and a 077406[5] (octal) is loaded into the corresponding MM-page-descriptor register. The sixteen high-order bits are cleared with the exception of bits 9-7, which indicate which U-mode MM register pair is used. The general-purpose registers are shifted six bits to the left. The sixteen high-order bits of the result are the virtual address for U-mode reads or writes. This virtual address is used with the MFPI/MTPI instructions, with a special case if the zero bit is set: this indicates a byte address not on a word boundary. Unfortunately, the MFPI/MTPI instructions only transfer from/to word boundaries. To MFPI this first byte, the function MFPIs a word from the virtual address whose bit zero is cleared, then returns the high-order byte of the word fetched. To MTPI this first byte, the function MFPIs a word from the virtual address whose bit zero is cleared, then places the first byte in the high-order byte of the word fetched, finally it MTPIs the word back to the same virtual address. When all transfers are completed, the old values of the memory management registers are restored, and then the old value of the PSW is restored.

3.2.2 Instruction Backup. In order to increase a user's stack space, the UNIX operating system must be able to restart a user's instruction. To restart an instruction, all addressing-mode side effects on general-purpose registers must be undone. The addressing modes that have such side effects are auto-increment/decrement and explicit references through the program counter. The algorithm used to correct the general-purpose registers starts by fetching the instruction to be restarted and deciding upon the number and type of its addressing modes. The number and type of addressing modes are calculated by decoding bits 15-12 for all instructions, bits 11-9 for instructions with bits 15-12 equal to 0000, 1000, or 1111 (binary), and bits 8-6 for instructions with bits 15-9 equal to 1111000 (binary). The possible side effects on the general-purpose registers are calculated for each addressing mode, assuming a MM trap did not occur. A MM trap will cause instructions to be partially executed, which means that not *all* the side effects necessarily occur. Thus, it must be determined which addressing mode caused the MM trap in order to determine which side effects must be undone. If the instruction has one addressing mode affecting a general-purpose register, then that is the addressing mode that caused the fault. The general-purpose register is corrected and the routine exits. If the instruction has two addressing modes affecting general-purpose registers, the source and the destination addressing must be checked to determine which one caused the fault. If the source addressing mode

4. A *core click* is defined as 64 bytes of memory.

5. Thereby allowing the reading and writing of 8K bytes.

caused the fault, only the source general-purpose register is corrected. If the destination addressing mode caused the fault, both general-purpose registers are corrected. This algorithm is not capable of correcting the general-purpose registers for instructions using the same general-purpose register for both source and destination addressing modes with side effects. Fortunately, the C compiler does not generate instructions using this combination of addressing modes.

3.2.3 Program-Interrupt-Request Register. The UNIX operating system requires the ability to detect when the CPU is running at a priority level equal to or lower than a level determined by the program-interrupt-request (PIR) register. To exactly simulate this register, the CPU priority should be examined before each change in priority level. This is expensive in terms of CPU time and, fortunately, unnecessary for UNIX; it is sufficient to check the priority level before each return from an interrupt. Just before the return-from-interrupt instruction is executed, the simulated PIR register is examined. If it is zero, the normal return from an interrupt sequence is followed. Otherwise, a register is counted down from 7, as the high-order byte of the simulated PIR register is shifted left. When a one is shifted out of the high-order byte of the PIR, the count-down register contains the priority level that should be used to compare against. The register is shifted left, moved to the low-order byte of the PIR, shifted another 4 bits left, and *or*'ed to the low-order byte of the PIR. The setting up of this byte is necessary to correctly simulate the PIR register of the PDP-11/70. The register now contains the desired priority level. Bits 7-5 of this register are compared to bits 7-5 of the PSW that was saved on the kernel stack when the interrupt occurred. If the saved PSW is greater then the desired priority level, the normal return from an interrupt sequence is followed. Otherwise, the contents of the program-interrupt-request vector (locations 0242 and 0240 octal) are pushed on the kernel stack and a return-from-interrupt instruction is executed to simulate the the PIR interrupt.

3.2.4 Set-Priority-Level Instruction. The UNIX operating system must be able to change CPU priority levels upon entering and exiting critical sections of code. To change priority levels, the PDP-11/34 must use a combination of bit-set and bit-clear instructions on the PSW. To change priority levels, the PSW must be brought to the high-priority level by bit-setting the PSW with a 0340 (octal) and then dropped down to the desired priority level by bit-clearing the unwanted priority bits. Changing to a high-priority level ensures that interrupts of a lower-priority level are not granted until the proper time. The only two exceptions are changing to priority-level 7, which is done by bit-setting the PSW with a 0340 (octal), and changing to priority-level 0, which is done by bit-clearing the PSW with a 0340 (octal).

4. INCREASING EFFECTIVE KERNEL SPACE

After the machine-language assist functions were written, the UNIX operating system ran on the PDP-11/34. It had enough room for an RL01 disk driver, a DZ11 terminal multiplexer, 30 processes, 9 system buffers, and 70 inodes. That PDP-11/34 UNIX operating system utilized less than 64K bytes of memory. However, this did not leave room for more device drivers, other desired kernel functions, or growth of system tables. Because the virtual-address space is limited by the PDP-11/34's MM hardware, only the effective-address space of the kernel may be increased. This section discusses the possible algorithms to increase the effective-address space and their interaction with the UNIX operating system.

4.1 Buffers

The single largest resource within the UNIX operating system is dedicated to the I/O buffer pool. Each entry consists of a buffer header of 26 bytes and an actual buffer of 512 bytes. Because the UNIX operating system does not always require direct addressability of its system buffers, the buffers may be moved out of kernel-address space. There are two possible algorithms for moving the buffers out of kernel-address space.

The first algorithm changes a K-mode MM register whenever addressability of a given buffer is required. This algorithm is fast. However, effective use of space requires the operating system to have 16 buffers, which fully utilized the address space of a MM register. When using 10 to 15 buffers, the amount of CPU time spent in searching the buffer pool is equal to the CPU time spent in re-doing the I/O. Therefore, this algorithm is not well suited for the small number of buffers usually found in the PDP-11/34 operating system. The author chose not to implement this algorithm, but to implement the following algorithm.

The second algorithm does not require a kernel MM register. It copies the contents of the buffers outside the kernel-address space to the kernel, using the machine-language assist functions. This algorithm is slower, but better suited to the number of buffers in the PDP-11/34. The impact of this algorithm on the PDP-11/34 operating system required the author to change buffer content references to function calls that copy bytes, integers, longs, and arbitrary numbers of bytes between physical addresses and virtual addresses, and place a copy of the current inode in the user block. For the sake of efficiency, a small number of kernel-addressable buffers is also maintained. These buffers are used as in-core copies of super-blocks and by some I/O devices.

4.2 User Block

The UNIX operating system controls the execution of a user process by keeping information about the state of the process in a structure called a user block. The PDP-11/70 operating system uses a *windowing* algorithm to address the user block. The *windowing* algorithm requires changing a kernel MM register to map a user block into its address space. Thus, the user block (which resides in memory locations preceding the corresponding process) is addressed as part of the operating system. The user block occupies 1K bytes of the 8K bytes available for mapping by a MM register. This windowing algorithm allows the operating system to quickly exchange user blocks by modifying a MM register. In expanding the effective-address space for the PDP-11/34 operating system, the author chose to use a slower algorithm that exchanges user blocks by copying them between kernel-address space and the locations preceding the user process. The advantage of this approach is that the MM register used by the PDP-11/70 version to address the 1K byte user block is now used to map 8K bytes of kernel-address space. This results in a gain of 7K bytes of kernel-address space. This algorithm required changes to the routines that exchange user blocks. These routines involve saving the current state of a process and resuming the previous state. When a user process is saved, the kernel-addressable user block is saved in the memory locations preceding the process. When a user process is resumed, the kernel-addressable user block is restored from the memory locations preceding the process. For the sake of efficiency *setjmp* and *longjmp* routines have replaced *save* and *resume* routines where only non-local *goto*s were required.

4.3 Other Possibilities

The author investigated many other possibilities to increase the effective-address space of the PDP-11/34 UNIX operating system. Following is a list of ideas that were considered, with the reasons for their rejection:

1. Temporary removal of inactive inodes from kernel-address space. This would be a saving of 76 bytes per removed inode. The amount of code to implement this idea had a break-even point of about 15 inodes. In the PDP-11/34 system, there are rarely 15 inactive allocated inodes.

2. Export of read-only super-blocks. The amount of code to implement the moving of the read-only super-blocks out of kernel-address space outweighs the advantage of their removal due to the small number of read-only file systems.

3. Pruning of the operating system. Space can be recovered by removing infrequently used operating system functions. The error logger, *ptrace*, and profiling routines could be removed. The author feels this form of space saving should only be used as a last resort,

because the resulting system is no longer a true UNIX system.

4.4 Under Consideration

The author is currently investigating other possibilities to increase the effective-address space of the PDP-11/34 UNIX operating system. Following is a list of ideas that are being considered:

1. Implementing device drivers as user programs. Infrequently used device drivers may be written as user programs. This, coupled with a system call that gives the user addressability of the I/O page, could be an effective saver of space for such device drivers as magnetic tape and line printers.

2. Using segmentation overlay within the operating system. Infrequently used functions within the operating system could be placed outside of kernel-address space. When these functions are needed, the contents of the MM registers would be modified to place these segments in kernel-address space. This would be done (invisibly to both the operating system and to user programs) by modifying the loader and adding machine-language assist functions. The modifications would insert, at subroutine calls, code for invoking machine-language assist functions that would, in turn, modify appropriately the contents of the MM registers.

ACKNOWLEDGEMENT

I would like to thank Larry A. Wehr for advice that lead to the first version of UNIX for the PDP-11/34. I would like to especially thank Sharon Murrel and James Goodnow, II for being patient users of my many experimental operating systems.

REFERENCES

[1] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1905-29.

[2] Thompson, K., UNIX Time-Sharing System: UNIX Implementation, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1931-46.

January 1981