# A UNIX(TM) Operating System for the DEC VAX-11/780 Computer*

*Thomas B. London*

*John F. Reiser*

*ABSTRACT*

A UNIX operating system together with a complete user environment has been implemented on the VAX-11/780 computer manufactured by Digital Equipment Corporation. The VAX-11/780 system provides 32-bit addresses and data. It uses the same input/output devices as the PDP-11 family and is controlled through a console computer which can be remotely accessed. Additionally, the VAX-11/780 is priced nearly the same as a PDP-11/70 and executes most programs somewhat faster than a PDP-11/70.

This memorandum describes the VAX-11/780 hardware and the UNIX operating system and the C programming language software implementation, records some observations on program portability, and speculates ways in which the operating system overhead can be significantly reduced. The authors conclude that the VAX-11/780 provides an excellent hardware environment for running UNIX and C software.

## Introduction

The VAX-11/780 [1] is a new, general-purpose, stored-program electronic digital computer manufactured by Digital Equipment Corporation. At minicomputer prices it provides addresses and data which are 32 bits wide; the traditional minicomputer address space bound of 64K is gone. This memorandum describes the VAX-11/780 and the implementation of a UNIX operating system and complete user environment for it. Section 2 contains an overview suitable for general consumption; details normally of interest only to devotees of computer system architecture appear in Section 3. The authors comment on software portability in Section 4.

## Overview

**Environment.** A user of UNIX and C software on the PDP-11 will find that the VAX-11/780 provides a very similar environment. There are no apparent differences in the command language or the vast majority of programs which are customarily invoked directly from the shell. A casual user probably will not be able to distinguish the hardware, except by issuing the command "who am i" (which identifies the hardware and the current user) or by noting that one of the columns printed by the process status command ps is in hexadecimal rather than octal. The C language programmer will find that **int**, **long**, and pointer data types all occupy 4 bytes (a **short** still occupies 2 bytes), and that a **long** has its two halves stored in a different order on the PDP-11 than on the VAX-11. Characters still suffer sign extension when converted to longer integer types, but one may use the declaration **unsigned char.**

**Hardware.** The VAX-11 is a follow-on computer to the PDP-11. The architecture seen by the user-mode assembly-language programmer of a VAX-11 is "culturally compatible" with the PDP-11. Specific details differ, but a programmer familiar with the PDP-11 can quickly understand the differences. The

*This is a revived version of an internal Bell Labs memo by Reiser and London, dated July 7, 1978. This rendition was generated by OCR from a scanned copy of the original memo, with the result edited by Dennis Ritchie. Many flaws and typos in the OCR were found by Peter Jeremy, Aharon Robbins, and Naoki Hamada. Others no doubt remain, for which I am responsible.

--DMR

VAX-11 provides UNIBUS and MASSBUS interfaces and uses the same input/output peripheral devices as a PDP-11.

Significant new features of the VAX-1l include an extended virtual address space, intelligent console, and dramatically improved physical packaging. The address space of a process is divided into a few gigantic segments. Each segment is further divided into a large number of small pages. Sufficient hardware exists to make demand paging a viable memory management strategy. All console functions are handled by an LSI-11 microcomputer through a standard ASCII terminal. The terminal may be remotely located from the processor and can still halt, boot, or diagnose the VAX-11. The mechanical and physical design of the VAX-11/780 is well done. The processor contains no sliding drawers or moving cables. All parts are easily accessible for servicing. Adequate airflow is maintained even under maintenance conditions.

**Configuration.** The actual configuration purchased by Department 1353 is:

VAX-11/780 cpu
0.5 megabytes memory with battery backup
floating-point accelerator
12Kbyte user-writeable control store
UNIBUS adaptor with DZ11 (8 RS-232C lines)
MASSBUS adaptor with TE16 tape drive (800/1600 bpi)
MASSBUS adaptor with two RP06 disk spindles (176M bytes per spindle)
additional BA11KE UNIBUS box

The list price of the above configuration in February 1978 was $241,255; the price including a DEC discount to a Bell Labs purchaser was $200,242.

**Software.** We have implemented a UNIX operating system [2] and complete user software environment on the VAX-11/780. The operating system is Research version 7 as of April 15, 1978. The environment includes the Bourne shell, C compiler, code improver *c2*, assembler, loader, debugger, standard I/O subroutine library **libS**, C subroutine library **libc**, source code control system SCCS, *nroff/troff*, and more than 130 commands. Maintenance programs for file system checking, bootstrapping, and physical disk pack handling have also been implemented.

We began with the C language code of Research version 7 of the UNIX operating system, and a PDP-11/45 running UNIX as a bootstrap machine. Creating a C compiler which produced VAX-11 native-mode assembly code was the first task. The code generator portion of the portable C compiler was rewritten to do this. An assembler and loader, based on similar code for the Interdata 8/32, completed the basic support software. Existing PDP-11/70 device drivers for disk, tape, and terminal communication lines were adapted to the VAX-11/780. Assembly language interfaces (trap handlers, hardware initialization, etc.) were completely rewritten. We then created magnetic tapes in the proper format for an initial file system and for deadstart load, and physically carried these tapes from the PDP-11/45 to the VAX-11/780.

Work on the C compiler began in mid-December 1977. The hardware arrived on March 3. We held a party on May 19 to celebrate successful multiuser operation of the system.

**Performance.** Identical documents were formatted by *nroff* on our VAX-11/780 and on a PDP-11/70 running Research version 7 UNIX; both systems used RP06 disks. Identical C programs were compiled and assembled on the VAX-11/780 and on the PDP-11/70. As reported by the *time* command, the results (converted to seconds) were:

nroff -ms -e -T450-12 ios.r >/dev/null

| | real | user | sys |
|---|---|---|---|
| VAX-11/780 | 47.0 | 28.6 | 8.7 |
| PDP-11/70 | 54.0 | 36.9 | 7.9 |

cc -c -0 pftn.c

| | real | user | sys |
|---|---|---|---|
| PDP-11/70 (Ritchie compiler) | 86.0 | 43.5 | 11.8 |
| VAX-11/780 (portable compiler) | 82.0 | 64.0 | 10.5 |

| | | | |
|---|---|---|---|
| PDP-11/70 (portable compiler) for Interdata 8/32) | 153.0 | 114.6 | 16.6 |

From the statistics on *nroff* one should conclude that, based on user-mode CPU time, the VAX-11/780 can execute the code produced by the VAX-11 C compiler approximately 22% faster than the PDP-11/70 an execute the code produced by the PDP-11 C compiler. This is a measure of the combined power of the hardware and efficiency of the code generated by the compiler. Except as an upper limit, the figures give no indication as to the throughput, response time, or efficiency of the operating system. The differences in real time and system time between the VAX-11/780 and the PDP-11/70 are not significant.

The times given for compilation of the file *pftn.c* are an attempt at a "black box" comparison of apples and oranges. The black box is any program (compiler) which takes C language input and produces executable instructions. The black-box comparison is that the current VAX-11 C compiler running on the VAX-11/780 and compiling code for the VAX-11 requires 49% more user-mode CPU time than the current PDP-11 C compiler running on the PDP-11/70 and compiling code for the PDP-11. The apples and oranges aspect arises because the two compilers, while equivalent from the black box viewpoint, are (on the inside) totally different pieces of software. The PDP-11 compiler is a production compiler written by D. M. Ritchie; the VAX-11 compiler is a portable compiler based on work by S. C. Johnson. The figures for the portable compiler running on the PDP-11/70 and compiling for the Interdata 8/32 are included for those who wish to compare two portable compilers. We have no VAX-11 equivalent to the Ritchie compiler, and thus cannot run the tests which would enable comparison of two production compilers.

The loaded size in bytes of the operating system and seven other programs appears in Table 1. One should note the general similarity between the text (instructions) sizes on the PDP-11 and on the VAX-11, and between the bss (uninitialized data) sizes on the VAX-11 and on the Interdata 8/32. The particular PDP-11 UNIX system chosen has several more input/output device drivers and experimental multiplexing software not in the VAX-11 system, which accounts for its larger text size. If many global integer variables (or large arrays) are used, there is a tendency for the data and bss portions to double in size when going from a PDP-11 to a VAX-11 or an Interdata 8/32 because an **int** occupies two bytes on the PDP-11 and four bytes on the other machines. However, character arrays occupy the same amount of space on all machines. An unusually large number of references to global variables in the *nroff* program accounts for its increase in text size on the VAX-11 compared with the PDP-11. A program can be written to automatically change the addressing modes used in the VAX-11 code so that most references to global data become shorter than at present, but this has not been done.

**Evaluation.** We believe that the VAX-11/780 provides an excellent hardware environment for running UNIX and C software. With the software in its current state, we view the system as operationally equivalent to a PDP-11/70 running UNIX software, except that the 64K limit on process address space is gone and programs run faster. We believe that the advanced memory management and user/system communication capabilities of the VAX-11/780 offer an opportunity to construct future UNIX-like systems with substantially higher throughput than provided by today's UNIX on a PDP-11/70.

## Details

### Hardware

Four main subsystems -- the central processor, console, main memory, and input/output -- constitute the VAX-11/780 computer system. The central processor, memory, and input/output subsystems are connected by the Synchronous Backplane Interconnect (SBI), an internal synchronous bus with a maximum data throughput of 13.3 megabytes per second. The SBI deals in physical addresses which are 30 bits wide. Half of the SBI address space is reserved for memory addresses, and half for input/output device registers. Arbitration for bus cycles on the SBI is distributed; each subsystem decides if it will use the next bus cycle.

The central processor is a microprogrammed 32-bit general-register computer. The architecture seen by the user-mode assembly-language programmer is "culturally compatible" with the PDP-11; an expert programmer familiar with the PDP-11 can learn and understand the differences in one day or less. The processor handles binary integers of 8, 16, and 32 bits; single precision (32 bit) and double precision (64 bit) floating-point numbers; character strings up to 65535 bytes long; bit fields up to 32 bits wide; and IBM-style packed decimal strings up to 31 digits long. Bit fields have no alignment restrictions whatsoever, all other data types require alignment only to a byte (8 bit) boundary. The central processor provides sixteen 32-bit general registers. Register 15 is the program counter **pc**. Software operating in one of the privileged access modes (see below) must use register 14 as a stack pointer **sp**. The instructions which implement high-level procedure call and return (**pushl, calls, callg, ret**) assume a convention about the use of **sp**, register 13 (**fp**, the frame pointer) and register 12 (**ap**, the argument pointer). The instructions which handle character and packed decimal strings use registers 0 through 5 to hold pointers and counters, so as to be interruptible. Floating-point operations may use the general registers; there are no separate floating-point registers. Instructions take from zero to six operands. The operation code occupies one byte and is followed by the operands, which require from one to nine bytes each. Nine addressing modes (including all the PDP-11 modes except *-(r)) are allowed, and the addressing modes are independent of the operation code. When the central processor is executing in the context of a process, there are four access privilege modes (user, supervisor, executive, kernel), each with its own stack pointer; software which desires a per-process kernel stack is easy to implement. A fifth stack pointer is used when executing in a special system-wide interrupt context. The VAX-11/780 processor includes an eight kilobyte, two-way set associative, write-through, memory data cache; an eight-byte instruction stream buffer; and a 128-address virtual address translation buffer. Most of the processor is implemented in Schottky TTL MSI logic. A programmable realtime clock and a time-of-year clock (battery operated during loss of line voltage) are standard equipment. Options include a hardwired floating-point accelerator and user-writeable control store.

The console subsystem consists of an LSI-11 computer, local memory, floppy disk, DECwriter terminal, and remote-access communications port. The console is connected directly to the central processor and performs all the functions of a conventional "lights and switches" front panel. The floppy disk serves as the initial bootstrap device for normal operation and holds special microcode for diagnostic operation. When activated by a key switch on the central processor, the remote-access port becomes the console. A terminal connected through the remote-access port can halt the central processor, boot it, diagnose it, etc.

The virtual address space of a process running on the VAX-11/780 consists of 2**32 8-bit bytes. The two high-order bits of a 32-bit address determine one of four segments. Two of these segments are system segments common to the address space of all processes. One of the system segments is reserved for future use. The other two segments are separately defined for each process and are automatically managed by the context switching instructions. One of the per-process segments is designed for a stack which grows towards lower-numbered memory addresses. Segments are divided into pages of 512 bytes. Memory mapping hardware translates virtual addresses into physical addresses using page tables. A page table contains one four-byte entry for each page mapped; the entry contains a valid bit, a four-bit field which encodes access privileges, a modify bit, and the physical page-frame number where the page is mapped. (There is no reference bit which is maintained by hardware!) A base register and a limit register describe the page table of each segment. The base register of a per-process segment contains a virtual address within the system segment; the base register for the system segment contains a physical memory address. The VAX-11/780 central processor contains a virtual address translation buffer holding 128 virtual address-page frame number pairs which eliminates the need for extra memory references during address translation for (typically) 98% of all memory references. The memory is implemented using MOS semiconductor RAMs with an error correcting code which corrects all single-bit errors and detects all double-bit errors and 70% of all greater-than-double bit errors. A memory controller can handle 8 memory boards; using 4K chips each board can hold 128K bytes. There can be two memory controllers, thus the maximum amount of physical memory is currently 2 megabytes. When 16K chips are used (forecasted for late 1978), each board will hold 512K, and physical memory can be 8 megabytes. There is a battery backup option for maintaining data in the event of a power failure. Each optional battery will maintain l megabyte for 10 minutes.

The input/output subsystem consists of UNIBUS adaptors and MASSBUS adaptors. A UNIBUS adaptor (UBA) is an interface between a standard UNIBUS and the SBI. The UBA does the bus arbitration and everything else necessary to administer the UNIBUS. It also contains a set of registers for mapping UNIBUS addresses to and from SBI addresses. The maximum throughput on a UBA is 1.5 megabytes per second. A MASSBUS adaptor (MBA) is an interface between the SBI and MASSBUS devices (RP06 disk, TE16 tape, etc.). An MBA would be more properly called an RH-780 controller, analogous to the RH-11 controller on a PDP-11/70 MASSBUS; only one unit may transfer data at a time, although several similar units connected to the same MBA can execute control functions simultaneously. The MBA contains the device control registers normally found in an RH controller. The registers lie in the I/O section of SBI addresses. An MBA also contains a set of mapping registers which translate device byte addresses to and from SBI addresses. The maximum throughput on a MBA is 2.0 megabytes per second. The published limits are 1 UBA and 4 MBAs per system. Theoretically one could have any number of either kind as long as the sum of the number of central processors, memory controllers, MBAs, and twice the number of UBAs were 15 or less, since the SBI has 15 "ports".

The physical packaging of the system has been dramatically improved compared with the PDP-11. The VAX 11/780 processor cabinet contains no drawers or moving cables. The SBI is fixed and rigid. Three one-third horsepower squirrel-cage blowers provide sufficient air flow -- even while servicing the CPU. Any logic card, power supply, or blower can be replaced within twenty minutes by one person using only a screwdriver. The CPU stands 1.53m x 1.17m x 0.77m (HWD); cabinets housing the CPU, UNIBUS devices, and tape drive are usually bolted together to form a single unit 1.53m x 2.51m x 0.77m. Our configuration (see section 2) weighs 3452 pounds and requires 42050 BTU/hr cooling.

**C Compiler**

A VAX-11 "native mode" C compiler was constructed using S. C. Johnson's portable compiler as a base. After one month, a reasonable version began to evolve: it produced code which was good enough to exercise the assembler, loader, and debugger (on the bootstrap PDP 11/45). This initial version did not make use of VAX-11 indexed addressing (which does single-level array subscripting including appropriate index shifts), bit field instructions, or autoincrement/decrement addressing. It contained its share of bugs, particularly since the hardware had not arrived and could not be used to actually run the generated code.

Substantial effort has been subsequently directed towards improving all aspects of the compiler: bugs have been corrected, routines have been made to execute more efficiently, and the quality of the generated code has been improved. All addressing modes are supported, bit field instructions are used for programmer-defined bit fields, and autoincrement and autodecrement addressing as well as three-address instructions are used.

Overall, our experience with the compiler has been very favorable. When the VAX 11/780 was delivered, the compiler worked well enough to compile itself, the UNIX kernel, and many user-level commands. In fact, since the delivery of the machine, only about a half-dozen serious bugs have been detected. Additionally, the framework of the compiler has proven itself to be flexible: a compiler for the Interdata 8/32 was transformed into a compiler for the VAX-11/780, some improvements and extensions were easily added, and, in general, a quickly evolving compiler has remained stable and productive. The authors feel that, with a few extensions to the model of the compiler and a certain amount of tuning, the current VAX-11 compiler could easily remain as the production VAX-11 compiler.

There are still some deficiencies in the current version of the compiler, as well as in the basic "product" itself. The compiler is slow and quite large; see the statistics in section 2 and Table 1. Some of the blame for the size and lethargy of the first pass can be attributed to the use of *lex* for the scanner and *yacc* for the parser, and to the use of ASCII to communicate information between passes. Both *lex* and *yacc* produce large routines: the scanner is 17K bytes in length (over 4.5K bytes of instructions), and the parser is 16K bytes long (over 5.5K bytes of instructions). On the average, the first pass spends 20% of its time in the lexical scanner *yylook*, and 9% of its time in the parser *yyparse.*

Using ASCII to communicate between the two paves causes an additional speed penalty for character conversion. On typical programs, the first pass (parser) spends roughly 30% of its time performing output services (i.e., calls to *_doprnt* (18%), *_strout* (8%), and *printf* (4%), while the second pass (code generator) spends roughly 21% of its time reading it back in (i.e., calls to *read* (18%) and *rdin* (3%). (Additionally, the routine used to convert from ASCII to binary contained a bug which caused "-2147483648" (which is -(2**31) ) to be read as zero on our PDP-11/45.)

The above problems are not inherent to the compiler model. To speedup compilation, the scanner can be hand-coded (as in the standard PDP-11 compiler), and the interpass data can be formatted in binary (or the two passes can be combined). With these simple modifications (some are already in progress), it should be possible to produce a compiler almost twice as fast as the current one.

Two features of the VAX-11 architecture -- three-address instructions and indexed addressing mode -- were difficult to model within the basic structure of the compiler. The full implementation of three-address instructions proved to be so difficult that it was not really attempted. Instead, *c2,* the assembly language code improver, tries to merge several instructions into an appropriate three-address instruction. For example, the statement *a = b + c* compiles

```
addl3 b,c,r0
movl r0,a
```

which the improver can change to:

```
addl3 b,c,a
```

for a savings of three bytes and over 400 nanoseconds. However, c2 will not always succeed in this shortening. It cannot tell the difference between

```
a = b + c
return;
```

and

```
return(a = b + c );
```

since register r0 must be considered "live" (i.e., contains a value which may be required later) across the return statement.

The VAX-11 has six indexed addressing modes which yield the address of an element of a one-dimensional array of a base type (**char**, **short**, **int**, **long**, pointer, **float**, or **double**). The statement

```
a[i] = b[j] * c[k];
```

where *i*, *j*, and *k* are declared **register int** and *a*, *b*, and *c* are **double** arrays (either external or local) can be compiled into the single instruction:

```
muld3 b[j],c[k],a[i]
```

Although the index specifier (e.g. *i* in the above example) must be a register, the base address specifier can be any addressing mode except register, literal, or another indexed mode. For example, the C-language constructs *a[i] , (*p)[i]*, *(--p)[i]*, *(p++)[i]* and *(*p++)[i]* (or their equivalents **(a+i)*, **(*p+i)*, **(--p+i)*, **(p++ +i)*, and **(*p++ +i)*, respectively) all can be done with a single VAX-11 address (where *a* is an array of base type, *p* is a pointer to the same type, and *i* is of type *register int*). It is usually difficult to recognize or conveniently represent such constructs (e.g., *(*p++)[i]* is fun), or generate the possible cases (e.g., *a[i]* where *a* is not readily addressable).

The fact that the code generator can easily recognize only expression trees of height one (two if OREG and UNARY MUL nodes are taken into account) causes substantial difficulty in making use of indexed mode, three address instructions, and indirect addressing. Expression trees of non-trivial height occur not infrequently (e.g. as a worst case, the statement

```
a = b + (*p++)[i];
```

has an expression tree of height six, but can be compiled into the single instruction

```
addl3 b,*(p)+[i],a
```

if *p* and *i* are **register** variables). The complexity of the code generator is raised by forcing the compression of subtrees into single nodes which are then treated with special checks, special code, etc.

The size and alignment attributes of data objects are logically independent, even though previous hardware architectures (IBM 360, PDP-11, Interdata 8/32, ...) have imposed alignment restrictions based on size. The VAX 11/780 has no such restrictions, although programs run faster with data aligned on natural boundaries. The C language has little notion of alignment; because of run-time penalties, the VAX-11 C compiler aligns all the basic data types on address boundaries which are a multiple of **sizeof** the basic type. Due to questions about alignment, both the language and the compiler have difficulty with the declaration *char c:10;*.

The decision to naturally align most data items has undesirable side effects which cannot be ignored.

Consider the structure declaration

```
struct foo {
        char c;
        float f;
} bar;
```

On the PDP-11, **sizeof** ( *foo* ) is 6 bytes while on the VAX-11, **sizeof** *(foo)* is currently 8 bytes (the offset of *f* within bar is 2 and 4 respectively). **sizeof** *(foo)* could be 5 bytes in each case. Although both machines use the same data formats for chars and floats, the differing alignment imposed by the the VAX-11 C compiler means that the two machines cannot speak directly to one another using media which record structures containing binary information. Since alignment is important, we feel that it ought to be specifiable in the C language.

**Operating system conversion**

A UNIX system running on a PDP-11/45 was used as the base for transporting software to the VAX-11/780. The software itself originated with the code produced by members of Center 127, Computing Science Research, for the Interdata 8/32. Programs were cross compiled, assembled, loaded, and put on magnetic tape in *tp* format; absolute bit-string files were put on tape in *dd* format. Tapes were then carried across the room to the VAX-11/780. An absolute tape boot (in machine language), *tp* boot and primary disk boot (in assembly language), secondary disk boot (in C), and stand-alone utilities (disk formatter, disk verifier, tape-to-disk, disk-to-tape, disk-to-disk, and disk-to-console, all in C) were then used to bring up the system.

Establishing an initial file system on the disk took longer than expected. The PDP-11/45 was running USG issue 3 of the UNIX operating system with a "16-bit" file system and the VAX-11/780 was to have a Research version 7 "32-bit" file system. Also, C-language code on the VAX-11 expects the bytes of a 32-bit integer to be stored in a different order than C-language code on the PDP-11. We swallowed these two red herrings hard, and suffered. We now know that the proper way to create an initial file system is to modify the program *mkfs* so that its output (on the bootstrap machine) is a file containing the proper bits, put that file on tape, and use the tape-to-disk utility on the target machine.

Mapping the software architecture of the UNIX operating system onto the hardware architecture of the VAX-11 required a number of decisions. Commentary on these decisions follows. The SCB (system context base) processor register contains a page-aligned physical memory address which is the base of the hardware interrupt vector. The UNIX system puts this vector at physical memory address zero. Operating system code, data, kernel stacks, and interrupt stack occupy the VAX-11/780 system segment (virtual addresses 80000000 to bffffff). User code and data are loaded into segment zero (0 to 3fffffff) and the user stack is initialized in segment one (7fffffff to 4000000). User processes pass arguments to system service code using the ordinary *calls* subroutine calling sequence. The **chmk** instruction is then used to gain kernel privileges. The **chmk** instruction switches the stack pointer **sp** from the user stack to the kernel stack, but does not change the argument pointer **ap** or the frame pointer **fp**. The kernel uses the value in **sp** to copy the arguments into *u.u_arg*. The VAX-11 hardware allows the values to be directly addressed, but the kernel software requires the copy.

The *u area* is a per-process data structure in which the operating system keeps swappable information about a process. The kernel virtual address of the *u area* must be a constant across all processes. The PDP-11 implementation puts the *u area* at kernel address 0160000; when process switching occurs the *u area* is switched by changing a kernel data space segmentation register. Since the operating system can address user memory on a VAX-11, the *u area* could be placed in (protected) user memory, say at address 0 or at 7fffe000. However, it was desirable for the first implementation to make the page tables for user segments part of the *u area*, which creates timing problems unless the *u area* lies in system space. The base of the *u area* was assigned kernel virtual address 80020000. When process switching occurs, the *u area* is changed by changing the system-space page table and invalidating the page-table translation cache for the appropriate pages.

Since the operating system can directly address the memory of the current user process, the procedures *fubyte, subyte, fuword,* etc., are unnecessary and could be made into macros which would merely do the appropriate load or store. However, these procedures (along with *copyin* and *copyout*) were kept to ensure that each access to user space is valid.

A VAX-11/780 internal processor register called the PCB (process context base) points to an area in which the VAX-11/780 saves the hardware state of the machine (96 bytes) when switching context. This save area was put in the *u area* as *u_rsav*.

The implementation of context switching required major effort. The VAX-11 has two very nice instructions (**svpctx**, save process context; and **ldpctx**, load process context) which facilitate context switching. Unfortunately, they do not implement the mechanism which the UNIX system expects. (The mechanism used by UNIX is so dispersed and intricately detailed that it is hard to imagine any hardware which implements it directly.) The temptation to drastically change the UNIX code has been resisted so far. The *savu/retu/aretu* tar pit was VAXinated, but it took more than a week. The newer *save/restore* primitive does make the C-language code prettier, but the assembly-language side (at least for the VAX-11) is just as dirty as ever. The UNIX context switching mechanism requires three state save areas, *u.u_rsav*, *u.u_ssav*, and *u.u_qsav* because the same mechanism is also used for abnormal returns. The VAX-11 context-switching instructions use only a single state save area. To make use of the VAX-11 instructions, the software simulates a great deal of microcode and bastardizes call frames in a most ugly manner. Context switching is certainly high on the list of things to rewrite in the second implementation (even for the PDP-11!).

The procedures *sureg* and *estabur* were also tricky to implement. They were designed with the assumption that only a small number (16 or fewer) of registers would be needed to map the address space of a user process, while on the VAX-11 a 32K process requires 64 page table entries. Furthermore, the memory map of a process is diddled in tricky ways, particularly in *expand* and *getxfile.*

Handling DMA I/O hardware was the other major implementation bottleneck. The UBA and MBA mapping registers contain physical memory page numbers, and physical addresses are hard to handle. It is not pleasant to deal with the hardware which implements the mapping registers. If an I/O transfer is in progress then the mapping registers may be neither read nor written; this applies even to registers which would not be used by the transfer. As a result, the map for the next I/O operation cannot be set up during the current I/O operation. Furthermore, a single transfer is limited to 64K bytes because the byte counter is only 16 bits wide. Thus swapping a process to the disk can require multiple I/O operations. The solution to these problems involved permanently reserving the last 129 registers in each map to service both swap and physical I/O operations. The remaining map registers are available to map the system buffers, and are loaded at system initialization time. Disk ECC error correction is currently done only for I/O involving the system buffers. Disk errors on raw I/O cause process termination; the swap area on disk had better be error-free.

Like the UNIX system for the PDP-11, the current implementation for the VAX-11/780 maintains each process in contiguous physical memory and swaps processes to disk when there is not enough physical memory to contain them all. Reducing external memory fragmentation to zero by utilizing the VAX-11/780 memory mapping hardware for scatter loading is high on the list of things to do in the second implementation pass. To simplify kernel memory allocation, the size of the user-segment memory map is an assembly parameter which currently allows three pages of page table or 192K bytes total for text, data, and stack. This also deserves to be rewritten, both to allow varying process size, and to allow processes larger than physical memory through demand paging. Dynamic page table size would mean dynamic *u area* size if the page table remained part of the *u area.*

The code in *sendsig* for sending a signal to a process involves a tedious simulation of the **calls** instruction due to the problem of "inward return" across privilege modes upon termination of the routine which handles the signal. Making a portion of the kernel code readable by a user-mode process would simplify *sendsig*. Motivated by a problem with the Bourne shell, the signal number is passed as a parameter to the signalled routine.

Interprocess communication via signals (*signal* and *kill*) uses the low-order bit of a machine address for something other than addressing. This implies that a procedure which handles signals must start on an even byte boundary, which means that every procedure must start on an even byte boundary. The C compiler thus issues a pseudo-op to the assembler to align the beginning of each procedure. This can waste memory on a VAX-11. It also imposes a nontrivial requirement on the assembler, since if the resolution of conditional jump instructions can change the parity of the length of a procedure then the alignment directive must also be handled like a conditional jump. In hindsight, it would have been better if a distinct value (say +1 or -1) were used for *ignore,* rather than multiplexing the bottom bit.

The VAX-11/780 provides a (non-maskable) trap for integer division by zero. The system would like to turn this into a signal to the process. A similar situation exists for subscript range trap. Integer overflow, floating overflow, floating underflow, and reserved operand also need signal numbers. Perhaps only one "error" signal is needed with some other means for determining the true fault. The whole business of interrupts, signals, asynchronous I/O, and the use of the hardware AST mechanism deserves more attention.

A bug was discovered in the UNIX code for process termination involving the *proc* and *xproc* structures. (The problem also existed on the PDP-11, but it would only be noticed if a process had accumulated more than 65535 ticks of system time, which is highly unlikely.) When a process dies its resource utilization statistics (currently only exit status, system, and process CPU time) are temporarily saved so that they can be added to the totals for the descendents of the parent process. The actual accumulation is done by the kernel when the parent process issues a **wait** system call; the child process is then completely erased. The kernel was overlaying the statistics in a part of the *proc* structure normally used by the scheduler to contain the pointer *p_textp.* Ordinarily the exit was processed immediately, causing no harm. But if the system was loaded so that swapping was necessary, then the scheduler could sneak in after the child exited and before the parent read the statistics, and would interpret the timing data in the zombie *xproc* structure as a pointer. This invariably caused an illegal memory reference from kernel mode on the VAX-11/780.

One of the greatest disappointments with the current system stems from a design quirk in the FP-11 floating-point processor for the PDP-11. When converting between floating-point and 32-bit integer, the FP-11 expects the high-order 16 bits of the integer to be stored at the lower memory address; this is not in line with the general "right to left" design of the PDP-11, which would place the low-order 16 bits in the lower memory address. C code for the PDP-11 uses the FP-11 convention for storing **long** integers. The VAX-11 hardware stores the least significant bit of any integer data type in the lowest addressed byte. C code for the VAX-11 uses the hardware convention. This means that files containing long integers represented in the local convention are not binary compatible between a UNIX system on the VAX-11 and a UNIX system on the PDP-11. This is the only exception for data types common to both machines: **char, short, float,** and **double** all have a common representation. Except for this (and the structure alignment problem noted earlier), disk packs containing 32-bit file systems, tapes, etc., would have been interchangeable. The fact that DEC's Fortran-IV Plus for the PDP-11 avoided the FP-11 convention, and that RSX-11 files are binary compatible between the VAX-11 and the PDP-11, is only salt on an open wound!

**Subroutine libraries**

**libc.** Conversion of the system-call interface routines was straightforward but tedious. Most routines are merely

```
        .word 0x0000
        chmk Snn
        bcc L1
        jmp cerror
L1:     ret
```

The routines *printf*, *ecvt*, and *fcvt* were left to **libS** and were not implemented in *libc.*

**libS**. Conversion of the standard input/output library **libS** posed no problems except for *_doprnt,* the routine which constructs character representations of other datatypes for the printing routines *printf, fprintf,* and *sprintf.* Since many programs spend 15% to 20% of their execution time within *_doprnt,* it pays to code the routine for speed in assembly language. Packed-decimal instructions handle decimal, unsigned, and floating-point conversions. The algorithm chosen for converting from floating-point to character string revealed a microcode bug in the VAX-11/780's **ashp** (arithmetic shift and round packed) instruction. Under certain conditions a carry from the rounded digit propagated both to the adjacent digit and to the digit eight places further left. This usually caused an overflow, since the destination packed-decimal string was typically not long enough to represent the spurious carry. DEC claims to have a fix for the bug, but the

FCO has not arrived. In the meantime a five-instruction patch detects and corrects the spurious overflow.

**Commands**

**as, ld.** Code developed by Center 127 for the Interdata 8/32 was the model for an interpretation by a VAX-11/780 artist. The assembler uses an algorithm described in [3] with heuristic improvement of [4] to resolve conditional jump pseudoinstructions. Variable-length, unaligned instructions and address constants forced the relocation information in object tiles to include the explicit segment-relative address for each relocatable datum, rather than deducing the address from a one-to-one correspondence between the position in the segment and the corresponding position in the relocation table. This caused a slight change in the header information within object files.

**c2.** The code improver for the assembly language generated by the VAX-11 C compiler is based on a similar program for the PDP-11. A "backwards" register usage pass, performed once and before anything else, was a major addition. Knowing that no temporary register is live across a backwards jump, the register usage pass introduces three-address instructions wherever possible. It also recognizes situations where jump on bit (**jbc, jbs, jlbc, jlbs**), extract field (**extzv, movzbl**) and move address (**moval, movab, pushal, pushab**) instructions can be used. The code for insertion of fancy loop control instructions **sob, aob, acb** was also extended.

**adb.** The most significant change to the symbolic debugging routine was the writing of a disassembler for VAX-11 native-mode instructions. Additionally, the character input and output routines were modified to use a default radix for all numeric values. The radix is initialized to sixteen.

**sh.** The (Bourne) shell is the standard user command interpreter. It required by far the largest conversion effort of any supposedly portable program, for the simple reason that it is not portable. Critical portions are coded in assembly language and had to be painstakingly rewritten. The shell uses its own *sbrk* which is functionally different from the standard routine in **libc.** The shell wants the routine which fields a signal to be passed a parameter giving the number of the signal being caught; *signal* was also a private routine. This was handled by having the operating system provide the parameter in the first place, doing away with the private code for *signal.* The code in *fixargs* (for constructing the argument list to an **exec** system call) had to be diddled.

**ps, iostat.** The process and input/output status commands consistently referenced /dev/mem (physical memory) when they should have referred to */dev/kmem* (kernel virtual memory). *iostat* also assumed that certain variables maintained by the kernel were allocated contiguously, even though they were not declared as part of a structure.

**pr.** The command which formats and prints files had a bug that caused a division by zero when it was asked to print several files and the first file in the list did not exist. On a PDP-11 division by zero returns the dividend, but on a VAX-11 it gives an unmaskable trap.

**cat, dc.** These two commands did not count their arguments using the first parameter *argc,* but rather assumed that an additional argument (*argv[argc]*, initialized as -1) could be used as a pointer. On the PDP-11 the resulting address references the fixed end of the stack; on the VAX-11, -1 is an illegal address.

**nroff/troff** The source code for the document preparation and phototypesetter commands is not portable; several weeks were required to produce properly running version of these commands. Use of the explicit (or worse, implicit) constant "2" instead of **sizeof** *(int)* was quite common. The code assumes that variables which are adjacent in external declarations occupy contiguous memory at execution time. Several tables are initialized by assembly-language programs. Converting the tables was merely tedious; changing the code which thought it knew the format of an *a.out* file required some effort. This memorandum was created using the converted *nroff/troff* programs on the VAX-11/780.

**SCCS.** Version 4 of the Source Code Control System [5] is used to provide version backup for software in case disastrous bugs are introduced. The source for SCCS itself had not quite been converted to version 7 UNIX, and the header files required some massaging. The PWB routines *logname* and *pexec* had to be simulated. The utility procedures for dynamic storage allocation required some work to integrate them with **libS** and to remove PDP-11 dialect. The exit status of the *diff* command changed in version 7, causing *delta* to bomb. The code implicitly assumed that all checksums were computed modulo 65536. The documentation is incorrect: everywhere "99999" appears it should really say "65535". The procedure *satoi* returns two values, storing one of them indirectly through a pointer parameter. Naturally, *satoi* and its callers did not agree on **sizeof** the stored value; this took a day to track down.

**Software portability**

We thank the members of Center 127, Computing Science Research, for their efforts in producing the basic software and for their recent efforts towards making the software portable. The fact that people other than the original developers can quickly create a running system for a new machine is a tribute to how well the original work was done.

Yet in our effort to transport a complete UNIX system to the VAX-11/780 we stumbled across a large number of nonportable constructions and were dismayed by the seeming lack of appropriate facilities to detect and prevent them. Based on our experience, we strongly recommend that the C language and its compilers be enhanced so that

1. The actual arguments in a procedure call are type checked against the procedure declaration, and a "dummy" declaration which specifies types is permitted even if the called procedure is not actually declared in the same compilation.
2. The '->' operator is checked to insure that the structure element on the right is a member of a structure to which the pointer on the left may point.
3. A structure element may be declared with any name as long as the name is unique within the immediately surrounding structure. (The current requirement that a structure element name must uniquely correspond to an offset from the beginning of the structure, across all structures in a compilation, creates naming problems and frequently leads to errors of the type noted in item 2 above.)
4. The issue of alignment to an even-byte (or other) boundary is brought into the open, so that arbitrary data structures an be accurately described.

There is a program called *lint* [6] which, if conscientiously used throughout the life of a piece of software provides type checking which partially addresses the first two points in the above list. The problem is that *lint* is big, noisy, relatively recent and unknown, and (partially as a result) infrequently used. There is little incentive for the average programmer to use *lint* as a matter of course. The authors believe that type checking belongs in the everyday compiler as the default, where it is very inexpensive to implement. Those who wish to do "dirty" work may request that type checking be disabled; those who wish to bless their dirty work may use type casts.

We believe that these four enhancements would go a long way towards making C language software portable as a rule rather than as an exception, thus preserving Bell Laboratories' investment in present and future C software.

*Acknowledgements.* Thank you, D. M. Ritchie and S. C. Johnson, for answering questions at key moments; G. K. Swanson, for assistance with boot procedures and stand-alone utilities; J. F Jarvis, for the mathematical function library; and D. K. Sharma, for help in bringing up user-level commands. Additional thanks go to many other members of Centers 127 and 135, and Department 8234, for helpful comments and suggestions.

**References**

1. Digital Equipment Corporation, *VAX-11/780 Architecture Handbook,* Maynard, Massachusetts, 1977.
2. D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, CACM 17, 7 (July 1974), 365-375. See also BSTJ 57, 6 (July-August 1978), 1905-1929.
3. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler,* American Elsevier, New York, 1975.
4. J. F. Reiser, Common Instances of Pathological Span-dependent Instructions, TM 78-1353-3.
5 *SCCS/PWB User's Manual, The Source Code Control System.*
6. S. C. Johnson, *lint,* a C Program Checker. Computing Science Technical Report #65, Bell Laboratories, December 1977.

| Program | System | Text | Data | Bss | Total |
|---|---|---|---|---|---|
| /unix | | | | | |
| | PDP-11 | 48064 | 2470 | 44040 | 94574 |
| | VAX-11 | 34476 | 4292 | 39448 | 78216 |
| | Interdata 8/32 | 79976 | 11904 | 39208 | 131088 |
| C, pass 1 | | | | | |
| | PDP-11 | 36736 | 19826 | 17656 | 74218 |
| | VAX-11 | 37520 | 29492 | 23512 | 90524 |
| | Interdata 8/32 | 60606 | 32192 | 24920 | 117718 |
| C, pass 2 | | | | | |
| | PDP-11 | 21248 | 6254 | 5246 | 32748 |
| | VAX-11 | 23408 | 9092 | 7552 | 40052 |
| | Interdata 8/32 | 35652 | 9032 | 7560 | 52244 |
| ed | | | | | |
| | PDP-11 | 10752 | 302 | 4390 | 15444 |
| | VAX-11 | 11552 | 212 | 4556 | 16320 |
| | Interdata 8/32 | 21886 | 480 | 4576 | 26942 |
| grep | | | | | |
| | PDP-11 | 4736 | 408 | 1906 | 7050 |
| | VAX-11 | 4864 | 476 | 1936 | 7276 |
| | Interdata 8/32 | 11950 | 1160 | 1936 | 15046 |
| ls | | | | | |
| | PDP-11 | 7104 | 768 | 3856 | 11728 |
| | VAX-11 | 6884 | 1140 | 5764 | 13788 |
| | Interdata 8/32 | 15660 | 1920 | 5768 | 23348 |
| nroff | | | | | |
| | PDP-11 | 29312 | 6684 | 7842 | 43838 |
| | VAX-11 | 36360 | 9408 | 10636 | 58836 |
| | Interdata 8/32 | - | - | - | - |
| sort | | | | | |
| | PDP-11 | 6656 | 1578 | 2104 | 10338 |
| | VAX-11 | 6580 | 1764 | 2788 | 11132 |
| | Interdata 8/32 | 13886 | 2208 | 2792 | 18886 |

Table 1. Loaded Program Sizes (in bytes)