TRANSPORTING A PORTABLE OPERATING SYSTEM: UNIX TO AN IBM MINICOMPUTER

PAUL J. JALICS and THOMAS S. HEINES

Cleveland State University

Paul J. Jalics has research interests in the design and performance measurement of operating systems, transportability of programs, and the performance of user application programs. Thomas S. Heines has research interests in the design, measurement, and evaluation of operating systems.

Authors' Present Address: Paul J. Jalics and Thomas S. Heines, Department of Computer and Information Science, Cleveland State University, Cleveland, Ohio 44115

UNIX is a trademark of Bell Laboratories. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. © 1983 ACM 0001-0782/83/1200-106675¢

BACKGROUND

An operating system is a complex set of computer programs that manages the execution of application programs. Every computer typically has its own operating system written in a machine-dependent language. Thus, every time a new machine is developed, a new operating system and new compilers must also be developed. In addition, existing application programs need modification because the new operating system requires facilities that are different from the old one.

The first development promising to simplify this complex process was that of higher level implementation languages designed specifically for writing systems software [2, 4, 8]. With the aid of such a language, one could write a system at a higher level with less machine dependency. Later, experience [3, 6] revealed that most machine dependency could be confined to 10% or less of the operating system.

The second such development involved a number of research efforts in which an entire operating system was "transported" from one machine architecture to another [3, 6, 7]. Three basic approaches to transporting were explored:

- 1. Create a hardware emulator for an existing computer on a newer computer and then simply run the operating system and applications of the older machine on the new machine under the emulator. For example, IBM 7090 and 1401 emulators running in IBM 360 computers. This approach was successful but had a continuous performance penalty.
- 2. Create an instruction set and architecture for a mythical machine that is "higher level" and easier to program than normal hardware. Then, write an operating system in the assembly language of this mythical machine. Finally, execute the operating systems and application programs written for the mythical machine by emulation. The advantage here is that only the emulator is machine-dependent. The San Jose labs of IBM implemented EDX this way on the IBM System/7 computer [5]. EDX was later transported to the IBM Series/1. The performance penalty of emulation still had to be paid, but the fact that the emulated "instructions" were higher level may have reduced this effect somewhat.
- 3. Write the entire operating system and the compilers in a higher level language and isolate all machine-dependent parts of the operating system and compilers. Then, re-

SUMMARY: The "portable" UNIX operating system was transported to an IBM Series/1 minicomputer. The process of transporting is described with emphasis on (1) adapting to the target machine architecture; (2) the selection of the approach taken to transporting; (3) a description of the problems encountered; (4) the degree of portability of the UNIX system; and (5) a summary of the portability lessons learned. write the code generator for this language to generate machine instructions for the new machine. Rewrite the machine-dependent parts of the operating system for the new machine. Finally, recompile all of the operating system using the modified compiler to come up with an operating system that will execute on the new machine. This approach offers the best potential for readable, understandable, transportable operating systems with the best possible performance.

SPECIFIC GOALS

Just as our interest in portable software became more focused, Bell Labs made available a so-called "Portable UNIX" operating system (Version 7), which embodied substantial lessons learned from transporting UNIX to two different machines. It also included improvements of the user applications which had also been transported to various machines on which the UNIX C compiler was available.

We obtained a copy of this UNIX operating system and proceeded to start a portability project of our own to transport UNIX to an IBM Series/1 available in our lab. The goals of the project were as follows:

- 1. To follow up on the work of Ritchie and Johnson [3] and verify the transportability claims for UNIX.
- 2. To add to the base of experience in the transportability of operating systems especially regarding new problems arising from aspects of the new machine's architecture.
- 3. To transport an operating system without prior intimate knowledge of its internals. Most other projects were carried out by people who had a great deal of prior knowledge. We felt that the perspective of an outsider would yield interesting insights and different estimates of the work involved.
- 4. To make more effective use of the IBM Series/1 machine. The operating system supplied, RPS, was found to be difficult to use and slow for program preparation. In addition, it required so much memory to execute efficiently that there was little room left for user programs.

PREPARATORY STEPS

At the time we started work on the project, we had no PDP-11 machine available to us. We did not fully appreciate the difficulty of transporting UNIX without having a UNIX home base to run on. Also, UNIX documentation is terse and best understood by people that already know most of its contents.

The first challenge was to get cross-referenced listings from the release tape containing UNIX source modules. This was no small task since the tape contained over 2000 files and did not include a description of the tape format. A bit of detective work was necessary to decode the file system image on the tape.

At this point, we attended a UNIX Users' Conference (Toronto, 1979) where we learned that Professor D.J. Farber of the University of Delaware was just completing his modifications to the standard UNIX C compiler to generate code for the IBM Series/1. Dr. Farber was kind enough to let us use his compiler, thus saving a substantial amount of effort.

After carefully studying the material, we decided that we lacked the resources to carry out this project without a working UNIX system; we placed an order for a PDP-11/34.

During the six months we spent awaiting the PDP-11, we studied the internals of UNIX using the commentary by Lions [1]. In a way, it was an advantage not to have the opportunity to start immediately; we had time to sit back and study the problem carefully instead.

HARDWARE CHARACTERISTICS OF THE IBM SERIES/1

The main features of the IBM Series/1 are described below:

- 1. A 16-bit word size, with byte addressing and the potential of a 16 mbyte physical memory.
- 2. Four sets of eight "general purpose" registers of 16 bits each.
- 3. A stack mechanism with instructions for allocating stack frames from 1 byte to 16K bytes, and for pushing and popping data. Stack instructions address a three-word stack control block in memory. Stack overflow and underflow are detected by the hardware, and instructions causing stack faults may be restarted easily.
- Memory segmentation using 2K-byte blocks, with eight 64K-byte address spaces where separable address spaces can be specified for instruction fetching, source operands, and destination operands.
- 5. A large instruction set with instructions of one, two, or three words. The instruction set is not particularly symmetric with some operations available in all forms while others are restricted to register-to-register or register-tomemory only.

ADAPTING UNIX TO NEW HARDWARE: THE DESIGN DECISIONS

Subroutine Linkage and Register Usage

The Series/1 has eight 16-bit registers like the PDP-11 and should be at an advantage since none of the registers are dedicated. In fact, the Series/1 has only three really useful registers because of the asymmetric addressing modes. Some addressing modes can use registers 0-7, but others are restricted to 0-3 or 1-3. This makes registers 0 and 4-7 only marginally useful especially in a compiler code generator, which does not easily lend itself to a complex structure or special cases.

The decisions here were already made by the Delaware C Compiler implementors. We simply reviewed them and saw that they were adequate for UNIX:

- Register 0: the address of the stack control block. This is not necessary but reduces by one word each stack-related machine instruction.
- Register 1: frame pointer to local variables.
- Register 2: parameter register for UNIX system calls and function-return register.
- Register 3: second parameter register for UNIX system calls and second part of function return for 32bit values.
- Register 4-6: rarely used temporary registers.
- Register 7: return address from last procedure call.

The calling sequence pushes parameters on the stack, does a branch and link to the subroutine, and pops parameters from stack. The subroutine prologue consists of one instruction which saves registers, allocates local storage on the stack, and saves a control word on the stack which describes these actions. The subroutine epilogue then consists of one instruction which undoes all these actions and returns.

Address Space and Memory Management

All addresses on both the PDP-11 and Series/1 are 16-bit; therefore the basic address space is 64K bytes long. The PDP-11 is able to double this by having separate address spaces for instructions and data on some models. The Series/1 has similar facilities and a slightly more generalized structure on some models. There are eight complete, 64K-bytes address spaces, each having its own set of 32-segmentation registers. There is an address key register designating which of the eight address spaces is to be used for:

- 1. fetching instructions (like PDP-11 I space).
- 2. source data operands (like PDP-11 D space, but only for source operands).
- destination data operands (like PDP-11 D space, but only for destination operands. Having separate source and destination fields is useful for transferring data between address spaces).

The availability of eight-address spaces required a decision on how to best use them in UNIX. All I/O is mapped on the Series/1, meaning that any process doing raw I/O (I/O directly into user memory rather than buffered through system buffers) must also have one of the eight-address spaces. Thus the following allocation of address spaces was made:

- Address space 0: operating system data space.
- Address space 1: operating system instruction space.
- Address space 2: current-user data space.
- Address space 3: current-user instruction space.
- Address space 4–7: dynamically allocated to raw I/O.

Main Memory Allocation

The PDP-11 main memory is allocated in UNIX in 64-byte units called clicks, in harmony with the PDP-11 segmentation hardware. The Series/1 allows 2K-byte segments that must start on 2K-byte memory boundaries. The question arose as to whether the definition of clicks should be changed. Study indicated that memory allocation was always done in physical segments; and so the click was redefined to be 2,048 bytes to match the Series/1 segmentation hardware.

Allocation of Hardware Levels and Interrupt Masks

The Series/1 has four complete sets of registers each associated with a level. A program running at Level 2 can be interrupted by any I/O or external interrupt at Level 0 or 1 but not Level 3. SVC and some error interrupts always occur on the same level as the program causing the interrupt. In addition, there is a four-bit level mask which will prevent any IO or external interrupts at the levels where the mask bit is set. Finally, there is a summary-mask bit in the status register of each level which controls all I/O and external interrupts while that level is active.

The UNIX system is based on the PDP-11 architecture which has eight levels of priority where user programs use Level 0; a long clock routine uses Level 1; terminal interrupt handlers use Level 4; disk interrupt handlers use Level 5; the clock interrupt handler uses Level 6; and all interrupts are prevented at Level 7. Throughout UNIX, there are subroutine calls that change the priority. On the Series/1, it is relatively expensive to change hardware levels since register set contents have to be copied. Analysis of the uses and durations of sections in the operating system where priority was changed yielded the following design:

- Level 0: hardware clock and high-priority I/O devices.
- Level 1: medium-priority I/O devices.
- Level 2: low-priority I/O devices and the long clock processing.
- Level 3: user programs and system call processing.

The UNIX routines setting the priority were changed to set/

reset the four level masks and the summary mask was not used.

The hardware facilities were not well-matched here with UNIX. The Series/1 has a more complex structure, but one that did not provide any extra advantages.

Miscellaneous Considerations

A button on the front panel causes a console interrupt which was used to toggle a debugging facility to turn the periodic clock interrupts on and off. Most of UNIX will run without a clock. This allowed single-stepping and breakpointing most of UNIX or user programs. This low-level debugging is very useful in the first stages when every component is suspect.

The Series/1 was not designed for demand paging, with the segmentation exception seemingly treated as an error as in the PDP-11. On the other hand, stack faults (overflow/underflow) were detected by the hardware and the instruction simply had to be restarted to continue. This was used to automatically extend the stack and reexecute the instruction causing the fault, which cannot be done reliably on some models of the PDP-11.

TRANSPORTING STEPS

The C Compiler

Most of UNIX and its application programs are written in the C language which is an implementation language giving reasonable access to bits, bytes, and pointers which are normally thought to be machine-dependent. C programs have been found to be quite transportable.

The C compiler received from Delaware was a modified Ritchie Compiler. This compiler has three phases: syntactic analysis, code generation, and code optimization. The first of these is almost completely portable, the second and third phases had been almost completely rewritten. Finally, the output of Phases 2 and 3 is a UNIX-style Assembly language. The Delaware implementors wrote an Assembler for the Series/1 code entirely in C. The PDP-11 UNIX Assembler is written in Assembler and is not portable at all.

THE UNIX C RUN-TIME LIBRARY

Most of the work needed for the run-time library entailed writing the Assembler interface routines which allow UNIX system calls from C programs. Some 2,560 lines of Assembler statements were rewritten, but it was a repetitive process because of the similar structure of all the routines. The only routines requiring substantial insights were the ones dealing with signal interception and processing, output formatting, and double-length integer arithmetic. Some of the formatting routines (like *printf*) could easily have been written in C, thereby making them quite transportable. Instead, they were written in Assembler presumably for performance reasons. Floating-point emulation routines were written but were not an essential part of the project since UNIX does not use floating point nor do most commands.

TRANSPORTING UNIX

The minimal UNIX supporting one disk and a console terminal consists of 34 modules totaling some 9,200 lines and two assembly modules totaling about 850 lines.

Rewriting the Assembly Modules

One assembly module simply described the low-memory layout and initialized the interrupt vectors. This was completely machine-dependent and needed design planning to adapt to UNIX requirements. Once this was done, the work involved in this module was minimal.

The other module contained the code that directed hardware interrupts to the appropriate C language handler, plus a host of short routines called from C for initiating I/O, setting up segmentation, and copying data to/from the user-address space. While a number of small decisions had to be made about how to use the existing hardware, this code followed the pattern of the module for the PDP-11. Thus, we had to understand what the PDP-11 routines did and then write analogous code.

Transporting Memory Management

This was one of the most machine-dependent parts of the operating system. Almost all of the code relating to memory management/segmentation was contained in one short module and a few Assembler routines. Once the design decisions about the use of the segmentation hardware had been made, rewriting routines was straightforward.

The only difficult bugs encountered involved the redefinition of clicks from 64 to 2,048 bytes. Several places were found in the operating system where the magic numbers 64 or 6 (2**6=64) were used. After finding a couple of these, we used our cross-reference for all occurrences of 5, 6, 7, 63, and 64 and thus finding all the places. These had to do with converting addresses to/from clicks and setting or reading segmentation register contents.

Transporting the Machine-Dependent C Code

A number of short routines that were machine-dependent were in one C module. Once one understood what was intended with each routine, it became straightforward to modify or rewrite it. Reasonable header comments for most routines gave a good indication of what a routine did. Routines for starting the clock or an I/O operation are done on the PDP-11 by addressing the appropriate UNIBUS locations directly from C. The Series/1 does not use memory-mapped I/O, so Assembler routines were coded to use the appropriate I/O instructions.

The other main module dealing with machine dependencies was the C interrupt handler for all system calls and error interrupts. This had to be modified substantially to take care of all possible Series/1 interrupts. The format of the stack at interrupt time was quite different from that of the PDP-11, but once one understood what the original code did, adapting to the new was straightforward. We tried to make the best possible use of the different kinds of I/O interrupts, error interrupts, and system calls on the Series/1. The I/O interrupts did not need to save the environment since they took over an unused hardware level, whereas the error interrupts and system calls occurred on the same level as the executing program. For the error and system call interrupts, the environment was saved in a block which was unique to the kind of interrupt that occurred. An I/O interrupt might at its completion have to call the scheduler which was normally called from error interrupts or system calls. Also, an error or system call interrupt handler could not save its environment in the special save blocks for long since another interrupt of the same kind could occur before completion of the first.

We decided to transform all types of interrupts into a common type with all of the environment saved on the system stack. This was another case where a more complex but incompatible architecture denied us the advantage of some Series/1 facilities for quick switching for I/O interrupts.

Transporting Device Drivers

I/O device drivers are among the most machine-dependent parts of an operating system. One must understand the device-driver interface to the rest of the operating system. This is best done by studying existing drivers for character and block-type devices. The task of writing a device driver became one of mimicking most of the functions of the old.

The console terminal driver, for example, was over two thirds identical to that for the PDP-11. The system interface for drivers is simple and the time was spent getting the device-dependent details right for some of the devices. The matrix printer required a control table to allow it to respond to ASCII character codes rather than to the EBCDIC codes which were used by default. The terminal multiplexor worked in only half duplex and also reversed the bits in each character requiring that a translation be done into ASCII.

The problem of half-duplex terminal hardware was made more complicated because the break key worked only when output was active. Periodic switching from output to input mode created a system which felt like UNIX but was not quite as responsive to control commands to quit or stop output. Later, acquisition of full-duplex hardware solved this problem.

TRANSPORTING THE REST OF THE UNIX

The remainder of the operating system had only a few minor changes. This code consisted of 6,942 lines of C, only 123 of which were changed. These changes were for machine dependencies that related to segmentation and memory allocation.

This code included most of the code of the system calls, the management of terminal input/output queues, buffer management, scheduling, and swapping. A number of these were made to work without comprehension of their logic. This was the only such area; most other components had to be understood in fine detail before they could be transported.

Bootstrapping UNIX on the Series/1

UNIX expects to see a file system on a disk when it is booted, so a UNIX operating system and a file system must be brought to the target machine. UNIX has no overlays or nonresident segments which allows the operating system to load simply. But before UNIX is started up, a UNIX file system must be on the disk.

The UNIX system had to be taken from the development machine to the Series/1. Initial plans were to use a standalone utility to create the image of a UNIX file system onto the Series/1 disk and copy UNIX via a communication link.

We decided against the above strategy because we did not want to write the programs needed and because it would have been difficult to debug the file system on the Series/1 until UNIX was up. We were anticipating more debugging than was actually required. We finally decided to create a UNIX file system on a diskette on the PDP-11, then modify it to make the PDP-11 file system acceptable to the Series/1 UNIX. This diskette was then taken over to the Series/1 and booted from the diskette drive. The diskette contained the required file system and a file containing UNIX.

One advantage of this strategy was that once the system was tried and the file system was suspected of having been altered or destroyed; we could use UNIX utilities to look at it. This could be done by taking the diskette back to the PDP-11, reversing the modifications, and looking at the file system with the standard UNIX utilities. We did this a number of times and it helped us to find some early bugs.

The UNIX Byte-Reversing nUxi Problem: Inheritance from the PDP-11

A persistent problem encountered in the transporting process was one related to addressing on the PDP-11. We noticed this in one of the first programs brought over to the Series/1. The program was supposed to print out Unix and instead printed out nUxi (reversing pairs of bytes). Each machine addresses words by an even address. The PDP-11 addresses a word by its low-order byte, and the Series/1 addresses a word by its high-order byte. The effect is that byte-oriented data are correct when taken from one machine to another; but wordoriented data, including instructions, must have the bytes of each word interchanged. This architectural difference had substantial implications in moving any data between the PDP-11 and the Series/1. In addition, this difference was incorporated into program logic whenever a two-byte sequence was considered at times as a 16-bit binary word, and at other times a sequence of two 8-bit bytes.

Moving source files pose no problems. Object programs were one of the most important things to be moved, and they did contain both character and binary data. It is impossible to take a file which contains a mixture of words and bytes arbitrarily intermixed and "fix" it so that it can be interpreted properly on the Series/1. Object files on UNIX are all created by the UNIX Assembler, which distinguishes between characters and binary words because the C Compiler makes this distinction when creating its output for assembly later. Thus, the cross-assembler which ran on the PDP-11 was able to ensure that the load module arrived correctly on the Series/1.

Transporting a file system was more complicated because the file directories and the allocation tables were a mixture of characters and binary words of 16 or 32 bits. In addition, file systems contain files which contain characters, binary data, and object programs. To create a file system on the PDP-11 which is acceptable to the Series/1, the three-step sequence of modifications was undertaken.

The first step was to swap all pairs of bytes on the entire file system, thereby making all the binary integers correct on the Series/1. This made all integers and longs correct but left all character sequences incorrect. It turned out that character sequences are used in the file system data structures only for storing file names or directory path names. So a slightly modified version of the *ncheck* utility was then used as a second step to reverse pairs of characters in the file directories.

At this point, the entire file system data structures were acceptable to the Series/1. Then, the byte-swapping program was run individually on the character files to complete the modifications.

While all this seems complicated, it did not involve substantial programming and took only 5 minutes to run.

nUxi Problems in C Programs

A more difficult problem came from styles of programming which made implicit assumptions about the placement of bytes in words. Most often this happened when a 16-bit quantity was sometimes treated as a binary number and at other times treated as two separate characters. Several examples of this were found in UNIX and in commands.

TRANSPORTING UNIX COMMANDS

Once UNIX as an operating system has been booted, you

quickly find that everything works but there is very little that can be done. Most of UNIX's facilities are the commands. There were so many commands that we made a priority list and transported the most important ones like the editor and the file-system utilities first. A surprising number of them (editor, loader, cp, tar, shell, etc.) worked immediately without any changes. A few, like the dump processor od worked partially, but had small nUxi problems. Our machine had no floating-point hardware and our C Compiler did not compile floating point, so a few items relating to floating point were missing from our C library. This affected only a few commands.

TIME SCHEDULE

1. The UNIX Version 7 release tape and documentation arrived in June 1979. About two man-weeks were spent decoding the tape format and generating cross-referenced listings of selected groupings of source files from that tape.

2. The rest of that summer, full-time efforts were directed to understanding the UNIX Kernel with considerable help from the Lions Commentary [1] which described an earlier UNIX.

3. Part-time activities in the fall and winter completed the design decisions for the Series/1. Stand-alone I/O programming on the various Series/1 I/O devices was also done.

4. The spring quarter of 1980 saw the implementation of the assembler parts of the operating system, the modifications to memory management, and other machine-dependent parts.

5. The first 6 weeks of the summer were spent wrapping up the changes to the operating system, including writing terminal and disk I/O drivers. Concurrent with this, the assembly parts of the C run-time library were written. The system was first booted in early August and within a month became quite stable and usable.

6. Part-time activity in the fall of 1980 saw the implementation of I/O drivers for the remaining devices and the transporting of the UNIX commands to the Series/1. This was followed by the transporting of the C Compiler and Assembler to the Series/1.

7. In January 1981, we called the project "finished." This was necessary since the task of maintaining and embellishing operating systems and user environments is never really finished.

SUMMING UP PORTABILITY LESSONS

UNIX Version 7 is a portable operating system, i.e., one that can be transported with less effort than writing it anew. We estimate that the project took about 16 man-months, starting with a good knowledge of operating systems but no knowledge of UNIX. This does not include the modification of the C Compiler and the writing of the Assembler which we received from the University of Delaware. Since the UNIX system has several man-years of effort in it, it would seem that the system is highly portable because it takes much less effort to transport than to rewrite.

The authors believe that UNIX cannot be transported to a new architecture without intimate knowledge of UNIX and the C Compiler. While major sections of the operating system will work unchanged on the new machine, one needs substantial insights to make reasonable design decisions and to do the actual debugging.

A continual problem on transporting UNIX was the socalled nUxi problem, which arose from PDP-11 and Series/1 hardware differences. Future transporters might well consider TABLE I. Report of Changes by Groups

Group Identification	Lines Changed (%)	Original Number of Lines	IBM S/1 Lines	Number of Changes	Lines Deleted	Lines Inserted
Stand-Alone	100	0	727	3	0	727
Commands	4	23,298	24,035	82	142	879
Kernel	25	9,565	11,901	328	637	2,973
C Library	37	6,463	6,678	305	2,252	2,467
C Compiler	61	10,093	13,799	831	4,654	8,360
Assembler	100	0	4,537	8	0	4,537
Totals	33	49,419	61,677	1,557	7,685	19,943

starting with a version of UNIX which is similar to their target machine hardware.

A substantial portion of the debugging of the Series/1 UNIX was spent tracing bugs related to bad code generated by the C Compiler. The C Compiler we received from the University of Delaware was already in fairly good shape. The point is that a lot of time can be wasted tracing even minor Compiler bugs with a not-yet-working operating system. The authors believe that a battery of tests for the C Compiler ought to be developed and run before ever trying to port an operating system with a new Compiler. These programs could be written to be largely machine-independent. UNIX has fine facilities for comparing outputs of correctly running tests to those being tested, and so much of the C code generation validation could be automated with concise reporting of exceptions.

The differences between the original and modified source files for the entire project are summarized in Tables I and II. They were generated with the UNIX *diff* command, which strives to find the minimum number of lines different. Although the number of lines of code different may not be the best indicator of effort required, it provides a simple, unbiased value. For example, the effort required for the 2,422 lines of Assembler code for the C library of UNIX is much less than the 850 lines of Assembly code rewritten for the operating system.

The majority (77%) of the lines changed relate to the C Compiler language system. A substantial part (10%) of the remainder went to rewriting the I/O drivers for a number of devices on the Series/1. Only 7% was involved with Assembler parts of the operating system. The remaining 6% was minor changes to the operating system and UNIX commands.

Many of the same problems we encountered will be encountered by future transporters of UNIX. For this reason, we created a complete set of line-by-line difference files which were produced for every module transported. The statistics themselves are useful to future transporters and the difference

TABLE II.	Summary	of Code	Changes
-----------	---------	---------	---------

Group Identification	Lines Changed	C Lines Changed	ASM Lines Changed
UNIX Kernel	3,700	2,275	1,425
UNIX Commands	879	879	0
C Compiler	8,360	8,360	0
C Library	2,467	45	2,422
Assembler	4,537	4,537	0
Totals	19,943	16,096	3,847

files could be useful to pinpoint the machine-dependent parts of the system as well.

The UNIX developers did not put many comments in programs. Many of the C procedures have header comments that indicate what they do, and these are welcome. The Assembler modules, on the other hand, are almost totally lacking in comments. This was disconcerting to a person trying to understand UNIX and the PDP-11 at the same time. Lack of comments in parts that obviously have to be modified or rewritten is a serious impediment to portability.

Much of the reprogramming in UNIX was quite straightforward because there were existing models for the code that needed rewriting. The ease of transporting I/O drivers, especially, surprised us since the manner of doing I/O is so different on the two machines.

Using a diskette file system for the transporting of UNIX from the host system was very satisfactory. Usually, the full cycle of changes to source code, compilation, relinking, creation of a Series/1 acceptable file system, and the booting of that system on the Series/1 took less than 15 minutes.

We believe it requires a Herculean effort to transport UNIX without having a host UNIX system for the program development. The program development facilities in UNIX are so good that using UNIX for development is preferable even if you must adapt a UNIX Assembler to do it. Which brings up another point: the Assembler for UNIX Version 7 is written in PDP-11 Assembler. Since UNIX style Assemblers are so similar for various machines, why isn't the Assembler written in C in a portable manner? Also, a rather lengthy editing routine (for printf) is written in Assembler and it could easily be written in C. We assume the above are in Assembler for performance reasons (or possibly historical reasons).

CONCLUSION

The effort invested in UNIX seemed well-rewarded both in the portability project and the quality of the end result. It became clear that transporting a real operating system involves a substantial amount of effort, even if that system is designed to be "portable." While one might reduce the effort required to a fraction of that required here, the performance and flexibility of the end result will be affected proportionately.

Acknowledgments. We wish to thank Dr. D.J. Farber of the Electrical Engineering Department at the University of Delaware for supplying us with the Series/1 C Compiler and Assembler. Also, a great deal is owed to the graduate students who contributed substantially to our research efforts:

R.M. Szabo and T. Babej for the I/O drivers; and M.A. Venar for the C library.

REFERENCES

- 1. Event-Driven Executive Basic Supervisor and Emulator. Publication SB30-1053, IBM Corporation, San Jose, California, 1977.
- Goos, G., Lagally, K., and Sapper, G. PS440: Progrommiersprache fuer Systemimplementierung. Bericht 70002, Technische Universitaet Muenchen, Munich, West Germany, 1970.
- 3 Johnson, S.C., and Ritchie, D.M. "Portability of C programs and the UNIX system. Bell Syst. Tech. J. 57, 6, Part II August 1978, 2021-2048.
- 4. Kernighan, B.W., and Ritchie, D.M. The C Programming Language. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.
- Lions, J., A Commentary on the UNIX Operating System. Bell Laboratories, Murray Hill, New Jersey, 1977.

- Melen, L.S. A Portable, Real-Time Executive, Thoth. Master's Thesis, Department of Computer Science, Waterloo, Ontario, Canada, University of Waterloo, 1976.
- Thalmann, D., and Levrat, B. SPIP: a way of writing portable operating systems. Proc. ACM Comput. Symp. (1977), pp. 452-459.
- Wulf, W.A. et al., BLISS Reference Manual. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1970.

CR Categories and Subject Descriptors: D.2 [Software]: Software Engineering D.2.7 [Software Engineering]: Distribution and Maintenance portability: D.4 [Software]: Operating Systems

General Terms: Design, Verification

Additional Key Words and Phrases: C, UNIX, transporting, operating system

Received 12/82; revised 4/82; accepted 6/82

ACM SPECIAL INTEREST GROUPS

ARE YOUR TECHNICAL INTERESTS HERE?

The ACM Special Interest Groups further the advancement of computer science and practice in many specialized areas. Members of each SIG receive as one of their benefits a periodical exclusively devoted to the special interest. The following are the publications that are available through membership or special subscription.

SIGACT NEWS (Automata and Computability Theory)

SIGAPL Quote Quad (APL)

- SIGARCH Computer Architecture News (Architecture of Computer Systems)
- SIGART Newsletter (Artificial Intelligence)
- SIGBDP DATABASE (Business Data Processing)
- SIGBIO Newsletter (Biomedical Computing)
- SIGCAPH Newsletter (Computers and the Physically Handicapped) Print Edition

SIGCAPH Newsletter, Cassette Edition

- SIGCAPH Newsletter, Print and Cassette Editions
- SIGCAS Newsletter (Computers and Society)
- SIGCHI Bulletin (Computer and Human Interaction)
- SIGCOMM Computer Communications Review (Data Communications)
- SIGCPR Newsletter (Computer Personnel Research)
- SIGCSE Bulletin (Computer Science Education)

SIGCUE Bulletin (Computer Uses in Education)

SIGDA Newsletter (Design Automation)

SIGDOC Newsletter (Systems Documentation)

- SIGGRAPH Computer Graphics (Computer Graphics)
- SIGIR Forum (Information Retrieval) SIGMAP Newsletter (Mathematical
 - Programming)

SIGMETRICS Performance Evaluation Review (Measurement and Evaluation)

```
SIGMICRO Newsletter
(Microprogramming)
```

SIGMOD Record (Management of Data)

SIGNUM Newsletter (Numerical Mathematics)

SIGOA Newsletter (Office Automation)

SIGOPS Operating Systems Review (Operating Systems)

SIGPC Newsletter (Personal Computing)

SIGPLAN Notices (Programming Languages)

SIGPLAN ADATEC Newsletter (Technical Committee on Ada)

SIGPLAN FORTEC Newsletter (Technical Committee on Fortran)

SIGSAC Newsletter (Security, Audit and Control)

- SIGSAM Bulletin (Symbolic and Algebraic Manipulation)
- SIGSIM Simuletter (Simulation and Modeling)

SIGSMALL Newsletter (Small Computing Systems and Applications)

SIGSOFT Software Engineering Notes (Software Engineering)

SIGUCCS Newsletter (University and College Computing Services)