AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

UNIX Operating System Porting Experiences

By D. E. BODENSTAB,* T. F. HOUGHTON,* K. A. KELLEMAN,* G. RONKIN,† and E. P. SCHAN*

(Manuscript received January 9, 1984)

One of the reasons for the dramatic growth in popularity of the *UNIX*™ operating system is the portability of both the operating system and its associated user-level programs. This paper highlights the portability of the *UNIX* operating system, presents some general porting considerations, and shows how some of the ideas were used in actual *UNIX* operating system porting efforts. Discussions of the efforts associated with porting the *UNIX* operating system to an Intel™ 8086-based system, two UNIVAC™ 1100 Series processors, and the AT&T 3B20S and 3B5 minicomputers are presented.

I. INTRODUCTION

One of the reasons for the dramatic growth in popularity of the *UNIX*[1,2] operating system is the high degree of portability[3] exhibited by the operating system and its associated user-level programs. Although developed in 1969 on a Digital Equipment Corporation PDP-7[‡], the *UNIX* operating system has since been ported to a number of processors varying in size from 16-bit microprocessors to 32-bit main-

* AT&T Bell Laboratories. † AT&T Bell Laboratories; present affiliation Bell Communications Research, Inc.

‡ Trademark of Digital Equipment Corporation.

frames. This high degree of portability has made the *UNIX* operating system a candidate to meet the diverse computing needs of the office and computing center environments.

This paper highlights some of the porting issues associated with porting the *UNIX* operating system to a variety of processors. The bulk of the paper discusses issues associated with porting the *UNIX* operating system kernel. User-level porting issues are not discussed in detail. However, some architectural issues (e.g., byte ordering) are common to both user- and kernel-level code. The processors discussed are the Intel* 8086 microprocessor, the AT&T 3B20S minicomputer, the AT&T 3B5 minicomputer, and the UNIVAC† 1100 Series mainframes.

II. PORTING ISSUES

"Given that I have processor *X*, what do I have to do to get the *UNIX* operating system up and running on that processor?" This is the first question that should be in the mind of anyone interested in porting the *UNIX* operating system to another processor. Before the porting is to begin this question should be refined into the following questions:

1. Of the existing processors that support the *UNIX* operating system, which one will be used as the base? That is, which *UNIX* operating system source will be used as the starting point of the port (e.g., that of the PDP-11/70‡ or VAX-11/780‡ minicomputers)?
2. Is the software generation system (i.e., compiler, assembler, loader) for the target processor available?
3. Is there a mechanism to load object code into the target processor?
4. Is there a mechanism to make the initial file system?
5. Are kernel-level debugging tools available?

The following sections give guidelines to help answer these questions.

2.1 Choosing the appropriate source base

Before any kernel source modifications are attempted, the appropriate base must be chosen. This decision should be based on several criteria that evolve around the architecture of the target processor:

1. Word size.
2. Byte ordering. Are bytes within a word ordered in the same way?

* Trademark of Intel Corporation.

† Trademark of Sperry Corporation.

‡ Trademark of Digital Equipment Corporation.

3. Interrupt structure. Are interrupts handled in a similar way?

4. Input/Output (I/O) architecture. Are intelligent controllers supported?

5. Peripheral support. Do common device drivers exist?

Therefore, if the target processor is a 16-bit microcomputer, the source of the PDP-11/70 processor could be used as a base. Likewise, if the target processor is a 32-bit minicomputer, the source of the VAX-11/780 computer could be used as a base.

2.2 Portable software development system

If any piece of software is to be portable, it should be written in a high-level language capable of running efficiently on a large number of processors. The C programming language,[4] the primary language of the *UNIX* operating system, is a language that meets this criterion.

Although not originally written with portability in mind, the *UNIX* operating system and C have been enhanced to obtain maximal portability. Beginning with the Version 7 release, the *UNIX* operating system has decreased its use of machine language and restricted processor-dependent C code to particular files within the kernel. The development of the portable C compiler, pcc, has greatly improved the portability of both the C language and the *UNIX* operating system. The portable concept has been expanded to a portable assembler and a portable loader. Together, these portable-tools are bundled into a common Software Generation System (SGS). Also included in the common SGS is a Common Object File Format (COFF) and a portable archive file format. Because of this commonality, an SGS and a cross-SGS can be developed for a target processor by changing only the processor-dependent portions of the SGS.

2.3 Executing object files on the target processor

If the target of the port is a stand-alone processor, a host processor is used as a base of operations during the development stages.* All programs are compiled through a cross-SGS and, possibly, tested through a simulator on the host before being placed on the target processor. However, since the target and host processors are independent, a mechanism should exist to allow the host to down load compiled code into the memory of the target processor. This is typically done by connecting the two processors by means of an asynchronous communication line and using simple file transfer programs to populate the memory of the target processor. Once the executable code has

* This is typically the case. However, in the case of the UNIVAC 1100 Series the *UNIX* operating system runs as a task on top of the resident operating system. Therefore, the target and host are the same processor. (See Section IV.)

been placed in memory, its execution must be started by some form of bootstrap monitor. The monitor should give the user the ability to examine memory locations, start and stop program execution, etc. If a bootstrap monitor is not available, it should be developed and placed on the target processor in a manner that will facilitate easy start-up [i.e., read from a floppy disk, placed in Read-Only Memory (ROM), etc.].

2.4 Initializing the file system

As the porting effort progresses, the time will come when it is necessary for the target processor to perform *UNIX* system file accesses. For those lucky enough to have common peripherals this poses no problem. The file systems can be made and populated on the host and placed on the target.

However, if the target and host have no common disk devices, a potential problem exists. This problem could be solved by using a modified memory down-load program. The memory down-load program could be modified to place the data read from the communication line onto the disk, instead of in memory. This, of course, means that a stand-alone disk driver would have to be incorporated into the down-load program.

2.5 Kernel debugging

Two forms of kernel debugging are necessary:

1. Those used to debug a kernel that fails to boot.
2. Those used to debug a kernel that crashes unexpectedly.

For the former case, appropriately placed print statements could be used to trace the execution steps of a suspect operating system. If a bootstrap monitor with a breakpointing capability is available, a breakpoint could also be placed at a suspect point. When the processor reaches the breakpoint, the status of the machine (e.g., examine registers, perform a stack back-trace, etc.) could be examined to try to uncover the error.

In those cases where the system crashes unexpectedly, some form of postmortem debugger should be available. The debugger should be capable of running on either the host or target machine and should have the ability to display the contents of key data structures. A stack back-trace option would also be useful.

2.6 Caveats

The suggestions presented in the previous sections are not meant to be an all-encompassing survey. They are meant only to inspire thoughts by presenting some of the possibilities that exist. The following sections describe how some of these ideas were used in porting the *UNIX* operating system to various processors.

III. THE *UNIX* OPERATING SYSTEM ON THE INTEL 8086

The *UNIX* operating system for the Intel 8086, referred to as the 8086 *UNIX* system, was developed in 1978 to run on a system specifically designed for the Intel 8086 microprocessor. The system was designed for, and is currently used in, some internal AT&T applications.

The central processing element of the 8086 *UNIX* system is the Intel 8086 microprocessor.[5] Main memory can range from 512K bytes to 2M bytes and is accessed via a Memory Management Unit (MMU). Three types of peripheral controllers are supported:

1. Disk controller. Facilities exist to support floppy and Winchester disk devices with capacities of 2M bytes and 20M bytes, respectively.
2. Line controller. The line controller is a programmable device that supports serial synchronous or asynchronous communication protocols.
3. Terminal controller. The terminal controller is a communications device capable of supporting 16 teletype Standard Serial Interface (SSI) lines.

3.1 Hardware-related porting issues

3.1.1 Memory management unit

Two hardware features are essential to support the secure multiuser environment that is needed by the *UNIX* operating system:

1. An address space larger than 64K bytes
2. Privileged (kernel) and nonprivileged (user) modes.

Because a stand-alone 8086 cannot support these features, an MMU was specially designed for the 8086 *UNIX* system. The MMU is similar to that of the PDP-11/70; 16-bit virtual addresses are translated into 22-bit physical addresses through the use of mapping tables and page address registers. The MMU consists of 16 address maps, where each map addresses 64K bytes of memory. The most commonly used address maps are in kernel instruction, kernel data, and exit kernel (user-mode) maps. The larger address space is provided by allowing for split Instruction and Data space (I/D). With split I/D, programs can address up to 64K bytes of text and 64K bytes of data. Split I/D is easily achieved by using two 64K-byte address maps, one for the text segment and the other for the data and stack segments. The division between kernel and user modes is achieved by mapping all user programs through the exit kernel map. While in user mode, any privileged memory accesses or attempts to alter the status of system execution (disable interrupts) by user programs results in a trap to a low-level handling routine where the problem will be rectified.

3.1.2 Peripheral controllers

The peripheral controllers share a basic scheme. In addition to its intrinsic hardware, each controller consists of a Zilog Z80* microprocessor with 32K bytes of Random Access Memory (RAM). This extra computing power permits greater flexibility in software controller development. Efficient disk search algorithms and line protocols are handled on the controlling device, thus eliminating the need for central processor intervention.

The 8086 communicates with each controller via a one-way shared-memory scheme; the 8086 can access the controller's memory but not vice versa. A kernel routine, `window`, exists to place the device specific address into a given location in the kernel data map, thus creating a window to that device.

3.2 Architectural and software-related porting issues

Porting the *UNIX* operating system to the 8086 required software changes at the operating system, library routine, and user-program levels. Because of the similarities between the MMU's of the 8086 *UNIX* system and the PDP-11/70 system, the PDP-11/70 version of the *UNIX* operating system was used as the basis for the 8086 *UNIX* system porting effort. A PDP-11/70 computer was also used as the host processor for 8086 *UNIX* system development.

Several software changes were necessitated by hardware differences between the PDP-11/70 processor and the 8086. The obvious changes included translating the assembly language routines in the *UNIX* operating system into 8086 assembly language and modifying the low core-interrupt routines to fit the 8086 *UNIX* system hardware. Several other basic hardware differences between the PDP-11/70 and the 8086 devices also had to be overcome.

3.2.1 Byte ordering

While the PDP-11/70 and 8086 processors both utilize the same byte ordering within a word, the ordering of words within a double word (*long*) is reversed. The 8086 implements double words with the low-order word occupying the least significant bit positions. Any programs that depended upon this byte ordering (e.g., any program that read long integer values from files) had to be modified. For instance, the example shown below will produce different results when

* Trademark of Zilog Inc.

run on a PDP-11/70 processor using the *UNIX* system from those produced on an 8086 *UNIX* system:

```
long    l = 0x12345678L;
short   *s;

s = (short *)&l;
printf(“%#x\n”, *s);
```

When run on a PDP-11/70 processor using the *UNIX* system the result will be:

```
0x1234
```

while the 8086 *UNIX* system will produce:

```
0x5678
```

Also, since the 8086 is byte oriented, odd function addresses are permitted. The kernel-level signal handling routine, `issig`, was modified to compensate for this difference.

3.2.2 System call interface

The PDP-11/70 version of the *UNIX* operating system uses self-modifying code to pass system-call parameters from user to kernel level. The 8086 *UNIX* system call interface was changed to use registers to pass system call parameters. The system call number is passed in the AX register of the 8086 and the DX register is used for parameter passing. On calls that require one parameter, that parameter is placed in the DX register. In the case where multiple parameters are required, the DX register contains a pointer to a parameter list.

3.2.3 Run-time calling convention

Calling-convention routines for the 8086 *UNIX* system (i.e., code added to implement stack frames) are also different. Since the 8086 does not have hardware restart capabilities, the user stack must be expanded gradually during the local storage allocation process to permit the proper handling of stack warning interrupts. This function is performed by a special function that is called in place of the normal runtime routine when local variables are present. In the process of growing the stack this function clears each word, thus ensuring that each local variable will be initialized to zero.

3.3 Development and test environment

3.3.1 The 8086 UNIX *system SGS*

Early 8086 *UNIX* system development was done using an already existing, internally developed 8086 simulator and a common SGS referred to as the Basic-16 package. Because the 8086 system was designed to make use of the majority of the user- and kernel-level code of the PDP-11/70 version of the *UNIX* operating system, the object file format of the 8086 system is similar to that PDP-11/70 version. Therefore, a tool was developed to convert the common object file format of the Basic-16 SGS to the 8086 *UNIX* system object file format. In addition, the 8086 system object file format was changed to include symbolic debugging information.

An SGS designed around the Basic-16 SGS was later developed to run on the 8086 *UNIX* system. The new SGS uses the Basic-16 compiler, a modified Basic-16 assembler, and a modified PDP-11/70 loader to directly produce 8086 *UNIX* system object files. Using the symbolic debugging information produced by the SGS, `sdb`, a symbolic debugger, was ported to the 8086 *UNIX* system.

3.3.2 The 8086 UNIX *system firmware monitor*

A firmware monitor was written specifically for the 8086 *UNIX* system. Stored in ROM, the monitor is activated on power-up and has its own command language that allows the user to examine memory, set breakpoints in memory, talk through to the host PDP-11/70 processor, etc. The monitor also allowed the user to down load programs directly into the memory of the 8086 *UNIX* system.

Because the 8086 *UNIX* system used a Winchester disk that was not common to the host processor, a stand-alone `mkfs` (Make File System) program was developed to initialize the file system. The stand-alone `mkfs` was down loaded into the memory of the 8086 *UNIX* system by a monitor command. Once execution began, the `mkfs` program performed a handshaking operation with the host to transfer files over an RS-232 port to the 8086 *UNIX* system disk.

3.4 Status

As we previously mentioned, the 8086 *UNIX* system is currently used as the basis for an internal AT&T application. As of this writing, the 8086 system supports *UNIX* System III. However, through kernel modifications similar to those used on the PDP-11/70 version, the 8086 system could be made to support *UNIX* system V.*

IV. THE *UNIX* OPERATING SYSTEM ON THE UNIVAC 1100 SERIES

The *UNIX* system for the UNIVAC 1100 Series[6] runs on Sperry

* Due to addressing limitations a memory management scheme referred to as overlaying was added to support *UNIX* System V on the PDP-11/70 system. The "Overlay" technique could be achieved by using the indexing capability of the 8086 and the unused kernel segment maps in the MMU of the 8086 *UNIX* system. Infrequently executed code could be addressed through the segment registers by appropriately adjusting the index registers.

1100/60 and 1100/80 processors. These processors have similar but not identical instruction sets. They run time-sharing, batch, transaction, and communications real-time programs, simultaneously, if desired, under the control of the OS 1100 operating system (commonly called EXEC). Each processor type can operate in configurations of from one to four Central Processing Units (CPUs) with one to four I/O processors (not all combinations are supported). Processor types cannot be mixed in a single configuration.

The *UNIX* system for the UNIVAC 1100 series was built as an integrated development environment for transactions that run directly on EXEC. Unlike most other implementations, therefore, it runs not directly on the hardware but as a collection of user-level activities under control of EXEC. These obtain services that would normally be provided by device drivers, and some process creation and management services from EXEC. Any configuration supplied by Sperry, including multiprocessor ones, can run the *UNIX* system.

4.1 Effects of hardware architecture on porting

Like all *UNIX* system implementations, this one dealt with peculiarities of the target system architecture. The 1100 hardware architecture differs from other architectures to which the *UNIX* system has been transported in a number of ways. These differences are discussed below.

4.1.1 Data type size

The 1100 C implementation has 9-bit characters (bytes), 18-bit shorts, and 36-bit integer and unsigned data types (longs are also 36 bits). The compiler does not attempt to make these types look like 8-bit multiple lengths to the programmer; the writer or transporter of code dependent on 8-bit bytes for proper functioning is responsible for making the code work with 9-bit bytes, or better, making the code portable.

4.1.2 Word addressing

The machine addresses words rather than bytes. All extension of the operator code field of the instruction can designate to which quarter of the operand the operation applies. Use of this feature requires compile time knowledge of the byte address, which is possible for cases such as references to automatics and structure leaves, but not for the dereferencing of pointers. Pointers contain a simulated byte address that must be dereferenced by generated code rather than addressing hardware. Since this has a considerable adverse effect on performance, the format of pointers was carefully designed to minimize the execution time of this generated code. Early versions of the compiler used simulated byte addresses to aid portability of existing code; later versions used pointers containing a word address in the less significant (right) word half and a byte offset in the left half.

4.1.3 One complement

The 1100 processors use one's complement arithmetic. The compiler makes no attempt to simulate two's complement arithmetic. As is the case with the byte size, writers or transporters of code must be aware of this difference. Fortunately, in actual practice, problems caused by one's complement arithmetic are rare. (Some of the nastiest ones are in the C compiler itself!)

4.1.4 Floating point

There is little uniformity of floating point formats among mainframes, and the 1100 series is no exception. The greatest difficulty was caused by the assumption embedded in the compiler's portable code, that a double may be made from a float by extending the mantissa with a word of zeros; on an 1100, the characteristics differ in size as well.

4.1.5 Banking

Memory management hardware on 1100/60 and 1100/80 processors maps program virtual addresses into the physical addresses of segments, or *banks*. These processors are atypical in that a given virtual address may refer to more than one physical address. In this case, disambiguation is by context, [i.e., whether the fetch is text or data, or which of two sets of mapping registers is *active* (an ambiguous virtual address will be resolved in favor of the active set)]. Each of these two sets has basing registers for a text segment (I bank) and a data segment (D bank). Therefore, only four banks can be addressable at any one time. To make another bank accessible, its address and limits must replace those of a currently based bank in at least one of the mapping registers. This is done by an instruction, which may be executed by user programs as well as EXEC. The implications of this unusual memory management scheme are that:

1. Since segments are a scarce resource, numerous bank switches must be done to accomplish *UNIX* system kernel functions.
2. The ability to address multiple-user and kernel-user address spaces is limited.
3. Demand paging is not possible.
4. The bank-switch mechanism used for system calls is more efficient than the processor-state switch used by most machines.

4.2 Layered implementation

4.2.1 Advantages and constraints

The advantages of basing a *UNIX* system upon a vendor's standard operating system, rather than bare hardware, outweigh the disadvantages for the system's intended use as an integrated development environment. The system is widely marketable to 1100 customers since all eligible hardware runs the same operating system (EXEC), necessary EXEC changes are distributed and supported by Sperry, and no existing capabilities (transactions, etc.) are removed from a machine by installing a *UNIX* system.

The system functions as an integrated development environment supporting the C Transaction Environment (an internal product different from the *UNIX* system, and one not commercially available for part of the licensed package). This C Transaction Environment has a compatible system call subset, supporting transactions against a Data Base Management System (DBMS). The *UNIX* system has extensions that allow processes to access parts of the EXEC environment. EXEC files may be reached from within the *UNIX* system with special path names. A character device creates EXEC time-sharing sessions on virtual terminals. These sessions may communicate directly with the *UNIX* system user via a `cu`-like command. Shells using this feature contribute extensively to the ease of transport of programs from the development environment to the transaction execution environment. Access to the system from EXEC batch runs is also possible, which facilitates system administration by console operators not familiar with the *UNIX* system.

Implementation under the EXEC also imposes some constraints. The EXEC analog of a process is called an activity. A process maps to an activity, but an activity has no unique address space of its own, so the *UNIX* system kernel `fork` system call must manage the banks for each process after using an EXEC primitive to create an activity. EXEC groups activities into runs, which are normally but not invariably associated with a terminal. *UNIX* system process activities must span a collection of runs for performance reasons. Creation of an EXEC activity in another run is not possible, so there cannot be a single parent for all processes. A new run is created by each user logging in, which contains all of the processes created by that user. All system calls return results as if process 1 did exist. EXEC file assigns (used by block devices) are accomplished by a run. Each run of a group of runs desiring to assign a file must do so separately. Similarly, EXEC has an analog to signals among activities within a run but not among runs. Such sharing among runs requires a set of local daemon activities for each run to service the shared status data, adding nontrivially to the complexity of the kernel.

4.2.2 Exclusion

The use of exclusion primitives to protect shared kernel data is necessary not only to handle multiple processors without races, but on a single processor system as well, since user-level EXEC activities can be arbitrarily preempted in kernel code and resumed in an arbitrary order. The hardware provides instructions for this purpose; the *UNIX* system kernel uses EXEC primitives based upon those instructions that queue blocked processes to avoid excessive EXEC dispatcher traffic.

4.2.3 Block and character devices

There is only one block-device type (major). Each minor device number is mapped to a different file name in the EXEC file system. The complete file structure in a *UNIX* system is present inside one of these EXEC files. File system block size is 3584 bytes. This size, unusual in that it is not a power of 2, is due to constraints imposed by use of EXEC I/O and disk controller microcode. The I/O itself is done with EXEC primitives rather than channel programs to bare hardware. It is otherwise unremarkable; in fact, management of file assigns among multiple runs is a much more difficult problem.

Of the character devices, the terminal driver is the most interesting. The low-level portion of it is a set of real-time EXEC communications activities. The resulting terminal interface has complete *UNIX* system character processing capabilities; full-duplex and character editing functions are available without modifying or bypassing the EXEC, and without an external front-end processor. The character processing overhead incurred by not having a front end is noticeable but no worse than that incurred by users of the conventional 1100 time-sharing terminal interface.

V. THE *UNIX* OPERATING SYSTEM ON THE 3B20S MINICOMPUTER

The AT&T 3B20S minicomputer[7] is a 32-bit minicomputer that was originally designed and developed to be used in telephone switching systems. The switching version of the 3B20 minicomputer, known as the 3B20 Duplex or 3B20D minicomputer, has duplicated CPU, memory, and DMA hardware components. A 3B20D minicomputer can be easily converted into two independent simplex machines. The 3B20S minicomputer, a repackaged half of a 3B20D minicomputer, is being used throughout AT&T as a general-purpose minicomputer. The latest version of the 3B20 minicomputer, the 3B20A minicomputer, has the two processor halves reunited, working in parallel as a multiprocessor unit.

5.1 Hardware-related porting issues

5.1.1 Memory management

The 3B20 minicomputer employs a two-level segmented and paged memory-address translation scheme similar to that of the IBM 370. A virtual address is 24 bits long, pages are 2K bytes, segments contain 64 pages, and each address space contains 128 segments. The original 3B20 minicomputer kernel was derived from the *UNIX* System III VAX-11/780 implementation. Both were swapping systems; however, the 3B20 minicomputer system used segments for managing the address space of user programs, while the VAX* system used pages. Employing segments made the implementation of shared text and shared memory simple; shared data pages had a common page table mapped by the segment table of the processes involved. A software segment table paralleled the hardware segment table and described what each segment contained: text, data, stack, or shared data.

With the addition of demand paging to *UNIX* System V, 3B20 minicomputers and VAX machines running *UNIX* systems have been unified in memory-management design and implementation. Both systems use logical segments or regions of contiguous pages and page-table entries as their basis.

5.1.2 I/O system

Perhaps the most unusual feature of the 3B20 minicomputer is its I/O architecture. There are two major types of I/O device controllers: the Input Output Processor (IOP) and the Disk File Controller (DFC). Both types are coupled to the CPU and DMA through high-speed serial data links.

The IOP is constructed of two levels: The first level, or front end, performs maintenance and data concentration functions for the second level of up to 16 Peripheral Controllers (PCs). The IOP driver reflects the two-level structure of the hardware. A common driver performs all maintenance and communication functions, and uses a switch table to pass completion reports to PC drivers.

PC drivers are generally less or equal in complexity to drivers written for other machines. For example, the teletypewriter (TTY) PC driver's only function is to provide data buffering, while all the *UNIX* system character processing functions are implemented in the PC itself.

The Disk File Controller (DFC) interfaces with up to four Moving Head Disks (MHDs). The DFC can buffer up to 256 I/O requests, and optionally it will execute an elevator alogorithm to minimize disk head movement.

IOP and DFC drivers communicate with their device controllers through message queues contained in main memory. Each controller has at least two queues: a command queue where the driver puts I/O requests, and a report queue where the controller returns the status of

* Trademark of Digital Equipment Corporation.

I/O requests that have been completed. To request an I/O operation, the driver loads a message into the command queue. Next, the controller reads the message DMA, processes it, and then puts a request completion message into the report queue. All the PCs on an IOP share a single pair of message queues.

A feature of the 3B20 minicomputer is that each DFC, IOP, MHD, and PC unit can be powered off or physically disconnected while the rest of the system is still active. Each unit can be logically *in service* or *out of service*, and the two new user-level commands were created to support this feature:

`don` Restore device to service (device on-line).
`doff` Remove device from service (device off-line).

While a unit is off-line, it can be diagnosed and repaired if necessary, and then restored to service. About one half of the total IOP and DFC driver code is used to support these maintenance features. PC drivers do not contain any maintenance code, but they do contain code to handle *in service* and *out of service* command requests.

5.2 Architectural and software-related porting issues

The 3B20 minicomputer has a CPU architecture typical of most minicomputers: 12 general-purpose registers, an orthogonal basic instruction set with eight addressing modes, plus additional special-purpose instructions for moving data, manipulating strings, and performing I/O and maintenance functions.

5.2.1 Byte ordering

Many of the problems encountered when porting software to a new processor have to do with byte ordering. The 3B20 minicomputer has the opposite byte ordering of the VAX minicomputer. Carelessly written programs may not be portable between different execution environments. For example, this program fragment will produce unexpected results on the 3B20 minicomputer processor:

```
int c = 'A';
write(fd, &c, 1);
```

The wrong byte address is passed to the subroutine, and a null byte will be written.

A second more subtle difference between the VAX and the 3B20 minicomputers is that the latter requires data objects to be aligned on their natural boundaries.* This example will cause a processor trap on the 3B20 minicomputer:

* A *long* is a line on a four-byte boundary, and a *short* is a line on a two-byte boundary.

```
short  a[10];
int *p;
p = &a[1];
*p = 0;
```

The program is attempting to reference a word on an inappropriate boundary.

Both of the fragments listed above are examples of dubious programming practice. Fortunately, the *UNIX* system kernel is generally free of such flaws, and most user-level code had already been ported to the IBM 370[8], a processor that has the same byte ordering as the 3B20 minicomputer, before the 3B20 minicomputer effort started.

5.3 Development and test environment

The 3B20 minicomputer operating system was developed in a host/target environment. The host was a PDP-11/70 processor running the *UNIX* Real-Time (RT) operating system.* The link between the host and target was a 9.6K-baud asynchronous port. On the 3B20 minicomputer end of the link was a hardware debugging tool known as the Micro-Level Test Set (MLTS). From the MLTS, any bit or byte of the machine can be accessed even while the processor is running. The initial system debugging was conducted entirely through the MLTS.

5.3.1 3B20S minicomputer SGS

The PDP-11/70 host machine supported a 3B20 minicomputer cross-SGS, based on the now standard Common Object File Format (COFF). Operating system object files were down loaded into the target memory through the MLTS link. Until the SGS was ported to the 3B20 minicomputer, commands were transported to it from the host via magnetic tape. Producing 32-bit object files on a 16-bit processor is a difficult job; the SGS uses a software paging scheme to handle the difference in address space size. Once the 3B20 minicomputer system was stable and the SGS was ported to it, the symbolic debugger, `sdb`, was modified to use the COFF. The VAX system has since converted to the COFF.

5.3.2 Kernel debugging tools

Debugging an operating system kernel can be tedious. A common technique used for debugging is to insert print statements into the source so that the kernel can be tracked while it executes. The 3B20 minicomputer has no generally available nonprogrammable TTY I/O

* The *UNIX*-RT operating system is an updated version of the Multi-Environment Real-Time (MERT)[9] operating system, a variant of the *UNIX* operating system with real-time support.

device, like the DEC* KL-11. Messages cannot be written to a TTY until after the kernel has bootstrapped itself and a TTY PC has been brought into service via `don`. This deficiency makes low-level debugging with print statements impossible, but the problem has been turned around to produce an extremely valuable debugging tool. All kernel-generated messages are saved in a circular memory buffer and saved permanently in any memory dump for future reference. The same scheme has since been adopted for the VAX kernel.

A major milestone in bringing a machine to life is creating the first root file system. The first step was to create an empty file system. A version of the kernel with the make file system (`mkfs`) command built into it was created to do the job. A system call was invented to allow `mkfs` to open a file by major and minor device number rather than by name. The second step was to populate the file system. Again, a special system version was built to do the job. Only two commands were needed: some form of the shell and some form of file copy. At this point, a system with a rudimentary initialization process built into it was booted and the remainder of the file system was populated by copying files from magnetic tape. An important command to get working early is the file system checker, `fsck`. Needless to say, the above series of steps was repeated many times before the system was stable enough to check its own root file system.

5.4 Status

The first 3B20 minicomputer-based *UNIX* system was deployed in July of 1981. Since then, both the operating system and the hardware have matured greatly. For example, over a dozen new peripherals have been added, and the instruction set has been expanded to support the IEEE floating-point standard and C-style string manipulations. The system refinements take full advantage of the 3B20 minicomputer hardware and upgrade the standard *UNIX* system features for 32-bit machines. These changes include a 1K-byte block file system and demand paging. The more recent 3B20 minicomputer hardware and software development is a multiprocessor *UNIX* system.

VI. THE *UNIX* OPERATING SYSTEM ON THE AT&T 3B5 MINICOMPUTER

The AT&T 3B5 minicomputer is a 32-bit minicomputer based on the *WE*® 32000[10] microprocessor. Development of the *UNIX* operating system for the 3B5 minicomputer was started in 1980 at the same time that the requirements for the hardware and microprocessor were being

* Trademark of Digital Equipment Corporation.

finalized. To minimize the time between the first hardware introduction and an integrated hardware and software package, extensive use was made of simulation, emulation, and a cross-development environment.

6.1 3B family compatibility

The 3B5 minicomputer is a member of the 3B family and thus shares many architectural features with the 3B20. The main objective of the 3B family is to provide a very high degree of C language, user-level program compatibility among members of the family. The 3B5 minicomputer and 3B20 support the same data types and use the same, bit-for-bit identical representations for each type. That is, byte ordering, bit significance, alignment restrictions, etc., are the same on both machines. The two machines also share a common subset of assembler-level instructions called IS25. This subset is defined to include all instructions that can be generated by the C language compiler.

This high degree of C language software compatibility simplified porting major portions of the operating system software. For example, the 3B5 minicomputer could automatically take advantage of solutions to many of the subtle data representation or "byte-order" problems found during the 3B20 port. However, the machine-dependent portions of the operating system required significant design effort as a result of some of the unique architectural characteristics of the 3B5 minicomputer and the *WE* 32000. The major areas that needed change were memory management, process creation, interrupt handling, context switching, system call interface, and exception handling.

6.2 WE *32000 architecture and related porting issues*

The *WE* 32000 is based on a large, single address space, which contains both the operating system and a user program. External MMU hardware, through the checking of access rights, provides the basic protection mechanism in the 3B5 minicomputer. The *WE* 32000 contains only a few privileged instructions and privileged internal registers. In addition to the single-kernel single-user address space, the *WE* 32000 assumes the use of a single stack for both user and kernel execution (separate stacks are provided for such things as stack exception, I/O interrupts, etc.).

6.2.1 System calls

The system call instruction, `gate`, changes the processor to a privileged state and passes control to the operating system, but does not switch to a separate stack. Therefore, the system call interface had to be carefully designed to avoid the possibility of a security breach arising from the mixture of user and kernel data on the same stack. The system has to be careful that a stack address, of a buffer for instance, passed to it is indeed in the user's portion of the stack. Care was also taken to ensure stack exceptions cannot occur when running in the kernel mode. This is done by manipulating the stack bounds registers at the system call interface to guarantee that upon entry to the system, sufficient stack space is available to complete the system call. The system call code that handles signals to user programs also required change. Upon entry, sufficient space (two words) for processing signals is reserved on the stack. If a signal is present at the completion of the system call, this reserved space is set up with return information before control is passed to the user's signal handler.

The `fork` and `exec` system calls were also affected by the single stack architecture. Both of these system calls must manipulate the user's stack. This is difficult to do if the kernel code is also using the same stack. Code in the system call interface explicitly switches to a separate stack for these system calls.

6.2.2 Process concept

The *WE* 32000 includes a notion of a process by providing privileged instructions that can call a process and return to a previous process. Process-state information is kept in a Process Control Block (PCB) data structure. Interrupts are essentially hardware-invoked call process instructions. This process concept is used by the 3B5 minicomputer *UNIX* system kernel to support user processes and interrupt handling. Upon interrupt, a process is dispatched by the hardware. All interrupt processes are part of the kernel, and reside in the system address space. All interrupt PCBs and stacks are statically allocated in kernel space. Since interrupt processes are not allowed to suspend themselves, interrupt processes of equal priority can share the same stack. Therefore, only one stack is needed per interrupt priority level.

6.2.3 Process switching

When a process is to be switched out, the process switcher (`swtch`) sets a Program Interrupt Request (PIR) at priority-level 1 (the lowest priority interrupt level). Since a user process runs at interrupt priority-level 0, the level-1 PIR is honored before any other user-level process is executed. The *WE* 32000 saves the state of the user process in its PCB, and dispatches the switcher. The switcher then picks another process to run, sets up its map, and performs a return process instruction to transfer control to the new user process.

6.2.4 Memory management

The MMU used for initial 3B5 minicomputer development supported a 24-bit segmented logical address space and supported virtual to physical address translation based on contiguous segments. A user process's virtual address space is divided into two equal address subspaces, the system space and the user space. A user process running in kernel mode has access to both address subspaces, whereas a user process in user mode only has access to the user address subspace.

Translation from a virtual to a physical address is done via map buffers. A total of 64 maps are supported by the memory management unit. The system space, by convention, is mapped through map 0. The system space is common to all user processes and is not affected by a context switch. The 3B5 minicomputer *UNIX* system kernel resides in the system address space, and all operating system functions are shared by all processes and are accessed via the *gate* mechanism by user processes. An address in user space is translated by using the map specified by an "active process ID" register. The operating assigns maps to user processes, and if more than 63 processes are in main memory, the maps are time shared.

To ease the sharing of the 63 maps among user processes, a new entry has been added to the process table to hold the map index if a map is assigned to the process. When a process is scheduled to run, the switcher determines if the process currently has a map assigned. If so, a switch to a user process' address map only requires reloading the "active process ID" register. If not, the switcher must allocate a map entry and load the process' map from its "*u*" *area* into the memory management unit. The switcher will either allocate a free map or, if all maps are in use randomly, deallocate a map owned by a sleeping process. A map is freed when a process is terminated or swapped.

6.3 Development and test environment

6.3.1 3B5 minicomputer SGS

Since no 3B5 minicomputer hardware existed at the time the project began, it was impossible to develop software for the 3B5 minicomputer using the native machine. A cross-software generation system based on the common SGS was developed and run on a VAX-11/780 processor. The cross-SGS included a C compiler; assembler; linker and associated support programs; and generated *WE* 32000 object code.

6.3.2 Emulation and debugging tools

The initial 3B5 minicomputer development strategy was based on the use of an emulation for developing virtually all of the software. An AT&T 3B20S minicomputer was microcoded to emulate the *WE* 32000 microprocessor and the 3B5 minicomputer. This emulation included the interrupt controller, programmed interrupts, memory management, central control Universal Asynchronous Receiver/

Transmitter (UART), Asynchronous Data Link Interface (ADLI) UARTs, Sanity and Interval Timer (SIT) and the Integrated Disk File Controller (IDFC). From the perspective of a developer, the emulation was an actual 3B5 minicomputer.

During the emulation stage, a 3B20 emulation control program was used as a debugging tool. The 3B20 minicomputer Emulation Control Program (MIP) executed on a support PDP-11/70 processor and provided a means to control the 3B5 minicomputer emulation microcode. Features included the ability to start and stop the emulation, load emulation memory from a file on the support processor, set and display registers and memory, and set emulation breakpoints.

Emulation program commands were bundled to form a debugging package. This package, which is not as yet an official AT&T product, and its related command language, referred to as the DEMON (DEbugging MONitor) monitor, served as the interface between the developer and the emulation program. DEMON provided debugging facilities comparable to the emulation control program. In addition, DEMON provided a single-step program debugging capability, and the ability to examine memory using physical or virtual addresses. A dedicated RS-232 link was used to provide down-load/up-load capability from a support processor. Once the 3B5 minicomputer hardware was available, a ROM-based version of the DEMON monitor was developed permitting stand-alone debugging.

6.3.3 Initial file system

Since the 3B5 minicomputer uses a disk drive, which was not supported on other processors, it was necessary to develop a method for creating the initial file system for a 3B5 minicomputer. A driver was developed that treated a block of memory as though it were a disk—an in-core file system. The cross-`mkfs` program was created that would build a file system image within a normal file. Once this file was loaded into emulation memory by either DEMON or the emulation control program, the in-core file system driver would access the data as though the data were the individual blocks of a file system. This technique had the additional advantage of providing access to a file system before a functioning disk driver was available. The in-core file system facility has proved to be a convenient way to move information from a support machine to a 3B5 minicomputer and continues to be used for that purpose.

6.3.4 The ultimate test

The *UNIX* operating system developed in the emulation environment was successfully running on actual 3B5 minicomputer hardware in less than a week after its arrival. This success confirmed the importance of using an emulation environment to port the *UNIX* operating system to a new processor without having the actual hardware.

6.4 Status

The 3B5 minicomputer has been available since October 1982. It has evolved through two releases. The current release includes the latest version of the *UNIX* operating system, as well as support for a wide range of peripherals. In the future, 3B5 minicomputer will continue to track standard *UNIX* operating system releases and increase the variety of supported peripherals.

6.5 Acknowledgments

The original 8086 *UNIX* system development was done by Tom Blumer and Ralph Muha. The authors gratefully acknowledge the contributions of C. H. Elmendorf and R. A. Kasten in preparing the 3B5 minicomputer section of this article.

VII. CONCLUSION

As technology advances, more processors will be introduced and software developers will be forced to adapt current software packages to fit new environments. In these situations the need for portable software is essential to maintain a familiar user environment. As evidenced by the previous sections, the *UNIX* operating system and its associated user-level programs have proven, and continue to prove, to be extremely portable. Because portability is a fundamental part of the *UNIX* system philosophy, the *UNIX* operating system can be made to adapt to the diverse computing environment that results from continuous technological advances.

REFERENCES

1. D. M. Ritchie and K. Thompson, "*UNIX* Time-Sharing System: The *UNIX* Time-Sharing System," B.S.T.J., *57,* No. 6, Part 2 (July–August 1978), pp. 1905–30.
2. K. Thompson, "*UNIX* Time-Sharing System: *UNIX* Implementation," B.S.T.J., *57,* No. 6, Part 2 (July–August 1978), pp. 1931–46.
3. S. C. Johnson and D. M. Ritchie, "*UNIX* Time-Sharing System: Portability of C Programs and the *UNIX* System," B.S.T.J., *57,* No. 6, Part 2 (July–August 1978), pp. 2021–48.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Englewood Cliffs, NJ: Prentice Hall, 1978.
5. *MCS-86™ User's Manual,* Intel Corporation, July 1978.
6. G. Ronkin, "*UNIX* Time-Sharing System for *UNIX* 1100 Series Systems," Bell Laboratories, September 1981.
7. T. F. Arnold and W. N. Toy, "Inside the 3B20 Processor," Bell Lab. Rec., *59* (March 1981), pp. 66–71.
8. M. J. Bach and S. J. Buroff, "The *UNIX* System: Multiprocessor *UNIX* Systems," AT&T Bell Lab. Tech. J., this issue.
9. H. Lycklama and D. L. Bayer, "*UNIX* Time-Sharing System: The MERT Operating System," B.S.T.J., *57,* No. 6, Part 2 (July–August 1978), pp. 2049–86.

10. A. Berenbaum, M. W. Condry, and P. M. Lu, "Operating System and Language Support Features of the BELLMAC 32," *SIGPLAN 17,* No. 4 (April 1982), pp. 48–56.

AUTHORS

Thomas F. Houghton, B.S. (Electrical Engineering), 1978, University of Delaware; M.S. (Computer Science), 1981, Northwestern University; AT&T Bell Laboratories, 1978—. Since joining AT&T Bell Laboratories, Mr. Houghton has worked in several areas of the *UNIX* operating system. He was part of a team that ported the *UNIX* operating system to an Intel 8086-based processor, worked on *UNIX* operating system performance characterization, and is currently working on *UNIX* operating system file system enhancements. Mr. Houghton also worked on the design of some *UNIX* RT operating system features. Member, Eta Kappa Nu, Tau Beta Pi, ACM, IEEE.

Keith A. Kelleman, B.S. (Electrical Engineering), 1979, Lafayette College; M.S. (Computer Science), 1981, Stevens Institute of Technology; AT&T Bell Laboratories, 1979—. At AT&T Bell Laboratories, Mr. Kelleman has been involved with *UNIX* system development. He is currently working on the development of a demand paged kernel for the *UNIX* system. His previous assignments were to develop a *UNIX* system for the AT&T 3B20 computer and to convert the *UNIX* system RJE facility to the virtual protocol machine.

George Ronkin, B.A. (Mathematics), 1974, Beloit College; M.S. (Computer Science), 1976, University of Wisconsin-Madison; AT&T Bell Laboratories, 1977–1984; present affiliation Bell Communications Research, Inc. While at AT&T Bell Laboratories, Mr. Ronkin worked on mainframe *UNIX* system implementation and advanced *UNIX* system architecture. He is currently a Member of Technical Staff of the Data and Video Services Research Division, Bell Communications Research, Inc., with interests in distributed programming environments and file system design for optical storage. Member, ACM.

Edward P. Schan, B.M.E., 1965, University of Louisville; S.M. (Information Sciences), 1968, University of Chicago; AT&T Bell Laboratories, 1965—. Mr. Schan is a member of the Small Processor Software Development Department. He has been involved in the architecture and operating system design of the AT&T 3B5 computer. He has also worked on other members of the 3B line of computers. Member, ACM, IEEE Computer Society.