The First Unix Port

Richard Miller
Miller Research Ltd
r.miller@acm.org

1 Prelude

The story of the first Unix port begins with a letter I received in April 1976 from Professor Juris Reinfelds, head of the newly created Computing Science Department at the University of Wollongong, Australia. At the time I was a fledgling systems programmer at SDL, an Ottawa computer service bureau which provided a remote batch and timesharing service on large IBM mainframes for clients in Canada and the U.S. I was preparing to join Juris later that year as the department's second member of staff, to set up and run a laboratory for software teaching and research.

Juris's idea of creating an interactive software laboratory, using timeshared terminals connected to a departmental minicomputer, was a progressive one for the time. The usual experience for students in computing was to queue up at a university computer centre to submit their programs and data on punched cards, and return some hours later to collect the printed output. Being constrained to two or three attempts per day at compiling and running a program may have encouraged a more disciplined approach to programming than is common nowadays, but it strictly limited the size of projects which students could be expected to tackle, and made it difficult to teach computing as an experimental science.

The plan had been to buy a DEC PDP-11 minicomputer to run Unix, a novel operating system from Bell Labs which was acquiring a cult following at universities around the world. Juris was impressed by the positive experience of the nearby University of New South Wales in Sydney, where Unix was used enthusiastically in programming courses as well as being itself an object of study in an Operating Systems course taught by John Lions. Juris had sent me a copy of Ritchie and Thompson's 1974 CACM paper *The UNIX Time-Sharing System*, which I found tremendously exciting. After the dispiriting experience of trying to use (and debug) early versions of IBM's TSO timesharing system, the simplicity and elegant minimalism of Unix seemed a brave new world indeed.

But Juris's letter brought a disappointment: the price of a PDP-11 was out of reach. Instead, he wrote, "unless you voice your objections loudly we will probably have an Interdata 7/32 with 10Mbyte disk and 192kbyte core as our lab computer... Since Interdata assembler is like IBM's it should be to your liking." As Unix was a PDP-11 operating system, no PDP-11 meant no Unix; we would have to build our software laboratory using Interdata's own OS/32 operating system.

2 The Interdata 7/32

I began work at Wollongong, and soon afterwards the Interdata minicomputer was installed: a 32-bit machine with a peak execution speed of somewhat less than 1 million instructions per second, along with seven "glass teletype" terminals, and a daisywheel typewriter which served as both system console and printer.

It did not take long to realise that OS/32 was not going to help us to build the kind of interactive system we wanted. Although it was a multi-tasking system, able to execute several programs at once, it could not really be considered to be multi-user or even timesharing in the usual sense. The terminals were strictly passive peripheral devices: you could run a program using a specific terminal for input and output, but programs (including system utilities like the text editor and compiler) could only be started from the central system console. If something went wrong with a program, the error messages would appear only on the system console. There was no concept of user names or logging in, no file protection of any kind, and a single linear directory of files on each disk device — a recipe for confusion with thirty students all wanting to call their first program `LESSON1.BAS`.

Clearly something had to be done before the Interdata would be usable in a student environment. As an emergency measure, I cobbled together a proto-timesharing system to run on top of OS/32, which I modestly named MOTH (My Own Terminal Handler). Essentially this was a simple multi-user shell, which would prompt for a user login at each terminal, and then read and execute simple commands on the user's behalf, with input and output automatically directed to the user's terminal. (I can't recall whether I succeeded in redirecting error messages as well.) Some measure of file protection was obtained by intercepting system calls from user programs, transforming `create` and `open` requests to make file names unique to each user. There was also a simple (voluntary) queuing mechanism which could be used to request exclusive use of the console printer before sending output there.

With MOTH in place, the first programming class were able to use the lab for their assignments. But the service was far from ideal. MOTH's control was not complete: for example, a program which entered an infinite loop (not uncommon in student programs), could not be interrupted but would have to be cancelled from the system console. OS/32 at that time had only a primitive memory allocation scheme with fixed-size partitions to share our 192 kilobytes of memory among seven users, with no capability of swapping to disk — it was very cramped. Among other quirks, the Interdata would crash instantly if the system console was taken offline. Since the console was our only hardcopy device, this happened frequently as students attempted to eject a page to remove their printout.

As our frustration with the OS/32 environment grew, Unix looked more and more attractive. Could we possibly return to the original plan of using Unix, even without a PDP-11? Why not attempt to make Unix run on the Interdata?

In the case of a typical computer manufacturer's operating system, the idea of portability to a new machine would have been preposterous. Conventionally operating systems were written in assembly language for a specific machine, exploiting low-level details of the hardware architecture which were often exposed to the user. For example in OS/360, disk files had to be allocated by the user in units of tracks and cylinders, and for efficient access one could write "channel programs" to be executed directly by the autonomous hardware which controlled the disks.

But Unix appeared very different. It was written in C, a (fairly) high level language for which compilers already existed for other machines besides the PDP-11. Also, Unix's device-independent input/output model presented a simplified and abstract view of the underlying hardware, which suggested machine independence as well. A Unix program writes a string of bytes onto a disk file or a magnetic tape or a terminal screen using exactly the same simple system call; it should be possible to write to an Interdata disk or a PDP-11 disk with equal indifference.

There seemed to be no reason in principle why Unix could not be adapted to a different machine. So in late 1976 the University of Wollongong ordered a source code licence for Unix (6th Edition), and I took on the project of making it run on the Interdata.

3 Porting the C Compiler

As most of the Unix kernel and nearly all the commands and utilities were written in C, the first task was to produce a C compiler for the 7/32. The obvious starting point was Dennis Ritchie's PDP-11 C compiler, which was itself written in C. I spent a few weeks learning the C language by reading a printed copy of the compiler source, and unravelling its structure. There was no internal documentation and the comments were sparse, but the clear and concise style made it quite easy to find one's way around. The compiler was in two sections: a syntactic front end which was fairly machine independent, and a code generating back end which fortunately was mostly table driven. I devised a new set of tables with code generation templates for the Interdata's instruction set, and wrote new versions of routines which performed storage layout, analysis of addressing modes, and other machine dependent functions.

I now had (I believed) an Interdata C compiler in C source code form, but I needed a running Interdata C compiler before I could compile it. This seeming paradox appears whenever a compiler is written in the language which it is meant to compile, and its solution is known as bootstrapping (a term which refers to lifting oneself up by one's own bootstraps). In this case I planned a half bootstrap, involving a second computer: I would use the PDP-11 C compiler running on a Unix machine at the University of New South Wales, to compile my Interdata C compiler.

On December 14 1976, I made the 80 kilometre journey from Wollongong to Sydney with the proposed changes to the C compiler on a deck of punched cards, and spent two days editing, compiling and debugging. This was my first actual experience as a user of Unix, but with some guidance from local gurus Peter Ivanov and Ian Johnstone, I soon felt at home. I quickly learned about the Unix principle of "trusting users to know what they are doing", when I mistyped a filename and inadvertently deleted several hours' work. But by December 16 the compiler was able to translate a handful of short test programs into what looked like correct Interdata assembly language. Using the compiler to translate its own source code then resulted in many thousands of lines of assembly code, which I copied onto a magnetic tape to take back to Wollongong.

At this point there was a logistical problem: our 7/32 had no tape drive. The Wollongong University computer centre had a Univac 1106 mainframe with a tape drive, but that was not compatible with PDP-11 tapes. Fortunately the Sydney office of Interdata knew of a client who could do tape conversions. So the translated compiler was copied there to a Univac-compatible tape to be taken to Wollongong, where it was loaded onto the computer centre's 1106, transmitted to an Interdata 8/16 (a smaller Interdata machine which was used as a front end communication processor for the Univac) and copied to a 5 megabyte removable disk pack which was carried upstairs and loaded onto the Computing Science Department's 7/32.

The assembly language version of the C compiler could now be run through the Interdata assembler to produce an executable program. Well, that was the theory: in fact, what it produced was pages of assembler syntax error messages. The compiler's own source code contained many new C constructs which my trivial test programs had not exercised, so there were still mistakes in the code generator. Another week's work on the compiler source followed by a second trip to Sydney on December 22 for cross-compilation resulted in a new tape full of assembly language which was brought back to Wollongong via the same tortuous route as before — and this time it passed successfully through the assembler.

One further thing was needed. Because the C compiler was originally a Unix program it made use of Unix system calls to perform input/output and memory allocation. While I was working on the compiler, a library of OS/32 routines to emulate some basic Unix system calls (open, close, read, write, malloc, and a few others) was being written by Ross Nealon, a Wollongong computing student. (Years later, Ross took over my place in the department when I left Wollongong. He died of cancer, tragically young, in 1988.)

Combining the assembled compiler with Ross's Unix compatibility library gave us, at last, an Interdata C compiler running on the 7/32. It was still far from correct, but it could now be maintained on our own machine without further trips to Sydney. Over the Christmas break I continued testing and correcting the compiler. Each time I made a change to the assembly language code, I would also make the corresponding change in the C source, which when eventually compiled would generate exactly the assembly source which I had just written by hand.

After many iterations, on January 5 1977 the compiler could successfully compile itself, producing assembly output identical to the running compiler.

4 Unix Tools on OS/32

The C compiler had been moved by a half bootstrap procedure, with an initial cross-compilation on another machine. Had a PDP-11 been available in Wollongong, I would doubtless have carried out a similar half bootstrap of the operating system, using the existing Unix system as a development environment to maintain and cross-compile a kernel, for downloading and testing on the 7/32. But if a PDP-11 had been available, I would not be doing the port in the first place! The 80 kilometres separating us from the nearest PDP-11 seemed to mandate a full bootstrap of the operating system, performing all the development on the target machine itself.

Before starting on the kernel, it seemed worthwhile to get some of the Unix development tools working on the Interdata. The system call emulation library, which enabled the Unix C compiler to run on OS/32, was extended a bit to support other simple Unix programs such as the editor `ed` and file manipulation and comparison commands. Having these available would be a significant aid to productivity in comparison with the existing Interdata tools. (The standard OS/32 text editor, for example, was only able to move sequentially through a file in one direction; after examining or modifying a line, one could not move to an earlier line of the file without exiting from the editor and beginning again.)

Porting Unix tools to the Interdata was a way of exercising more of the C compiler and increasing confidence in its reliability, and helped to point out compatibility problems in C semantics between the PDP-11 and Interdata. Although in theory a user level C program can be entirely machine independent, in the 6th Edition system rather a lot of shortcuts had been taken, with assumed knowledge of properties of the PDP-11 which failed on the Interdata. The most common of these were word size, the order of bytes within words, and sign extension of characters. I learned to be suspicious of any occurrence of the number 2 in a C program — more often than not, it really meant `sizeof(int)` which on the 7/32 should be 4.

5 The Unix Kernel on OS/32

An operating system like Unix can be decomposed into three layers. At the top is the interface defined by the repertoire of system calls which provide operating system services to user programs. In the middle are the mechanisms which implement these services and manage the sharing of resources among processes and users of the system. In Unix this layer is largely hardware independent; for example, all direct access devices are treated in an abstract way as indexed collections of fixed-size blocks. At the bottom are device drivers, interrupt handlers and other low-level routines which deal with the idiosyncrasies of actual hardware behaviour, and support the simpler view of idealised hardware used by the layer above.

Because each layer depends on the one below, it would appear that a new implementation of Unix would need to be built from the bottom up. But the 7/32 was not a bare machine: it already had an operating system, and in a sense the top layer of Unix was already working above OS/32 in the form of our library of emulated system calls. There was a compelling attraction to the idea of continuing from the top down, beginning with the easier task of porting and testing the middle layer over an idealised hardware interface provided by the facilities of OS/32, and leaving the complexities of the real hardware until the last.

Here I was influenced by my experience at SDL, where IBM's VM/370 operating system had been used to test multiple versions of OS/360 concurrently on the same machine. VM used the IBM 370 paging facilities to divide the computer into several disjoint virtual machines, each able to run its own operating system with the illusion of having complete control of the real hardware. Execution of the virtual machines was timeshared, with privileged instructions being intercepted and emulated by the VM kernel. The virtualisation was so complete that a VM virtual machine could be used to run another instance of VM itself.

OS/32 was not quite as sophisticated as VM/370, but it did have some useful facilities for real-time programming which could be exploited to build a kind of virtual machine environment. Like the 370, the Interdata had a Program Status Word (PSW) with an instruction pointer and control bits for enabling and disabling interrupts and memory protection and relocation. An interrupt or system call instruction would cause the old PSW to be stored in a fixed location in low memory, and a new PSW (typically with interrupts disabled) to be loaded from a table indexed by the type of interrupt.

Under OS/32, each program's address space began with a reserved area called the UDL, containing system information about the running program. This included things like pointers to open files, and a Task Status Word (TSW) which was analogous to the PSW. If a program requested an asynchronous input/output operation, the operating system would signal its completion by a TSW exchange, loading a new TSW and instruction pointer from a table in the UDL which was analogous to the hardware interrupt table in low memory. An OS/32 program which was declared at link time to be an E-task (executive task) had extra privileges, and could define its own system calls by specifying that certain system call instructions would load a new TSW from the program's own UDL instead of the kernel's table.

These mechanisms provided most of what was needed to emulate the lowest layer of Unix — asynchronous device I/O, process switching, interrupt handling, and system calls to switch between user and kernel mode — by manipulating TSWs within a single OS/32 partition. The missing mechanism was control over memory management, which I had to supply by trickery. As an E-task, the virtual Unix kernel had access to the 7/32's memory management registers, so it was able to perform its own address relocation, swapping multiple Unix processes in and out of its memory partition while maintaining the illusion of being a single OS/32 process.

Using this simple virtual machine environment on OS/32, porting the top and middle layers of Unix proved fairly straightforward. Nearly all the changes required were due to trivial word size and byte order problems — the program logic remained almost unaltered. Developing and testing a virtual version of Unix on OS/32 had practical advantages. There was no need for exclusive use of the machine; using MOTH the system could be used by students at the same time as my kernel testing. And the OS/32 interactive debugger was available for breakpointing and single-stepping through the Unix kernel just like any other program.

6 Unix Stands Alone

By February, it was possible to log in to a terminal assigned to the virtual Unix and run (very slowly) a Unix program. One of the 7/32's two 5 megabyte disk drives was initialised as a Unix file system, by writing a data-only assembly program whose structure was the image of an empty file system, assembling it, and copying the binary result to the disk. The file system was gradually populated with Unix commands, each one being compiled and assembled under OS/32, converted from OS/32 object to Unix `a.out` format, and moved to the Unix disk. At last it was time to perform the conjuring trick of whipping OS/32 out from underneath, leaving Unix to stand on its own.

Writing device drivers for the Interdata peripherals (disk drive, serial interface, clock) was challenging as expected: real hardware, unlike the virtual kind, is always full of surprises. I was able to use PDP-11 drivers as a pattern, but the Interdata architecture was rather more complex. Communicating to the disk drive, for example, involved a disk controller and DMA channel as well as the drive itself, all needing to be programmed separately and in the right order, and all generating separate interrupts (or not, depending on timing conditions). Fortunately the top-down approach meant that I was not trying to test drivers and interrupt handlers on a bare machine: I had all the rest of Unix, already running and stable, to use as a test bed. On April 28, Unix was running in full control of the 7/32.

Of course, the first successful “Hello, world” is not the end of the story. Until this point the C compiler was still running on OS/32, because the OS/32 assembler was needed to assemble its output. When I wrote a new Interdata assembler for Unix (in C), I was gratified to find that it was smaller and much faster than Interdata's own assembler (written in assembly language) — a victory for the principle of systems programming in a high level language. Many other Unix commands and utilities needed compiling and testing, and most had the odd trivial PDP-11 dependency to be corrected. Another three months of work were required until 25 July 1977, when OS/32 was retired and Unix formally became the production system on the Wollongong 7/32.

7 The 7th Edition

When Unix first began to run as a stand-alone system on the Interdata we contacted Bell Labs, expecting them to be surprised to hear that their operating system was portable. In fact there was a surprise on both sides: a team at Bell Labs was in the midst of doing their own port of Unix to an Interdata 8/32 (a slightly more powerful 32-bit minicomputer). They had begun work at the beginning of 1977 in anticipation of the delivery of their machine in April, and had a kernel working by June — less than two months after the Wollongong kernel first ran on the bare 7/32.

The Bell Labs Interdata port was never released to the public, but it became the basis for the 7th Edition of Unix, which was specifically engineered to be portable (with Steve Johnson's new portable compiler, and a much more disciplined approach to data typing in C). When the PDP-11 version of the 7th Edition became available, the University of Wollongong ordered a copy, and I ported that to the Interdata, with some assistance from Robert Elz at the University of Melbourne who had been using the Wollongong 6th Edition system on an 8/32. The project was somewhat easier the second time around, but there was still a lot of work because the kernel had already begin to grow bigger and more complex, and there were many new commands.

The appearance of the 7th Edition of Unix began a trickle and then a flood of porting projects in many places to machines of all kinds; by the early 1980's there were at least three companies making a full-time business of porting Unix. By the 1990's no computer, from micro to mainframe, could be considered to be a serious machine if it wasn't able to run some form of Unix. Now, twenty-one years after the first port, it's difficult to believe that there once was a time when operating system portability was an audacious idea.