UNIX

UNIX is a general-purpose, interactive, time-sharing operating system originally invented in the 1970s at Bell Labs. There are many varieties of UNIX in current use, both free and commercial, but they all have well-integrated, standard, networking and graphics systems and provide a rich environment for programming, document preparation, and scientific computing. UNIX has been ported to more different types of hardware than any operating system in history (1). Because these systems have to function uniformly over networks populated by a bewildering variety of hardware types and different UNIX versions, the software is usually careful about providing hardware-independent binary data. As trends in computing hardware have changed from minicomputers, to vector supercomputers, to distributed networks of workstations and PCs, UNIX has evolved to meet the challenges.

IN THE BEGINNING

UNIX was originally invented in 1969 by Ken Thompson on a surplus DEC PDP-7 at AT&T's Bell Labs. It was modeled after the Multics operating system, which introduced many new concepts such as symmetric multiprocessing, a hierarchical file system, access control, and virtual memory. The name UNIX is a pun on Multics (Multiplexed Information and Computing Service), replacing "Multiplexed" by "Uniplexed," as it was originally just a simple single user system. Dennis Ritchie, who created the C programming language, joined Thompson and rewrote UNIX almost entirely in C during 1972 to 1974 (2–3). As far as we know, this made UNIX the first source-portable operating system in history. Undoubtedly the best single reference on the early days of UNIX is the famous July-August 1978 issue of the Bell System Technical Journal. This issue includes articles by Ritchie and Thompson (4) on the design and implementation of UNIX as well as a dozen others on the history of UNIX (5), the UNIX shell (6), text processing (7), language development tools, and more.

Computers in the 1970s were big, expensive systems with proprietary hardware and software. Vendors sought to lock customers into a particular family of machines. Command interfaces, system calls, etc. were complicated and uncorrelated from vendor to vendor. UNIX 7th edition (then called "Version 7") became available for DEC hardware (especially the popular 11/70 family of minicomputers) from AT&T in the late 1970s. The system came as a single tape, with all source, and for the cost of copying and issuing a license at about $300. The system came with no support, no user contacts, and no promises.

In the beginning UNIX was licensed almost for free to universities, including the full source code. This led to its immense popularity in academic environments. But with the release of Version 7, the licensing policy became more restrictive and the source code could no longer be used for teaching (1). One year later, in October 1980, BSD (Berkeley Software Distribution) UNIX 4.0 was released, again freely available to the public. It was based on the older free sources of UNIX and further developed at the University of California at Berkeley. It included many enhancements like paged virtual memory and TCP/IP networking. Several commercial versions were derived from this, including SunOS and ULTRIX (produced by Sun and DEC, respectively).

From the technical user's point of view, and apart from considerations of academic fashion, UNIX offered a number of very appealing features:

1. Every file is simply a stream of bytes. This sounds obvious in retrospect, but at the time, a typical operating system (OS) had all kinds of extra file attributes, such as record type and length. That complexity made it difficult for user programs to deal flexibly with files, particularly binary files of different types.
2. Devices are files. Each OS then had various utilities for doing input/output (IO) to devices. Merging devices into the file system had a number of beneficial results:
   - There were not as many system calls to remember, although the ioctl() system call expanded to take up some of the slack in that area.
   - Programs gained additional functionality (the ability to handle IO directly to/from devices) with little or no additional programming effort.
   - The number of specialized system utilities decreased accordingly.
3. It has a process model that is easy to use. The possibility to link the input and output streams of programs via pipes (buffers allowing asynchronous communication between processes under control of the kernel) had a great impact on how we write programs, even in technical work. It also led to quiet programs since unnecessary output could prevent a program from being used as a filter in a pipeline.

   UNIX reduced the command interpreter, the shell, to just another user program, a change that much simplified the formal structure of the OS and led to the present proliferation of shells (overall a good thing). It also gave us convenient subprogram execution as a programming tool.

4. There was an inspired set of utilities. The core set was a then-unusual group of single-purpose programs such as sed, uniq, and tr. These made it possible to write the first spelling checker as just a pipeline of existing utilities, to serve as excellent examples to utility writers (which, in the end, much of UNIX programming is), and to give rise to the first entry of the trilogy of UNIX programming methods used to attack a problem—write a shell script, write a program, write a language.

   The distribution also included trof/eqn/tbl for typesetting (in fact, the famous 1978 issue of the Bell System Technical Journal was typeset using troff), as well as lex, a lexical analyzer, and yacc, a parser generator.

The possibilities contained in the UNIX utility set took a while to comprehend (because they were so different in structure from the large multifunction utilities with other operating systems), but once seen, they were inspiring and empowering.

The emergence of this powerful, and accessible bundle of capabilities showed a future rich with possibilities we had never seen before: an end to the mind-numbing proliferation of operating systems and their utilities and the emergence of a powerful, simple, user-oriented computing environment. All of this came about because the times were ready, and Bell Labs had one of those clusters of brilliant people that occur from time to time in every field. It was also because the international UNIX community had created a large and growing pool of freely-available software that will typically run with only minor changes on almost any UNIX system. This pool includes all of the components for software development (mostly from the efforts of the Free Software Foundation, http://www.fsf.org/, which is an independent group promoting the proliferation of free software with open source code) and some very powerful data management, manipulation, and display programs.

The current operating system is many times larger than Version 7 (mostly, for good reason), but its basic design and power are intact. Today, much of what has been described as a UNIX system has been fixed in the POSIX (Portable Operating System) standards, further improving portability. These IEEE standards define a UNIX-like interface (8), the shell and utilities, and real-time extensions (9).

In 1992, another major branch came into existence: Linux. It started as the hobby project of Finnish student, Linus Torvalds. Torvalds created a UNIX-like kernel for the Intel 80386 processor and released it to the public on the Internet. Others subsequently extended this kernel, drawing on the vast software resources from the UNIX world. By using the software which had been developed by the Free Software Foundation's GNU project and adding on the X-Windows interface provided by the XFree86 project, Linux was turned into a full featured major new UNIX system. The original UNIX tapped a rich vein of discontent and bottled-up creativity among technical users. Linux has done something similar in addition to capitalizing on the work of the Free Software Foundation. Today GNU/Linux has been ported to a wide variety of hardware and has turned out to be one of the most rapidly evolving operating systems ever.

For the price of a paper-back book, you can get the complete operating system, with all the bells and whistles (the kernel and all utilities, the X11 windowing environment, text processing software, compilers/debuggers, editors, file managers, and so on), along with all of the source code. This allows anyone to take cheap, ubiquitous hardware and build robust and reliable multiuser workstations that don't suffer the drawbacks of many commercial PC-based operating systems (frequent reboots, poor scheduling under load, weak security, and a single-user mentality). This democratization of desktop computing may ultimately be the most enduring legacy of the experiment begun at Bell Labs over 30 years ago.

CONCEPTS

Kernel

The kernel is the heart of an operating system. It provides the minimum functionality which defines it. Everything else is an add-on provided by external programs. One can, therefore, say that the kernel *is* the operating system.

The kernel provides the lowest abstraction layer of the underlying hardware. It is responsible for resource management and process scheduling. Traditionally, UNIX kernels included the device drivers for the hardware, all networking code, and all filesystem code. A change of drivers usually required recompiling the kernel and rebooting the system. Recent UNIX kernels are modular, so parts of them can be exchanged at runtime without rebooting.

Kernel Structure

One distinguishes between monolithic kernels and kernels with a message passing architecture. A monolithic kernel is compiled into one large binary file in which all its parts are accessible to all other parts, so that one kernel function can easily call any other one. While this provides minimum overhead, it can be unstructured and inflexible.

In a kernel with a message passing architecture, the kernel functions do not call each other directly, but send messages to each other, asking for certain operations to be performed. The same applies to user mode programs calling the kernel; the actual kernel call is performed by a library function which constructs the message required and passes it on to the kernel. This creates additional overhead and, therefore, is potentially slower than a monolithic kernel. Strictly speaking, a kernel with a message passing architecture could be monolithic as well, by being compiled into one binary, but still using messages to communicate between its parts. However, this makes relatively little sense, for it does not take full advantage of the message passing architecture.

The main advantage of a message passing architecture is that it can easily be split up and that operations can be delegated to external programs, device drivers, or other modules loaded at runtime. This makes it possible to build distributed systems that cooperate across multiple networked computers.

Microkernels

A common implementation of the message passing architecture is the microkernel. A microkernel provides only the minimum functionality of task switching and memory management. Everything else, including device IO and filesystems, is handled by external processes, so-called servers, which run outside the kernel. This improves flexibility, as servers can be changed or restarted at any time. It also improves security because the servers do not necessarily run in kernel mode but can run as normal user mode processes with fewer privileges.

All communication with these servers is done through the kernel's message passing system which routes the messages to the appropriate server. Such a microkernel is extremely small and easy to port to different hardware architectures. A popular example of this is the MACH microkernel developed at Carnegie-Mellon University (MACH homepage: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html).

As most of the system's functionality which defines its API (Application Program Interface) is provided by processes running outside the microkernel, such a system can provide different operating system personalities. This is different from emulation, where one system gets simulated by another one. A microkernel can truly run multiple operating systems on the same hardware at the same time.

Hardware Abstraction

Normal programs run in user mode and have no direct access to the hardware. It is only through the kernel, which runs with special privileges, that they can access the hardware. To do so, they call the kernel to perform the required operation. The changes between user mode and kernel mode are called context switches and are generally quite expensive in terms of computation time. Every context switch involves saving all the processor registers, passing the parameters to the kernel, and calling the kernel function. To avoid unnecessary context switches, programs frequently use buffered IO as provided by user level libraries.

Processes

A process is an executing program, including its program counter, CPU registers, and variables. On a UNIX system, each process has its own virtual CPU and address space. The kernel is responsible for switching the existing CPU(s) between the waiting processes. As UNIX provides preemptive multitasking (as opposed to cooperative multitasking), the processes do not need to explicitly release the CPU but get scheduled by the kernel. Processes carry different priorities which allow the scheduler to assign them more or less CPU time.

All processes are related by a parent-child relationship. The only way to create a new process is by using the fork() system call which makes an identical copy of the current process. To start a second program, a process needs to call fork(), and then, one of the two copies needs to replace itself with the new program by means of the exec() call. As a consequence of this, each process has a parent process and the relationship between processes is a tree-like structure with the **init** process at its root. The init process gets started by the kernel at boot time and is responsible for system initialization and boot-up.

The kernel maintains a process table with information on all processes, including the list of open files, the state (running, waiting, stopped, sleeping, etc.), the program counter, the stack pointer, the memory allocation map, working directory, user ID, group ID, priority, and parent process ID.

While processes are generally independent of each other, there are mechanisms for them to communicate with each other. The simplest of these are to send each other signals or to send data through pipes. Signals interrupt the normal flow of a process and force it into a special signal-handling routing to react to it, before continuing normal operation. Signals can be sent between any two processes belonging to the same user. Pipes can only exist between processes sharing a common parent or having a parent/child relationship. While signals just allow to tell the other process that the signal has been sent (i.e., no additional information can be transmitted), pipes allow full communication between the two processes, sending any kind of data through them. Another important one is the possibility of sharing memory between different processes. For this, they register a common area of physical memory to be shared. To control access to shared resources, UNIX supports a control mechanism invented by Dijkstra known as semaphores.

A concept similar to a process is a thread. Every process consists of at least one thread which can be thought of as the currently executed code of the process together with its stack. A process can create additional threads which can execute in parallel, each of them having their own stack but sharing the same address space and resources. Programs using multiple threads provide concurrent execution of functions, without the large overhead of creating multiple processes, but have to be written carefully to avoid problems caused by multiple threads accessing the same memory.

INFLUENCE ON OTHER OPERATING SYSTEMS

UNIX has pioneered many concepts now commonly found in other operating systems. Its history is closely linked to the C programming language, and it has been the predominant operating system that introduced networking and on which the Internet has been built. Many of these concepts have made their way into other systems. Today, the TCP/IP networking protocol has become the *de facto* standard across platforms. The BSD UNIX socket interface to network programming has been adopted by other systems, with the Windows Winsock interface being a prominent example. Other operting systems like Windows NT are becoming more and more UNIX-like and widely conforming to the POSIX standards.

BIBLIOGRAPHY

1. A. S. Tanenbaum, *Operating Systems Design and Implementation,* Englewood Cliffs, NJ: Prentice-Hall, 1987.
2. S. C. Johnson and D. M. Ritchie, Portability of C programs and the UNIX system, *Bell Syst. Tech. J.,* **57** (6): 2021–2048, 1978.
3. D. M. Ritchie et al., The C programming language, *Bell Syst. Tech. J.,* **57** (6): 1991–2020, 1978.
4. D. M. Ritchie and K. Thompson, The UNIX time-sharing system, *Bell Syst. Tech. J.,* **57** (6): 1905–1930, 1978.
5. M. D. McIlroy, E. N. Pinson, and B. A. Tague, Forward, *Bell Syst. Tech. J.,* **57** (6): 1899–1904, 1978.
6. S. R. Bourne, The UNIX shell, *Bell Syst. Tech. J.,* **57** (6): 1971–1990, 1978.

7. B. W. Kernighan, M. E. Lesk, and J. F. Ossana, Jr., Document preparation, *Bell Syst. Tech. J.,* **57** (6): 2115–2136, 1978.
8. D. Lewine, *POSIX Programmer's Guide,* O'Reilly & Associates, 1991.
9. B. Gallmeister, *POSIX.4, Programming for the Real World,* O'Reilly & Associates, 1995.

KARSTEN BALLÜDER
Heriot-Watt University
JOHN A. SCALES
Colorado School of Mines
MARTIN L. SMITH
New England Research

UNSTABLE OPERATING POINT. See CIRCUIT STABILITY OF DC OPERATING POINTS.

URL. See UNIVERSAL RESOURCE LOCATOR.