Influential Operating Systems

Now that you understand the fundamental concepts of operating systems (CPU scheduling, memory management, processes, and so on), we are in a position to examine how these concepts have been applied in several older and highly influential operating systems. Some of them (such as the XDS-940 and the THE system) were one-of-a-kind systems; others (such as OS/360) are widely used. The order of presentation highlights the similarities and differences of the systems; it is not strictly chronological or ordered by importance. The serious student of operating systems should be familiar with all these systems.

In the bibliographical notes at the end of the chapter, we include references to further reading about these early systems. The papers, written by the designers of the systems, are important both for their technical content and for their style and flavor.

CHAPTER OBJECTIVES

- To explain how operating-system features migrate over time from large computer systems to smaller ones.
- To discuss the features of several historically important operating systems.

20.1 Feature Migration

One reason to study early architectures and operating systems is that a feature that once ran only on huge systems may eventually make its way into very small systems. Indeed, an examination of operating systems for mainframes and microcomputers shows that many features once available only on mainframes have been adopted for microcomputers. The same operating-system concepts are thus appropriate for various classes of computers: mainframes, minicomputers, microcomputers, and handhelds. To understand modern operating systems, then, you need to recognize the theme of feature migration and the long history of many operating-system features, as shown in Figure 20.1.

A good example of feature migration started with the Multiplexed Information and Computing Services (MULTICS) operating system. MULTICS was

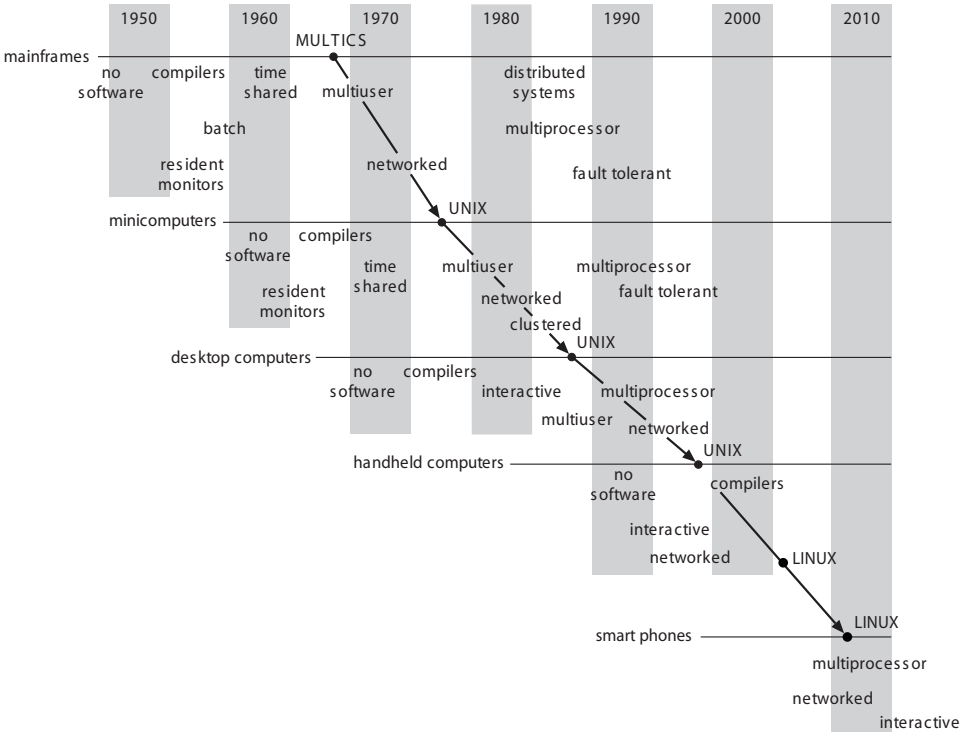

Figure 20.1 Migration of operating-system concepts and features.

developed from 1965 to 1970 at the Massachusetts Institute of Technology (MIT) as a computing **utility**. It ran on a large, complex mainframe computer (the GE 645). Many of the ideas that were developed for MULTICS were subsequently used at Bell Laboratories (one of the original partners in the development of MULTICS) in the design of UNIX. The UNIX operating system was designed around 1970 for a PDP-11 minicomputer. Around 1980, the features of UNIX became the basis for UNIX-like operating systems on microcomputers; and these features are included in several more recent operating systems for microcomputers, such as Microsoft Windows, Windows XP, and the Mac OS X operating system. Linux includes some of these same features, and they can now be found on PDAs.

20.2 Early Systems

We turn our attention now to a historical overview of early computer systems. We should note that the history of computing starts far before "computers" with looms and calculators. We begin our discussion, however, with the computers of the twentieth century.

Before the 1940s, computing devices were designed and implemented to perform specific, fixed tasks. Modifying one of those tasks required a great deal of effort and manual labor. All that changed in the 1940s when Alan Turing and John von Neumann (and colleagues), both separately and together, worked on the idea of a more general-purpose **stored program** computer. Such a machine

has both a program store and a data store, where the program store provides instructions about what to do to the data.

This fundamental computer concept quickly generated a number of general-purpose computers, but much of the history of these machines is blurred by time and the secrecy of their development during World War II. It is likely that the first working stored-program general-purpose computer was the Manchester Mark 1, which ran successfully in 1949. The first commercial computer— the Ferranti Mark 1, which went on sale in 1951—was it offspring.

Early computers were physically enormous machines run from consoles. The programmer, who was also the operator of the computer system, would write a program and then would operate the program directly from the operator's console. First, the program would be loaded manually into memory from the front panel switches (one instruction at a time), from paper tape, or from punched cards. Then the appropriate buttons would be pushed to set the starting address and to start the execution of the program. As the program ran, the programmer/operator could monitor its execution by the display lights on the console. If errors were discovered, the programmer could halt the program, examine the contents of memory and registers, and debug the program directly from the console. Output was printed or was punched onto paper tape or cards for later printing.

20.2.1 Dedicated Computer Systems

As time went on, additional software and hardware were developed. Card readers, line printers, and magnetic tape became commonplace. Assemblers, loaders, and linkers were designed to ease the programming task. Libraries of common functions were created. Common functions could then be copied into a new program without having to be written again, providing software reusability.

The routines that performed I/O were especially important. Each new I/O device had its own characteristics, requiring careful programming. A special subroutine—called a device driver—was written for each I/O device. A device driver knows how the buffers, flags, registers, control bits, and status bits for a particular device should be used. Each type of device has its own driver. A simple task, such as reading a character from a paper-tape reader, might involve complex sequences of device-specific operations. Rather than writing the necessary code every time, the device driver was simply used from the library.

Later, compilers for FORTRAN, COBOL, and other languages appeared, making the programming task much easier but the operation of the computer more complex. To prepare a FORTRAN program for execution, for example, the programmer would first need to load the FORTRAN compiler into the computer. The compiler was normally kept on magnetic tape, so the proper tape would need to be mounted on a tape drive. The program would be read through the card reader and written onto another tape. The FORTRAN compiler produced assembly-language output, which then had to be assembled. This procedure required mounting another tape with the assembler. The output of the assembler would need to be linked to supporting library routines. Finally, the binary object form of the program would be ready to execute. It could be loaded into memory and debugged from the console, as before.

A significant amount of **setup time** could be involved in the running of a job. Each job consisted of many separate steps:

1. Loading the FORTRAN compiler tape
2. Running the compiler
3. Unloading the compiler tape
4. Loading the assembler tape
5. Running the assembler
6. Unloading the assembler tape
7. Loading the object program
8. Running the object program

If an error occurred during any step, the programmer/operator might have to start over at the beginning. Each job step might involve the loading and unloading of magnetic tapes, paper tapes, and punch cards.

The job setup time was a real problem. While tapes were being mounted or the programmer was operating the console, the CPU sat idle. Remember that, in the early days, few computers were available, and they were expensive. A computer might have cost millions of dollars, not including the operational costs of power, cooling, programmers, and so on. Thus, computer time was extremely valuable, and owners wanted their computers to be used as much as possible. They needed high **utilization** to get as much as they could from their investments.

20.2.2 Shared Computer Systems

The solution was twofold. First, a professional computer operator was hired. The programmer no longer operated the machine. As soon as one job was finished, the operator could start the next. Since the operator had more experience with mounting tapes than a programmer, setup time was reduced. The programmer provided whatever cards or tapes were needed, as well as a short description of how the job was to be run. Of course, the operator could not debug an incorrect program at the console, since the operator would not understand the program. Therefore, in the case of program error, a dump of memory and registers was taken, and the programmer had to debug from the dump. Dumping the memory and registers allowed the operator to continue immediately with the next job but left the programmer with the more difficult debugging problem.

Second, jobs with similar needs were batched together and run through the computer as a group to reduce setup time. For instance, suppose the operator received one FORTRAN job, one COBOL job, and another FORTRAN job. If she ran them in that order, she would have to set up for FORTRAN (load the compiler tapes and so on), then set up for COBOL, and then set up for FORTRAN again. If she ran the two FORTRAN programs as a batch, however, she could setup only once for FORTRAN, saving operator time.

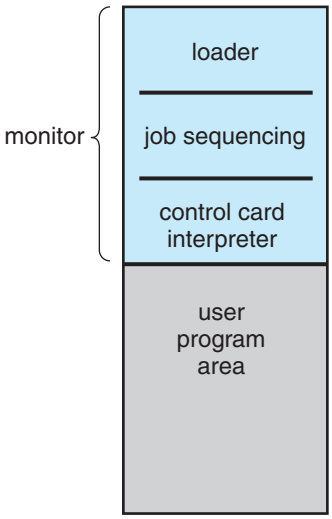

Figure 20.2 Memory layout for a resident monitor.

But there were still problems. For example, when a job stopped, the operator would have to notice that it had stopped (by observing the console), determine *why* it stopped (normal or abnormal termination), dump memory and register (if necessary), load the appropriate device with the next job, and restart the computer. During this transition from one job to the next, the CPU sat idle.

To overcome this idle time, people developed **automatic job sequencing**. With this technique, the first rudimentary operating systems were created. A small program, called a **resident monitor**, was created to transfer control automatically from one job to the next (Figure 20.2). The resident monitor is always in memory (or ***resident***).

When the computer was turned on, the resident monitor was invoked, and it would transfer control to a program. When the program terminated, it would return control to the resident monitor, which would then go on to the next program. Thus, the resident monitor would automatically sequence from one program to another and from one job to another.

But how would the resident monitor know which program to execute? Previously, the operator had been given a short description of what programs were to be run on what data. **Control cards** were introduced to provide this information directly to the monitor. The idea is simple. In addition to the program or data for a job, the programmer supplied control cards, which contained directives to the resident monitor indicating what program to run. For example, a normal user program might require one of three programs to run: the FORTRAN compiler (FTN), the assembler (ASM), or the user's program (RUN). We could use a separate control card for each of these:

$FTN—Execute the FORTRAN compiler.
$ASM—Execute the assembler.
$RUN—Execute the user program.

These cards tell the resident monitor which program to run.

We can use two additional control cards to define the boundaries of each job:

$JOB—First card of a job
$END—Final card of a job

These two cards might be useful in accounting for the machine resources used by the programmer. Parameters can be used to define the job name, account number to be charged, and so on. Other control cards can be defined for other functions, such as asking the operator to load or unload a tape.

One problem with control cards is how to distinguish them from data or program cards. The usual solution is to identify them by a special character or pattern on the card. Several systems used the dollar-sign character ($) in the first column to identify a control card. Others used a different code. IBM's Job Control Language (JCL) used slash marks (//) in the first two columns. Figure 20.3 shows a sample card-deck setup for a simple batch system.

A resident monitor thus has several identifiable parts:

- The **control-card interpreter** is responsible for reading and carrying out the instructions on the cards at the point of execution.
- The **loader** is invoked by the control-card interpreter to load system programs and application programs into memory at intervals.
- The **device drivers** are used by both the control-card interpreter and the loader for the system's I/O devices. Often, the system and application programs are linked to these same device drivers, providing continuity in their operation, as well as saving memory space and programming time.

These batch systems work fairly well. The resident monitor provides automatic job sequencing as indicated by the control cards. When a control card indicates that a program is to be run, the monitor loads the program into memory and transfers control to it. When the program completes, it

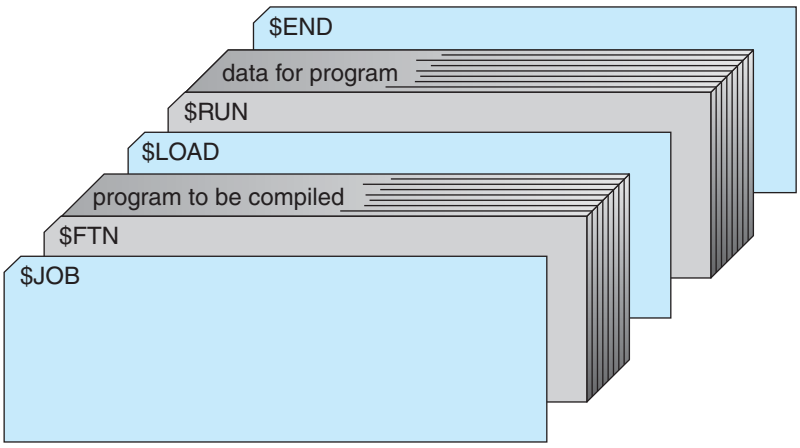

Figure 20.3 Card deck for a simple batch system.

transfers control back to the monitor, which reads the next control card, loads the appropriate program, and so on. This cycle is repeated until all control cards are interpreted for the job. Then the monitor automatically continues with the next job.

The switch to batch systems with automatic job sequencing was made to improve performance. The problem, quite simply, is that humans are considerably slower than computers. Consequently, it is desirable to replace human operation with operating-system software. Automatic job sequencing eliminates the need for human setup time and job sequencing.

Even with this arrangement, however, the CPU is often idle. The problem is the speed of the mechanical I/O devices, which are intrinsically slower than electronic devices. Even a slow CPU works in the microsecond range, with thousands of instructions executed per second. A fast card reader, in contrast, might read 1,200 cards per minute (or 20 cards per second). Thus, the difference in speed between the CPU and its I/O devices may be three orders of magnitude or more. Over time, of course, improvements in technology resulted in faster I/O devices. Unfortunately, CPU speeds increased even faster, so that the problem was not only unresolved but also exacerbated.

20.2.3 Overlapped I/O

One common solution to the I/O problem was to replace slow card readers (input devices) and line printers (output devices) with magnetic-tape units. Most computer systems in the late 1950s and early 1960s were batch systems reading from card readers and writing to line printers or card punches. The CPU did not read directly from cards, however; instead, the cards were first copied onto a magnetic tape via a separate device. When the tape was sufficiently full, it was taken down and carried over to the computer. When a card was needed for input to a program, the equivalent record was read from the tape. Similarly, output was written to the tape, and the contents of the tape were printed later. The card readers and line printers were operated ***off-line***, rather than by the main computer (Figure 20.4).

An obvious advantage of off-line operation was that the main computer was no longer constrained by the speed of the card readers and line printers but was limited only by the speed of the much faster magnetic tape units.

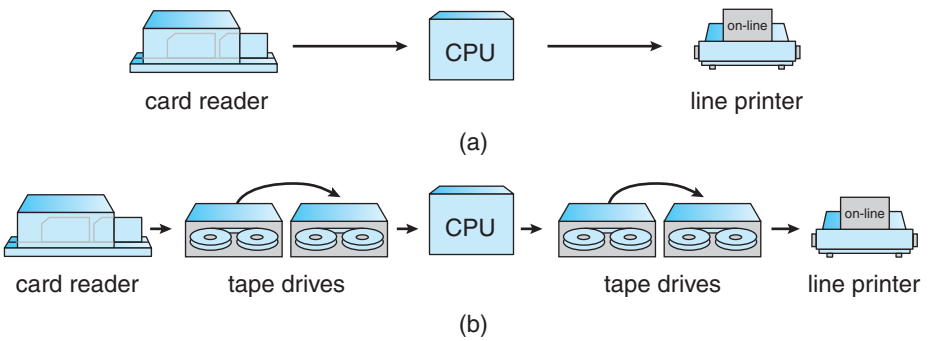

Figure 20.4 Operation of I/O devices (a) on-line and (b) off-line.

The technique of using magnetic tape for all I/O could be applied with any similar equipment (such as card readers, card punches, plotters, paper tape, and printers).

The real gain in off-line operation comes from the possibility of using multiple reader-to-tape and tape-to-printer systems for one CPU. If the CPU can process input twice as fast as the reader can read cards, then two readers working simultaneously can produce enough tape to keep the CPU busy. There is a disadvantage, too, however—a longer delay in getting a particular job run. The job must first be read onto tape. Then it must wait until enough additional jobs are read onto the tape to "fill" it. The tape must then be rewound, unloaded, hand-carried to the CPU, and mounted on a free tape drive. This process is not unreasonable for batch systems, of course. Many similar jobs can be batched onto a tape before it is taken to the computer.

Although off-line preparation of jobs continued for some time, it was quickly replaced in most systems. Disk systems became widely available and greatly improved on off-line operation. One problem with tape systems was that the card reader could not write onto one end of the tape while the CPU read from the other. The entire tape had to be written before it was rewound and read, because tapes are by nature **sequential-access devices**. Disk systems eliminated this problem by being **random-access devices**. Because the head is moved from one area of the disk to another, it can switch rapidly from the area on the disk being used by the card reader to store new cards to the position needed by the CPU to read the "next" card.

In a disk system, cards are read directly from the card reader onto the disk. The location of card images is recorded in a table kept by the operating system. When a job is executed, the operating system satisfies its requests for card-reader input by reading from the disk. Similarly, when the job requests the printer to output a line, that line is copied into a system buffer and is written to the disk. When the job is completed, the output is actually printed. This form of processing is called **spooling** (Figure 20.5); the name is an acronym for **s**imultaneous **p**eripheral **o**peration **o**n-**l**ine. Spooling, in essence, uses the disk

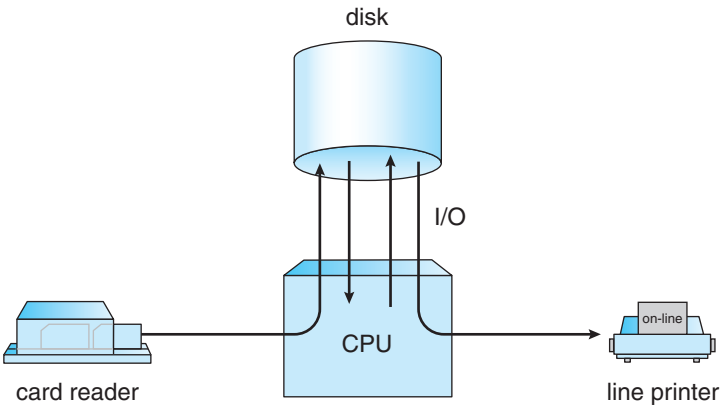

Figure 20.5 Spooling.

as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

Spooling is also used for processing data at remote sites. The CPU sends the data via communication paths to a remote printer (or accepts an entire input job from a remote card reader). The remote processing is done at its own speed, with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Spooling overlaps the I/O of one job with the computation of other jobs. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. During this time, still another job (or other jobs) may be executed, reading its "cards" from disk and "printing" its output lines onto the disk.

Spooling has a direct beneficial effect on the performance of the system. For the cost of some disk space and a few tables, the computation of one job and the I/O of other jobs can take place at the same time. Thus, spooling can keep both the CPU and the I/O devices working at much higher rates. Spooling leads naturally to multiprogramming, which is the foundation of all modern operating systems.

20.3 Atlas

The Atlas operating system was designed at the University of Manchester in England in the late 1950s and early 1960s. Many of its basic features that were novel at the time have become standard parts of modern operating systems. Device drivers were a major part of the system. In addition, system calls were added by a set of special instructions called ***extra codes***.

Atlas was a batch operating system with spooling. Spooling allowed the system to schedule jobs according to the availability of peripheral devices, such as magnetic tape units, paper tape readers, paper tape punches, line printers, card readers, and card punches.

The most remarkable feature of Atlas, however, was its memory management. **Core memory** was new and expensive at the time. Many computers, like the IBM 650, used a drum for primary memory. The Atlas system used a drum for its main memory, but it had a small amount of core memory that was used as a cache for the drum. Demand paging was used to transfer information between core memory and the drum automatically.

The Atlas system used a British computer with 48-bit words. Addresses were 24 bits but were encoded in decimal, which allowed 1 million words to be addressed. At that time, this was an extremely large address space. The physical memory for Atlas was a 98-KB-word drum and 16-KB words of core. Memory was divided into 512-word pages, providing 32 frames in physical memory. An associative memory of 32 registers implemented the mapping from a virtual address to a physical address.

If a page fault occurred, a page-replacement algorithm was invoked. One memory frame was always kept empty, so that a drum transfer could start immediately. The page-replacement algorithm attempted to predict future memory-accessing behavior based on past behavior. A reference bit for each frame was set whenever the frame was accessed. The reference bits were read

into memory every 1,024 instructions, and the last 32 values of these bits were retained. This history was used to define the time since the most recent reference (t_1) and the interval between the last two references (t_2). Pages were chosen for replacement in the following order:

1. Any page with $t_1 > t_2 + 1$ is considered to be no longer in use and is replaced.
2. If $t_1 \leq t_2$ for all pages, then replace the page with the largest value for $t_2 - t_1$.

The page-replacement algorithm assumes that programs access memory in loops. If the time between the last two references is t_2, then another reference is expected t_2 time units later. If a reference does not occur ($t_1 > t_2$), it is assumed that the page is no longer being used, and the page is replaced. If all pages are still in use, then the page that will not be needed for the longest time is replaced. The time to the next reference is expected to be $t_2 - t_1$.

20.4 XDS-940

The XDS-940 operating system was designed at the University of California at Berkeley in the early 1960's. Like the Atlas system, it used paging for memory management. Unlike the Atlas system, it was a time-shared system. The paging was used only for relocation; it was not used for demand paging. The virtual memory of any user process was made up of 16-KB words, whereas the physical memory was made up of 64-KB words. Each page was made up of 2-KB words. The page table was kept in registers. Since physical memory was larger than virtual memory, several user processes could be in memory at the same time. The number of users could be increased by page sharing when the pages contained read-only reentrant code. Processes were kept on a drum and were swapped in and out of memory as necessary.

The XDS-940 system was constructed from a modified XDS-930. The modifications were typical of the changes made to a basic computer to allow an operating system to be written properly. A user-monitor mode was added. Certain instructions, such as I/O and halt, were defined to be privileged. An attempt to execute a privileged instruction in user mode would trap to the operating system.

A system-call instruction was added to the user-mode instruction set. This instruction was used to create new resources, such as files, allowing the operating system to manage the physical resources. Files, for example, were allocated in 256-word blocks on the drum. A bit map was used to manage free drum blocks. Each file had an index block with pointers to the actual data blocks. Index blocks were chained together.

The XDS-940 system also provided system calls to allow processes to create, start, suspend, and destroy subprocesses. A programmer could construct a system of processes. Separate processes could share memory for communication and synchronization. Process creation defined a tree structure, where a process is the root and its subprocesses are nodes below it in the tree. Each of the subprocesses could, in turn, create more subprocesses.

20.5 THE

The THE operating system was designed at the Technische Hogeschool in Eindhoven in the Netherlands in the mid-1960's. It was a batch system running on a Dutch computer, the EL X8, with 32 KB of 27-bit words. The system was mainly noted for its clean design, particularly its layer structure, and its use of a set of concurrent processes employing semaphores for synchronization.

Unlike the processes in the XDS-940 system, the set of processes in the THE system was static. The operating system itself was designed as a set of cooperating processes. In addition, five user processes were created that served as the active agents to compile, execute, and print user programs. When one job was finished, the process would return to the input queue to select another job.

A priority CPU-scheduling algorithm was used. The priorities were recomputed every 2 seconds and were inversely proportional to the amount of CPU time used recently (in the last 8 to 10 seconds). This scheme gave higher priority to I/O-bound processes and to new processes.

Memory management was limited by the lack of hardware support. However, since the system was limited and user programs could be written only in Algol, a software paging scheme was used. The Algol compiler automatically generated calls to system routines, which made sure the requested information was in memory, swapping if necessary. The backing store was a 512-KB-word drum. A 512-word page was used, with an LRU page-replacement strategy.

Another major concern of the THE system was deadlock control. The banker's algorithm was used to provide deadlock avoidance.

Closely related to the THE system is the Venus system. The Venus system was also a layer-structured design, using semaphores to synchronize processes. The lower levels of the design were implemented in microcode, however, providing a much faster system. Paged-segmented memory was used for memory management. In addition, the system was designed as a time-sharing system, rather than a batch system.

20.6 RC 4000

The RC 4000 system, like the THE system, was notable primarily for its design concepts. It was designed in the late 1960's for the Danish 4000 computer by Regnecentralen, particularly by Brinch-Hansen. The objective was not to design a batch system, or a time-sharing system, or any other specific system. Rather, the goal was to create an operating-system nucleus, or kernel, on which a complete operating system could be built. Thus, the system structure was layered, and only the lower levels—comprising the kernel—were provided.

The kernel supported a collection of concurrent processes. A round-robin CPU scheduler was used. Although processes could share memory, the primary communication and synchronization mechanism was the **message system** provided by the kernel. Processes could communicate with each other by exchanging fixed-sized messages of eight words in length. All messages were stored in buffers from a common buffer pool. When a message buffer was no longer required, it was returned to the common pool.

A **message queue** was associated with each process. It contained all the messages that had been sent to that process but had not yet been received. Messages were removed from the queue in FIFO order. The system supported four primitive operations, which were executed atomically:

- `send-message` (`in` *receiver*, `in` *message*, `out` *buffer*)
- `wait-message` (`out` *sender*, `out` *message*, `out` *buffer*)
- `send-answer` (`out` *result*, `in` *message*, `in` *buffer*)
- `wait-answer` (`out` *result*, `out` *message*, `in` *buffer*)

The last two operations allowed processes to exchange several messages at a time.

These primitives required that a process service its message queue in FIFO order and that it block itself while other processes were handling its messages. To remove these restrictions, the developers provided two additional communication primitives that allowed a process to wait for the arrival of the next message or to answer and service its queue in any order:

- `wait-event` (`in` *previous-buffer*, `out` *next-buffer*, `out` *result*)
- `get-event` (`out` *buffer*)

I/O devices were also treated as processes. The device drivers were code that converted the device interrupts and registers into messages. Thus, a process would write to a terminal by sending that terminal a message. The device driver would receive the message and output the character to the terminal. An input character would interrupt the system and transfer to a device driver. The device driver would create a message from the input character and send it to a waiting process.

20.7 CTSS

The Compatible Time-Sharing System (CTSS) was designed at MIT as an experimental time-sharing system and first appeared in 1961. It was implemented on an IBM 7090 and eventually supported up to 32 interactive users. The users were provided with a set of interactive commands that allowed them to manipulate files and to compile and run programs through a terminal.

The 7090 had a 32-KB memory made up of 36-bit words. The monitor used 5 KB words, leaving 27 KB for the users. User memory images were swapped between memory and a fast drum. CPU scheduling employed a multilevel-feedback-queue algorithm. The time quantum for level i was $2 * i$ time units. If a program did not finish its CPU burst in one time quantum, it was moved down to the next level of the queue, giving it twice as much time. The program at the highest level (with the shortest quantum) was run first. The initial level of a program was determined by its size, so that the time quantum was at least as long as the swap time.

CTSS was extremely successful and was in use as late as 1972. Although it was limited, it succeeded in demonstrating that time sharing was a con-

venient and practical mode of computing. One result of CTSS was increased development of time-sharing systems. Another result was the development of MULTICS.

20.8 MULTICS

The MULTICS operating system was designed from 1965 to 1970 at MIT as a natural extension of CTSS. CTSS and other early time-sharing systems were so successful that they created an immediate desire to proceed quickly to bigger and better systems. As larger computers became available, the designers of CTSS set out to create a time-sharing utility. Computing service would be provided like electrical power. Large computer systems would be connected by telephone wires to terminals in offices and homes throughout a city. The operating system would be a time-shared system running continuously with a vast file system of shared programs and data.

MULTICS was designed by a team from MIT, GE (which later sold its computer department to Honeywell), and Bell Laboratories (which dropped out of the project in 1969). The basic GE 635 computer was modified to a new computer system called the GE 645, mainly by the addition of paged-segmentation memory hardware.

In MULTICS, a virtual address was composed of an 18-bit segment number and a 16-bit word offset. The segments were then paged in 1-KB-word pages. The second-chance page-replacement algorithm was used.

The segmented virtual address space was merged into the file system; each segment was a file. Segments were addressed by the name of the file. The file system itself was a multilevel tree structure, allowing users to create their own subdirectory structures.

Like CTSS, MULTICS used a multilevel feedback queue for CPU scheduling. Protection was accomplished through an access list associated with each file and a set of protection rings for executing processes. The system, which was written almost entirely in PL/1, comprised about 300,000 lines of code. It was extended to a multiprocessor system, allowing a CPU to be taken out of service for maintenance while the system continued running.

20.9 IBM OS/360

The longest line of operating-system development is undoubtedly that of IBM computers. The early IBM computers, such as the IBM 7090 and the IBM 7094, are prime examples of the development of common I/O subroutines, followed by development of a resident monitor, privileged instructions, memory protection, and simple batch processing. These systems were developed separately, often at independent sites. As a result, IBM was faced with many different computers, with different languages and different system software.

The IBM/360 —which first appeared in the mid 1960's — was designed to alter this situation. The IBM/360 ([Mealy et al. (1966)]) was designed as a family of computers spanning the complete range from small business machines to large scientific machines. Only one set of software would be needed for these systems, which all used the same operating system: OS/360. This arrangement

was intended to reduce maintenance problems for IBM and to allow users to move programs and applications freely from one IBM system to another.

Unfortunately, OS/360 tried to be all things to all people. As a result, it did none of its tasks especially well. The file system included a type field that defined the type of each file, and different file types were defined for fixed-length and variable-length records and for blocked and unblocked files. Contiguous allocation was used, so the user had to guess the size of each output file. The Job Control Language (JCL) added parameters for every possible option, making it incomprehensible to the average user.

The memory-management routines were hampered by the architecture. Although a base-register addressing mode was used, the program could access and modify the base register, so that absolute addresses were generated by the CPU. This arrangement prevented dynamic relocation; the program was bound to physical memory at load time. Two separate versions of the operating system were produced: OS/MFT used fixed regions and OS/MVT used variable regions.

The system was written in assembly language by thousands of programmers, resulting in millions of lines of code. The operating system itself required large amounts of memory for its code and tables. Operating-system overhead often consumed one-half of the total CPU cycles. Over the years, new versions were released to add new features and to fix errors. However, fixing one error often caused another in some remote part of the system, so that the number of known errors in the system remained fairly constant.

Virtual memory was added to OS/360 with the change to the IBM/370 architecture. The underlying hardware provided a segmented-paged virtual memory. New versions of OS used this hardware in different ways. OS/VS1 created one large virtual address space and ran OS/MFT in that virtual memory. Thus, the operating system itself was paged, as well as user programs. OS/VS2 Release 1 ran OS/MVT in virtual memory. Finally, OS/VS2 Release 2, which is now called MVS, provided each user with his own virtual memory.

MVS is still basically a batch operating system. The CTSS system was run on an IBM 7094, but the developers at MIT decided that the address space of the 360, IBM's successor to the 7094, was too small for MULTICS, so they switched vendors. IBM then decided to create its own time-sharing system, TSS/360. Like MULTICS, TSS/360 was supposed to be a large, time-shared utility. The basic 360 architecture was modified in the model 67 to provide virtual memory. Several sites purchased the 360/67 in anticipation of TSS/360.

TSS/360 was delayed, however, so other time-sharing systems were developed as temporary systems until TSS/360 was available. A time-sharing option (TSO) was added to OS/360. IBM's Cambridge Scientific Center developed CMS as a single-user system and CP/67 to provide a virtual machine to run it on.

When TSS/360 was eventually delivered, it was a failure. It was too large and too slow. As a result, no site would switch from its temporary system to TSS/360. Today, time sharing on IBM systems is largely provided either by TSO under MVS or by CMS under CP/67 (renamed VM).

Neither TSS/360 nor MULTICS achieved commercial success. What went wrong? Part of the problem was that these advanced systems were too large and too complex to be understood. Another problem was the assumption that computing power would be available from a large, remote source.

Minicomputers came along and decreased the need for large monolithic systems. They were followed by workstations and then personal computers, which put computing power closer and closer to the end users.

20.10 TOPS-20

DEC created many influential computer systems during its history. Probably the most famous operating system associated with DEC is VMS, a popular business-oriented system that is still in use today as OpenVMS, a product of Hewlett-Packard. But perhaps the most influential of DEC's operating systems was TOPS-20.

TOPS-20 started life as a research project at Bolt, Beranek, and Newman (BBN) around 1970. BBN took the business-oriented DEC PDP-10 computer running TOPS-10, added a hardware memory-paging system to implement virtual memory, and wrote a new operating system for that computer to take advantage of the new hardware features. The result was TENEX, a general-purpose timesharing system. DEC then purchased the rights to TENEX and created a new computer with a built-in hardware pager. The resulting system was the DECSYSTEM-20 and the TOPS-20 operating system.

TOPS-20 had an advanced command-line interpreter that provided help as needed to users. That, in combination with the power of the computer and its reasonable price, made the DECSYSTEM-20 the most popular time-sharing system of its time. In 1984, DEC stopped work on its line of 36-bit PDP-10 computers to concentrate on 32-bit VAX systems running VMS.

20.11 CP/M and MS/DOS

Early hobbyist computers were typically built from kits and ran a single program at a time. The systems evolved into more advanced systems as computer components improved. An early "standard" operating system for these computers of the 1970s was **CP/M**, short for Control Program/Monitor, written by Gary Kindall of Digital Research, Inc. CP/M ran primarily on the first "personal computer" CPU, the 8-bit Intel 8080. CP/M originally supported only 64 KB of memory and ran only one program at a time. Of course, it was text-based, with a command interpreter. The command interpreter resembled those in other operating systems of the time, such as the TOPS-10 from DEC.

When IBM entered the personal computer business, it decided to have Bill Gates and company write a new operating system for its 16-bit CPU of choice —the Intel 8086. This operating system, **MS-DOS**, was similar to CP/M but had a richer set of built-in commands, again mostly modeled after TOPS-10. MS-DOS became the most popular personal-computer operating system of its time, starting in 1981 and continuing development until 2000. It supported 640 KB of memory, with the ability to address "extended" and "expanded" memory to get somewhat beyond that limit. It lacked fundamental current operating-system features, however, especially protected memory.

20.12 Macintosh Operating System and Windows

With the advent of 16-bit CPUs, operating systems for personal computers could become more advanced, feature rich, and usable. The **Apple Macintosh** computer was arguably the first computer with a GUI designed for home users. It was certainly the most successful for a while, starting at its launch in 1984. It used a mouse for screen pointing and selecting and came with many utility programs that took advantage of the new user interface. Hard-disk drives were relatively expensive in 1984, so it came only with a 400-KB-capacity floppy drive by default.

The original Mac OS ran only on Apple computers and slowly was eclipsed by Microsoft Windows (starting with Version 1.0 in 1985), which was licensed to run on many different computers from a multitude of companies. As microprocessor CPUs evolved to 32-bit chips with advanced features, such as protected memory and context switching, these operating systems added features that had previously been found only on mainframes and minicomputers. Over time, personal computers became as powerful as those systems and more useful for many purposes. Minicomputers died out, replaced by general and special-purpose "servers." Although personal computers continue to increase in capacity and performance, servers tend to stay ahead of them in amount of memory, disk space, and number and speed of available CPUs. Today, servers typically run in data centers or machine rooms, while personal computers sit on or next to desks and talk to each other and servers across a network.

The desktop rivalry between Apple and Microsoft continues today, with new versions of Windows and Mac OS trying to outdo each other in features, usability, and application functionality. Other operating systems, such as AmigaOS and OS/2, have appeared over time but have not been long-term competitors to the two leading desktop operating systems. Meanwhile, Linux in its many forms continues to gain in popularity among more technical users —and even with nontechnical users on systems like the **One Laptop per Child (OLPC)** children's connected computer network (http://laptop.org/).

20.13 Mach

The Mach operating system traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system and task and thread management) were developed from scratch.

Work on Mach began in the mid 1980's and the operating system was designed with the following three critical goals in mind:

1. Emulate 4.3 BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.
2. Be a modern operating system that supports many memory models, as well as parallel and distributed computing.
3. Have a kernel that is simpler and easier to modify than 4.3 BSD.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2BSD kernel, with BSD kernel components replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for SUN 3 workstations followed shortly. Then, 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provided compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach made the kernels in these releases larger than the corresponding BSD kernels. Mach 3 moved the BSD code outside the kernel, leaving a much smaller microkernel. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows the replacement of BSD with another operating system or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the virtual machine concept, but here the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. With Release 3.0, Mach became available on a wide variety of systems, including single-processor SUN, Intel, IBM, and DEC machines and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled to the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. (Mach 2.5 was also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs of Apple Computer fame.) The initial release of OSF/1 occurred a year later, and this system competed with UNIX System V, Release 4, the operating system of choice at that time among UNIX International (UI) members. OSF members included key technological companies such as IBM, DEC, and HP. OSF has since changed its direction, and only DEC UNIX is based on the Mach kernel.

Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. This support is also exceedingly flexible, ranging from shared-memory systems to systems with no memory shared between processors. Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communication method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks. By providing low-level, or primitive, system calls from which more complex functions can be built, Mach reduces the size of the kernel

while permitting operating-system emulation at the user level, much like IBM's virtual machine systems.

Some previous editions of ***Operating System Concepts*** included an entire chapter on Mach. This chapter, as it appeared in the fourth edition, is available on the Web (http://www.os-book.com).

20.14 Other Systems

There are, of course, other operating systems, and most of them have interesting properties. The MCP operating system for the Burroughs computer family was the first to be written in a system programming language. It supported segmentation and multiple CPUs. The SCOPE operating system for the CDC 6600 was also a multi-CPU system. The coordination and synchronization of the multiple processes were surprisingly well designed.

History is littered with operating systems that suited a purpose for a time (be it a long or a short time) and then, when faded, were replaced by operating systems that had more features, supported newer hardware, were easier to use, or were better marketed. We are sure this trend will continue in the future.

Exercises

20.1 Discuss what considerations the computer operator took into account in deciding on the sequences in which programs would be run on early computer systems that were manually operated.

20.2 What optimizations were used to minimize the discrepancy between CPU and I/O speeds on early computer systems?

20.3 Consider the page-replacement algorithm used by Atlas. In what ways is it different from the clock algorithm discussed in Section 9.4.5.2?

20.4 Consider the multilevel feedback queue used by CTSS and MULTICS. Suppose a program consistently uses seven time units every time it is scheduled before it performs an I/O operation and blocks. How many time units are allocated to this program when it is scheduled for execution at different points in time?

20.5 What are the implications of supporting BSD functionality in user-mode servers within the Mach operating system?

20.6 What conclusions can be drawn about the evolution of operating systems? What causes some operating systems to gain in popularity and others to fade?

Bibliographical Notes

Looms and calculators are described in [Frah (2001)] and shown graphically in [Frauenfelder (2005)].

The Manchester Mark 1 is discussed by [Rojas and Hashagen (2000)], and its offspring, the Ferranti Mark 1, is described by [Ceruzzi (1998)].

[Kilburn et al. (1961)] and [Howarth et al. (1961)] examine the Atlas operating system.

The XDS-940 operating system is described by [Lichtenberger and Pirtle (1965)].

The THE operating system is covered by [Dijkstra (1968)] and by [McKeag and Wilson (1976)].

The Venus system is described by [Liskov (1972)].

[Brinch-Hansen (1970)] and [Brinch-Hansen (1973)] discuss the RC 4000 system.

The Compatible Time-Sharing System (CTSS) is presented by [Corbato et al. (1962)].

The MULTICS operating system is described by [Corbato and Vyssotsky (1965)] and [Organick (1972)].

[Mealy et al. (1966)] presented the IBM/360. [Lett and Konigsford (1968)] cover TSS/360.

CP/67 is described by [Meyer and Seawright (1970)] and [Parmelee et al. (1972)].

DEC VMS is discussed by [Kenah et al. (1988)], and TENEX is described by [Bobrow et al. (1972)].

A description of the Apple Macintosh appears in [Apple (1987)]. For more information on these operating systems and their history, see [Freiberger and Swaine (2000)].

The Mach operating system and its ancestor, the Accent operating system, are described by [Rashid and Robertson (1981)]. Mach's communication system is covered by [Rashid (1986)], [Tevanian et al. (1989)], and [Accetta et al. (1986)]. The Mach scheduler is described in detail by [Tevanian et al. (1987a)] and [Black (1990)]. An early version of the Mach shared-memory and memory-mapping system is presented by [Tevanian et al. (1987b)]. A good resource describing the Mach project can be found at http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html.

[McKeag and Wilson (1976)] discuss the MCP operating system for the Burroughs computer family as well as the SCOPE operating system for the CDC 6600.

Bibliography

[Accetta et al. (1986)] M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the Summer USENIX Conference* (1986), pages 93–112.

[Apple (1987)] *Apple Technical Introduction to the Macintosh Family*. Addison-Wesley (1987).

[Black (1990)] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, Volume 23, Number 5 (1990), pages 35–43.

[Bobrow et al. (1972)] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", *Communications of the ACM*, Volume 15, Number 3 (1972).

[Brinch-Hansen (1970)] P. Brinch-Hansen, "The Nucleus of a Multiprogramming System", *Communications of the ACM*, Volume 13, Number 4 (1970), pages 238–241 and 250.

[Brinch-Hansen (1973)] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).

[Ceruzzi (1998)] P. E. Ceruzzi, *A History of Modern Computing*, MIT Press (1998).

[Corbato and Vyssotsky (1965)] F. J. Corbato and V. A. Vyssotsky, "Introduction and Overview of the MULTICS System", *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 185–196.

[Corbato et al. (1962)] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, "An Experimental Time-Sharing System", *Proceedings of the AFIPS Fall Joint Computer Conference* (1962), pages 335–344.

[Dijkstra (1968)] E. W. Dijkstra, "The Structure of the THE Multiprogramming System", *Communications of the ACM*, Volume 11, Number 5 (1968), pages 341–346.

[Frah (2001)] G. Frah, *The Universal History of Computing*, John Wiley and Sons (2001).

[Frauenfelder (2005)] M. Frauenfelder, *The Computer—An Illustrated History*, Carlton Books (2005).

[Freiberger and Swaine (2000)] P. Freiberger and M. Swaine, *Fire in the Valley—The Making of the Personal Computer*, McGraw-Hill (2000).

[Howarth et al. (1961)] D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part II: User's Description", *Computer Journal*, Volume 4, Number 3 (1961), pages 226–229.

[Kenah et al. (1988)] L. J. Kenah, R. E. Goldenberg, and S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press (1988).

[Kilburn et al. (1961)] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part I: Internal Organization", *Computer Journal*, Volume 4, Number 3 (1961), pages 222–225.

[Lett and Konigsford (1968)] A. L. Lett and W. L. Konigsford, "TSS/360: A Time-Shared Operating System", *Proceedings of the AFIPS Fall Joint Computer Conference* (1968), pages 15–28.

[Lichtenberger and Pirtle (1965)] W. W. Lichtenberger and M. W. Pirtle, "A Facility for Experimentation in Man-Machine Interaction", *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 589–598.

[Liskov (1972)] B. H. Liskov, "The Design of the Venus Operating System", *Communications of the ACM*, Volume 15, Number 3 (1972), pages 144–149.

[McKeag and Wilson (1976)] R. M. McKeag and R. Wilson, *Studies in Operating Systems*, Academic Press (1976).

[Mealy et al. (1966)] G. H. Mealy, B. I. Witt, and W. A. Clark, "The Functional Structure of OS/360", *IBM Systems Journal*, Volume 5, Number 1 (1966), pages 3–11.

[Meyer and Seawright (1970)] R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-Sharing System", *IBM Systems Journal*, Volume 9, Number 3 (1970), pages 199–218.

[Organick (1972)] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).

[Parmelee et al. (1972)] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. Hatfield, "Virtual Storage and Virtual Machine Concepts", *IBM Systems Journal*, Volume 11, Number 2 (1972), pages 99–130.

[Rashid (1986)] R. F. Rashid, "From RIG to Accent to Mach: The Evolution of a Network Operating System", *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference* (1986), pages 1128–1137.

[Rashid and Robertson (1981)] R. Rashid and G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proceedings of the ACM Symposium on Operating System Principles* (1981), pages 64–75.

[Rojas and Hashagen (2000)] R. Rojas and U. Hashagen, *The First Computers—History and Architectures*, MIT Press (2000).

[Tevanian et al. (1987a)] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach Threads and the Unix Kernel: The Battle for Control", *Proceedings of the Summer USENIX Conference* (1987).

[Tevanian et al. (1987b)] A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi, "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach", Technical report, Carnegie-Mellon University (1987).

[Tevanian et al. (1989)] A. Tevanian, Jr., and B. Smith, "Mach: The Model for Future Unix", *Byte* (1989).

Credits

- Figure 1.11: From Hennesy and Patterson, *Computer Architecture: A Quantitative Approach, Third Edition*, © 2002, Morgan Kaufmann Publishers, Figure 5.3, p. 394. Reprinted with permission of the publisher.
- Figure 6.24 adapted with permission from Sun Microsystems, Inc.
- Figure 9.18: From *IBM Systems Journal*, Vol. 10, No. 3, © 1971, International Business Machines Corporation. Reprinted by permission of IBM Corporation.
- Figure 12.9: From Leffler/McKusick/Karels/Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, © 1989 by Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figure 7.6, p. 196. Reprinted with permission of the publisher.
- Figure 13.4: From *Pentium Processor User's Manual: Architecture and Programming Manual*, Volume 3, Copyright 1993. Reprinted by permission of Intel Corporation.
- Figures 17.5, 17.6, and 17.8: From Halsall, *Data Communications, Computer Networks, and Open Systems, Third Edition*, © 1992, Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figure 1.9, p. 14, Figure 1.10, p. 15, and Figure 1.11, p. 18. Reprinted with permission of the publisher.
- Figure 6.14: From Khanna/Sebree/Zolnowsky, "Realtime Scheduling in SunOS 5.0," Proceedings of Winter USENIX, January 1992, San Francisco, California. Derived with permission of the authors.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z