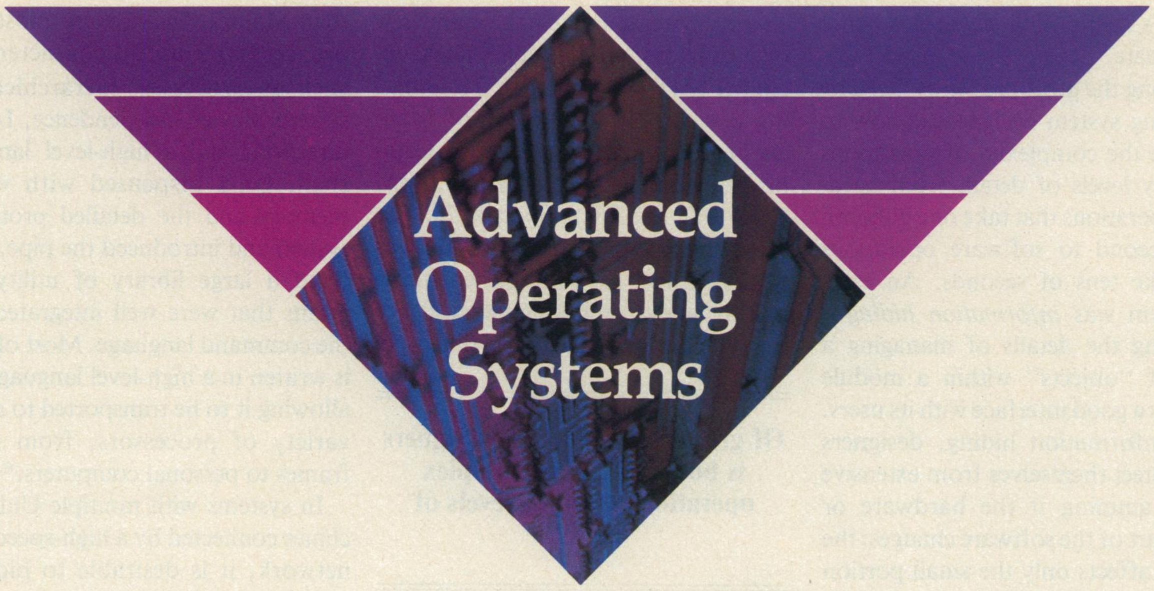


Advanced Operating Systems

Robert L. Brown and Peter J. Denning, Research Institute for Advanced Computer Science
Walter F. Tichy, Purdue University

Far from fading out of the picture, operating systems continue to meet the challenges of new hardware developments. Their success lies in abstraction levels that hide nonessential details from the user.

Operating systems of 1955 were control programs a few thousand bytes long that scheduled jobs, drove peripheral devices, and billed users. Operating systems of 1984 are much larger both in size and responsibility. The largest ones, such as Honeywell's Multics or IBM's MVS, are tens of millions of bytes long. Intermediate ones, such as Unix from AT&T Bell Laboratories or VMS from Digital Equipment Corporation, are several hundreds of thousands of bytes long. Even the smallest, most pared-down systems for personal computers are tens of thousands of bytes long.

The intellectual value of operating system research was recognized in the early 1970's. Virtually every curriculum in computer science and engineering now includes a course on operating systems, and texts are numerous. The continuing debates—over the set of concepts that should be taught and over the proper mix between concepts and implementation projects—are signs of the field's vitality.

Since 1975, personal computers for home and business have grown into a multibillion-dollar industry. Advanced graphics workstations and microcomputers have been proliferating. Local networks, such as Ethernet, ring nets, and wideband nets, and network protocols, such as X.25, PUP, and TCP/IP, allow large systems to be constructed from many small ones. The available hardware has grown rapidly in power and sophistication.

In view of these rapid hardware advances, we feel compelled to ask whether hardware will eventually obviate software control programs. Is the intellectual core recorded in operating system texts outmoded? Are operating systems an outmoded technology? On the contrary—we believe that the power and complexity of the new hardware intensifies the need for operating systems, that the intellectual core contains the concepts needed for today's computer systems, and that operating systems are and will remain essential.

What is an operating system?

Before we can explain our view on the future utility of operating systems, we need to agree on what they are. The oldest definition, which says an operating system is "a *control program* for allocating resources among competing tasks," describes only a small portion of a modern operating

0018-9162/84/1000-0173$01.00 © 1984 IEEE

system's responsibilities, and is hence inadequate.

Among the great problems faced by operating system designers is how to manage the complexity of operations at many levels of detail, from hardware operations that take one billionth of a second to software operations that take tens of seconds. An early stratagem was *information hiding*—confining the details of managing a class of "objects" within a module that has a good interface with its users. With information hiding, designers can protect themselves from extensive reprogramming if the hardware or some part of the software changes: the change affects only the small portion of the software interfacing directly with that system component. This principle has been extended from isolated subsystems to an entire operating system. The basic idea is to create a hierarchy of abstraction levels so that at any level we can ignore the details of what is going on at all lower levels. At the highest level are system users, who, ideally, are insulated from everything except what they want to accomplish. Thus, a better definition of an operating system is "a set of software extensions of primitive hardware, culminating in a virtual machine that serves as a high-level programming environment."

Operating systems of this type can support diverse environments: programming, text processing, real-time processing, office automation, database, and hobbyist.

Current systems

Most operating systems for large mainframes are direct descendants of third-generation systems, such as Honeywell Multics, IBM MVS, VM/370, and CDC Scope. These systems introduced important concepts such as timesharing, multiprogramming, virtual memory, sequential processes cooperating via semaphores, hierarchical file systems, and device-independent I/O.[1,2]

During the 1960's, many projects were established to construct timesharing systems and test the many new operating system concepts. These included MIT's Compatible Timesharing System, the University of Manchester Atlas, the University of Cambridge Multiple Access System, IBM TSS/360, and RCA Spectra/70. The most ambitious project of all was Multics (short for Multiplexed Information and Computing Service) for the General Electric 645 (later renamed Honeywell 6180) processor.[3] Multics simultaneously tested new concepts of processes, interprocess communication, segmented virtual memory, page replacement, linking new segments to a computation on demand, automatic multiprogrammed load control, access control and protection, hierarchical file system, device independence, I/O redirection, and a high-level language shell.

Of great concern to OS designers is how to manage complex operations at many levels of detail.

Another important concept of third-generation systems was the virtual machine, a simulated copy of the host. Virtual machines were first tested around 1966 on the M44/44X project at the IBM T. J. Watson Research Center. In the early 1970's virtual machines were used in IBM's CP-67 system, a timesharing system that assigned each user's process to its own virtual copy of the IBM 360/67 machine. This system, which has been moved to the IBM 370 machine, is now called VM/370.[4,5] Because each virtual machine can run a different copy of the operating system, VM/370 is effective for developing new operating systems within the current operating system. However, because virtual machines are well isolated, communication among them is expensive and awkward.

Perhaps the most influential current operating system is Unix, a complete reengineering of Multics, originally for the DEC PDP computers. Although an order of magnitude smaller than Multics, Unix retains most of its predecessor's useful characteristics, such as processes, hierarchical file system, device independence, I/O redirection, and a high-level language shell. Unix dispensed with virtual memory and the detailed protection system and introduced the pipe. It offered a large library of utility programs that were well integrated with the command language. Most of Unix is written in a high-level language (C), allowing it to be transported to a wide variety of processors, from mainframes to personal computers.[6,7]

In systems with multiple Unix machines connected by a high-speed local network, it is desirable to hide the locations of files, users, and devices from those who do not wish to deal with those details. Locus, a distributed version of Unix, satisfies this need through a directory hierarchy that spans the entire network.[8] MOS, another distributed Unix, also uses a global directory hierarchy as well as automatically migrating processes among machines to balance loads.[9]

In recent years, a large family of operating systems has been developed for personal computers, including MS-DOS, PC-DOS, Apple-DOS, CP/M, Coherent, and Xenix. All these systems have limited function, being designed for 8- and 16-bit microprocessor chips with small memories. In many respects, the growth pattern of personal computers is repeating that of mainframes in the early 1960's —for example, multiprocess operating systems for microcomputers have appeared only recently in the form of pared-down, Unix-like systems such as Coherent and Xenix. Because only large firms can sell enough machines to make their operating systems viable, there is strong pressure for standard operating systems. The emerging standards are PC-DOS, CP/M, and Unix.

Research on operating systems continues. Numerous experimental systems are exploring new concepts of system structure and distributed computation. The operating system for the Cambridge CAP machine exploits the hardware's microcode support for capability addressing to implement a large number of processes in separately protected domains. Data abstraction is easy to implement on this machine.[10]

Star OS, an operating system for the Cm* machine, supports the "task force," a group of concurrent processes cooperating in a computation. Star OS also uses capabilities to control access to objects. Another operating system for the Cm* machine, called Medusa, is composed of several "utilities," each of which implements a particular abstraction such as a file system. Each utility can include several parallel processes running on separate processors. There is no central control.[11]

Grapevine, a distributed database and message delivery system used widely within the Xerox Corporation, contains special nameservers that can find the locations of users, groups, and other services when given their symbolic names. Because Grapevine has no central control, it can survive the failures of the nameserver machines.[12] It does not provide all the services available in a high-level programming environment, however, so it is not considered a true operating system.

The V kernel is an experimental system aiming for efficient, uniform interfaces between system components. A complete copy of the kernel runs on each machine of the network and hides the locations of files, devices, and users. V is a descendant of Thoth, an earlier system worked on by the author of V.[13,14]

The Provably Secure Operating System, a level-structured system, has high-level code that has demonstrated success in the context of a rigorous, hierarchical design methodology developed at SRI International.[15] Although it was intended for secure computing, PSOS explored many principles that can help any operating system toward the goal of provable correctness.

These examples clearly illustrate that the new technology, far from diverting interest from operating systems, has created new control problems for OS designers to solve. In other words, the need for operating systems is stronger than ever.

A model operating system

Overview. In the hierarchical structure of a model operating system, functions are separated according to their characteristic time scales and their levels of abstraction. Table 1 shows an organization spanning 15 levels. It is not a model of any particular operating system but rather incorporates ideas from several systems, including facilities for distributed processing.

Each level is the manager of a set of objects, either hardware or software, the nature of which varies greatly from level to level. Each level also defines operations that can be carried out on those objects, obeying two general rules:

- *Hierarchy.* Each level adds new operations to the machine and hides selected operations at lower

Table 1. An operating system design hierarchy.

LEVEL	NAME	OBJECTS	EXAMPLE OPERATIONS
15	Shell	User programming environment scalar data, array data	Statements in shell language
14	Directories	Directories	Create, destroy, attach, detach, search, list
13	User Processes	User Process	Fork, quit, kill, suspend, resume
12	Stream I/O	Streams	Open, close, read, write
11	Devices	External devices and peripherals such as printer, display, keyboard	Create, destroy, open, close, read, write
10	File System	Files	Create, destroy, open, close, read, write
9	Communications	Pipes	Create, destroy, open, close, read, write
- - - -	- - - -	- - - -	- - - -
8	Capabilities	Capabilities	Create, validate, attenuate
7	Virtual Memory	Segments	Read, write, fetch
6	Local Secondary Store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive Processes	Primitive process, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Fault-handler programs	Invoke, mask, unmask, retry
3	Procedures	Procedure segments, Call stack, display	Mark_stack, call, return
2	Instruction Set	Evaluation stack, microprogram interpreter	Load, store, un_op, bin_op, branch, array_ref, etc.
1	Electronic Circuits	Registers, gates, buses, etc.	Clear, transfer, complement, activate, etc.

levels. The operations visible at a given level form the instruction set of an abstract machine. Hence, a program written at a given level can invoke visible operations of lower levels but no operations of higher levels.

- *Information hiding.* The details of how an object is represented or where it is stored are hidden within the level responsible for that type. Hence, no part of an object can be changed except by applying an authorized operation to it.

The principle of data abstraction embodied in the levels model traces back to Dennis and Van Horn's 1966 paper, which emphasized a simple interface between users and the kernel.[16] The first instance of a working operating system with a kernel spanning several levels was reported by Dijkstra in 1968.[17] The idea has been extended to generate families of operating systems for related machines[18] and to increase the portability of an operating system kernel.[14] The PSOS is the first complete level-structured system reported and formally proved correct in open literature.[15]

Single-machine levels: 1-8. Levels 1 through 8 in Table 1 are called single-machine levels because their operations are well understood from primitive machines and require little modification for advanced operating systems.

The lowest levels include the hardware of the system. Level 1 is the electronic circuitry, where objects are registers, gates, memory cells, and the like, and operations are clearing registers, reading memory cells, and the like. Level 2 adds the processor's instruction set, which can deal with somewhat more abstract entities such as an evaluation stack and an array of memory locations. Level 3 adds the concept of a procedure and the operations of call and return. Level 4 introduces interrupts and a mechanism for invoking special procedures when the processor receives an interrupt signal.

The first four levels correspond roughly to the basic machine as it is received from the manufacturer, although there is some interaction with the operating system. For example, interrupts are generated by hardware, but the interrupt-handler routines are part of the operating system.

Level 5 adds primitive processes, which are simply single programs in the course of execution. The information required to specify a primitive process is its stateword, a data structure that can hold the values of the registers in the processor. This level provides a context switch operation, which transfers the processor's attention from one process to another by saving the stateword of the first and loading the stateword of the second. This level contains a scheduler that selects, from a "ready list" of available processes, the next process to run after the current process is switched off the processor. This level also provides semaphores, the special variables used to cause one process to stop and wait until another process has signalled the completion of a task. Another characteristic of this level is that hardware is easy to implement.[19] Primitive processes are analogous to the *system processes* in PSOS and the *lightweight processes* in Locus.

The principle of data abstraction embodied in the levels model dates back to 1966.

Level 6 handles access to the secondary-storage devices of a particular machine. The programs at this level are responsible for operations such as positioning the head of a disk drive and transferring a block of data. Software at a higher level determines the address of the data on the disk and places a request for it in the device's queue of pending work; the requesting process then waits at a semaphore until the transfer has been completed.

Level 7 is a standard virtual memory, a scheme that gives the programmer the illusion of having a main memory space large enough to hold the program and all its data even if the available main memory is much smaller.[20] Software at this level handles the interrupts generated by the hardware when a block of data is addressed that is not in the main memory; this software locates the missing block in the secondary store, frees space for it in the main store, and requests level 6 to read in the missing block.

Level 8 implements capabilities that are unique internal addresses for software objects definable at higher levels. At this level, capabilities are read but not altered. A validate operation enables the progammer of higher level procedures to verify that actual parameters are capabilities of the expected types.

Through level 8, the operating system deals exclusively with the resources of a single machine. At the next level, however, the operating system begins to encompass a larger world including peripheral devices such as terminals and printers and other computers attached to the network. In this world, pipes, files, devices, user processes, and directories can be shared among all the machines.

Multimachine levels: 9-14. Every object in the system has two names: its *external name,* a string of characters having some meaning to users, and its *internal name,* a binary code used by the system to locate the object. The user controls mapping from external to internal names by means of directories. The operating system controls mapping from internal names to physical locations and can move objects among several machines without affecting any user's ability to use those objects. This principle, called *delayed binding,* was important in third-generation operating systems and is even more important today.[1]

To hide the locations of all sharable objects, both external and internal names must be global, that is be interpretable on any machine. Unique external names can be constructed as path names in the directory hierarchy (defined at level 14), while unique internal names are provided by capabilities (level 8). If the local network communication system (level 9) is efficient, software at the higher levels can obtain access to a remote object with little penalty.

Level 9 is explicitly concerned with communication between processes, which can be arranged through a single mechanism called a pipe. A pipe is a one-way channel: a stream of data flows in one end and out the other. A request to read items is delayed until they are actually in the pipe. A pipe can connect two processes on the same machine or on different machines equally well. A set of pipes linking levels in all the machines can serve as a broadcast facility, which is useful for finding resources anywhere in the network.[21] Pipes are implemented in Unix[6] and have been copied in recent systems such as i Max[22] and Xinu.[23]

Level 10 provides for long-term storage of named files. While level 6 deals with disk storage in terms of tracks and sectors—the physical units of the hardware—level 10 deals with more abstract entities of variable length. Indeed, a file may be scattered over many noncontiguous tracks and sectors. To be examined or updated, a file's contents must be copied between virtual memory and the secondary storage system. If a file is kept on a different machine, level 10 software can create a pipe to level 10 on the file's home machine.

Level 11 provides access to external input and output devices such as printers, plotters, and the keyboards and display screens of terminals. There is a standard interface with all these devices, and a pipe can again be used to gain access to a device attached to another machine.

Level 12 provides a means of attaching user processes interchangeably to pipes, files, or I/O devices. The idea is to make each fundamental operation of levels 9, 10, and 11 (OPEN, CLOSE, READ, and WRITE) look the same so that the author of a program need not be concerned with the differences in these objects. This strategy has two steps. First, the information contained in pipes, files, and devices is regarded simply as streams of bytes; requests for reading or writing move segments of data between streams and a user process. Second, a user process is programmed to request all input and output via *ports,* attached by the OPEN operation at runtime to specific pipes, files, or devices.

The level structure is designed to enable software verification, installation, and testing.

Level 13 implements user processes, which are virtual machines executing programs. It is important to distinguish the user process from the primitive process of level 5. All information required to define a primitive process can be expressed in the stateword that records the contents of the registers in the processor. A user process includes not only a primitive process, but also a virtual memory containing the program and its workspace, a list of arguments supplied as parameters when the process was started, a list of objects with which the process can communicate, and certain other information about the context in which the process operates. A user process is much more powerful than a primitive process.

Level 14 manages a hierarchy of directories that catalogs the hardware and software objects to which access must be controlled throughout the network: pipes, files, devices, user processes, and the directories themselves. The central concept of a directory is a table that matches external names of objects with capabilities containing their internal names. A hierarchy arises because a directory can include among its entries the names of subordinate directories. Level 14 ensures that subhierarchies encached at each machine are consistent with one another.

The directory level is responsible only for recording the associations between the external names and capabilities; other levels manage the objects themselves. Thus, when a directory of devices is searched for the string "laser," the result returned is merely a capability for the laser printer. The capability must be passed to a program at level 11, which handles the actual transmission to that printer.

Level 15 is the shell, so called because it is the level that separates the user from the rest of the operating system. The shell interprets a high-level command language through which the user gives instructions to the system. Incorporating a listener program that responds to a terminal's keyboard, it parses each line of input to identify program names and parameters, creates and invokes a user process for each program, and connects it as needed to pipes, files, and devices.

General comments on structure. The level structure is a hierarchy of functional specifications designed to impose a high degree of modularity and enable incremental software verification, installation, and testing.

In a functional hierarchy, a program at one level may directly call any visible operation of a lower level. No information flows through any intermediate level. The level structure can be completely enforced by a compiler, which can insert procedure calls or expand functions in-line.[18] A recent example of its use is Xinu, an operating system for a distributed system based on LSI 11/02 machines.[23]

It is important to distinguish the level structure discussed here from the layer structure of the International Standards Organization model of long-haul network protocols.[24] In the ISO model, information is passed down through all layers on the sending machine and back up through all layers on the receiving machine. Since each layer adds overhead to a data transmission, whether or not that overhead is required, models for long-haul network protocol structure may not be efficient in a local network.[8]

A significant advantage of functional levels over information-transferring layers is efficiency: a program that does not use a given function will experience no overhead from that function's presence in the system. For example, procedure calls will validate capabilities only when they are expected. Common objects (such as pipes, files, devices, directories, and user processes) are implemented by their own levels rather than as new "types" within a general type-extension scheme.[25]

Each level should be able to locate local objects by their internal names without having to rely on a central mapping mechanism—a strategy that enhances not only reliability in a distributed network, but also efficiency because central mechanisms are prone to be bottlenecks.

Operating system designers ought to take reasonable steps to verify that each level of the operating system meets its specifications. This goal also serves efficiency as well as reliability, since runtime checks can be omitted from system code except for conditions that cannot easily be verified a priori. System procedures must check at runtime only for the presence of expected capabilities as parameters because the calling programs may be unverified; other aspects of parameter checking can be performed by a compiler.

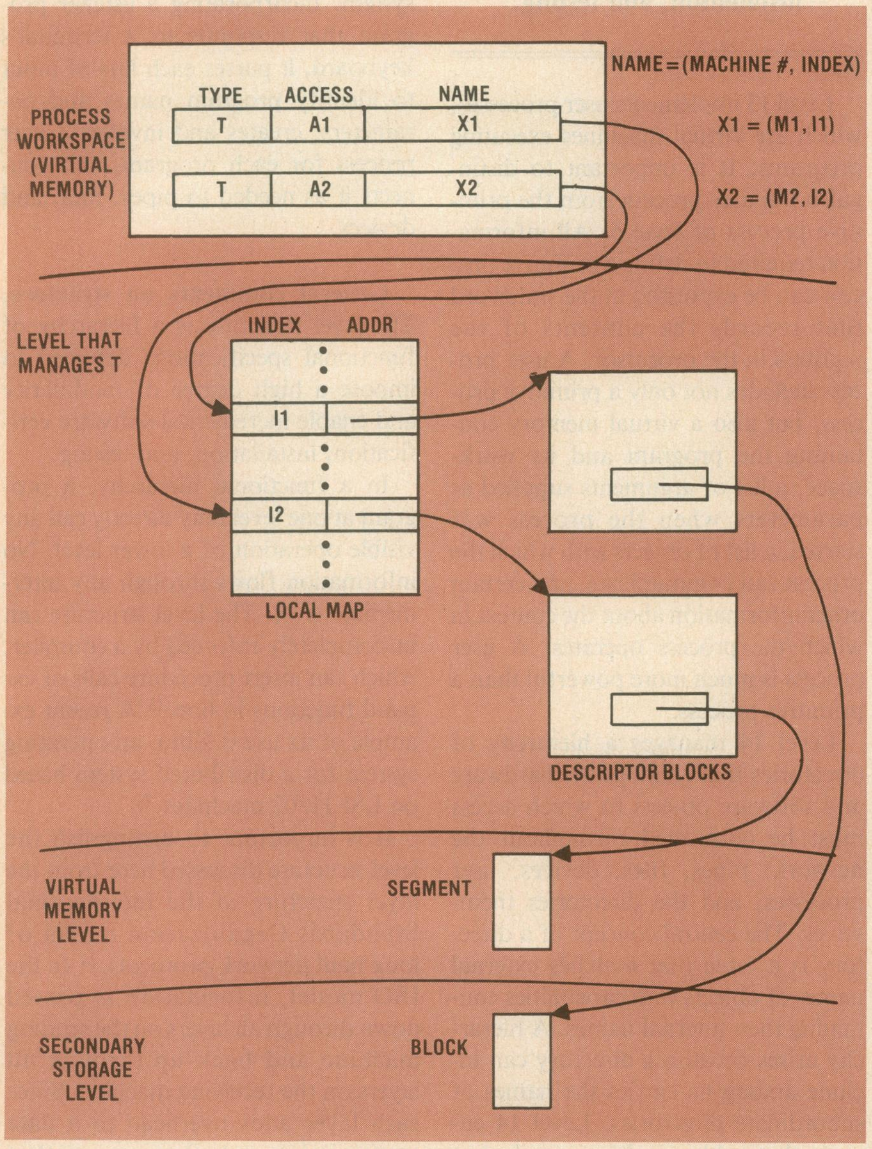


Figure 1. The storage structure for representing objects consists of a chain starting with a capability, through a local map under the control of the object's level, through a descriptor block, to the object itself. Changing the object's location requires no change in any capability. The level generates index numbers locally when it creates objects. In this example, the process holds two *T* capabilities; one object is in a segment in virtual memory and the other is in a block of secondary storage.

Distributed capabilities: level 8. The external names of sharable objects are character strings of arbitrary length that have meaning to users. Because these strings are difficult to manipulate efficiently, the operating system provides internal names for quick access to objects. One purpose of level 8 is to provide a standard way of representing and interpreting internal names for objects.

To prevent a process from applying invalid operations to an object with a known internal name, the operating system can attach a type code and an access code to an internal name. The combination of codes (type, access, internal name) is called a *capability*. All processes are prevented from altering capabilities. The system assumes that because a process holds a capability for an object it is authorized to use that object. Processes are thus responsible for controlling the capabilities they hold.

The simplest way to protect capabilities from alteration is to tag the memory words containing them with a special bit and to permit only one instruction, "create-capability," to set that bit.[10,26] The IBM System 38 is a recent example of an efficient system using tags to distinguish capabilities from other objects in memory.[27] Capabilities were first proposed as an efficient method of implementing an object-oriented operating system[16] and have continued to be used primarily for this reason.[22,25,27]

All existing implementations of capabilities are based on a central mechanism for mapping the internal name to an object. These mappings are direct extensions of virtual memory addressing schemes.[2,26] Unfortunately, a central mapping scheme cannot be used with a distributed system because the component machines may fail. Consequently, the responsibility for mapping must be distributed by allowing each procedure of levels 9 through 14 on each machine to read and interpret locally the fields of validated parameter capabilities.

The storage structure and mapping scheme for capabilities are illustrated in Figure 1. The name field of capability type *T* consists of a code *M* for the machine on which the capability was created and index number *I*. The machine number is needed because some capabilities (those for open pipes, files, and devices) can be used only on the issuing machine. The access code specifies which *T* operations can be applied to the object. Index number *I* is used by the level in charge of *T* objects on machine *M* to address a descriptor block for the given object. The descriptor block records control information about an object, time and date created and last updated, and current size and attributes of the object. The location of the descriptor block denotes the location of the object—moving the descriptor block from one machine to another effectively moves the object.

Procedures implementing operations at levels 9 through 15 must conform to certain standards that ensure the proper use of capabilities. One is an agreement on the codes for the object types, eight of which are listed in Table 2. We use the notation *T_cap* to denote a *T* capability, for example, file_cap.

The remaining standards concern the creation of capabilities pointing to new objects and the application of specific operations to those objects. Suppose level *L* ($L > 8$) is the manager of *T* objects. This level contains a procedure to create new objects of type *T* and one or more procedures to apply given functions to *T* objects. The CREATE operation must use a call of the form

T_cap: = CREATE_T(initial-value)

This procedure performs all the steps required to set up the storage for a new object of type *T*: it obtains space in secondary storage for the object and stores in it the given initial value, sets up a descriptor block, finds an unused index and sets up the entry in the local map, and finally creates a capability of type *T* (denoted "T_cap").

Creating a capability is a critical operation. Level 8 implements a special operation for this purpose:

T_cap: = CREATE_CAP(*I*)

where *I* is the local index number chosen by level *L*. When used inside the CREATE_T operation on machine *M*, CREATE_CAP constructs a capability (*T, A, M, I*), sets to 1 the capability bit of the memory word containing it, and returns the result. The code for *M* comes from a register in the processor. The code for *A* is the one denoting maximum access. The code for *T* comes from a field in the *program status word,* a processor register that also contains the program counter of the current procedure. The compilers must be set up to generate PSW = *T* only for the CREATE_T procedure and PSW = null for all other procedures. CREATE_CAP fails if executed when PSW = null.

The procedures for applying operations to a given object have the generic form

APPLY_OP(T_cap, parameters)

which means that OP (parameters) must be applied to the object denoted by T_cap. The compiler can validate that the first actual parameter on any call to APPLY_OP is indeed of type *T* by using another operation of level 8, called VALIDATE, which checks that this parameter is a capability whose type code is *T* and whose access code enables operation OP. VALIDATE can also be used to verify the presence of other capabilities among the other parameters.

A procedure may reduce the access rights of a capability it passes to another procedure by using the level 8 operation ATTENUATE. (Table 3 summarizes the operations implemented at level 8.)

Table 2: Capability type marks and their abbreviations.

LEVEL	TYPE MARK	ABBREVIATION
14	Directory	dir
13	User process	up
11	Device	dev
	Open device	op_dev
10	File	file
	Open file	op_file
9	Pipe	pipe
	Open pipe	op_pipe

Table 3. Specification of capability operations (level 8).

FORM OF CALL	EFFECT
T_cap : = CREATE_CAP (I)	If the type-mark in the current PSW is non-null, create a new capability with type field set to that mark, access code maximum, machine field the local machine identifier, and index I.
VALIDATE (p, n, (T1, a1), ... , (Tn, an))	Verify the capability at the caller's virtual address p. For at least one i=1, . . . , n the following must be true: the capability contains Ti in its type field and permits access ai. If Ti denotes op_pipe, op_file, or op_dev, the machine field must match the identifier of the local machine. (Fails if these conditions are not met.)
cap : = ATTENUATE (cap, mask)	Returns a copy of the given capability with the access field replaced by the bitwise AND of mask and the access field from cap.

Communications: level 9. The communications level provides a single mechanism, the *pipe,* for moving information from a writer process to a reader process on the same or different machines. The most important property of the pipe is that a reader must stop and wait until a writer has put enough data into the pipe to fill the request. Level 9 gives the higher levels the ability to move objects among the nodes of the network.

The external interface presented by the comunications level consists of the commands in Table 4. When two communicating processes are on the same machine, a pipe between them can be stored in shared memory and the READ_PIPE and WRITE_PIPE operations are implemented in the same way as SEND and RECEIVE operations for message queues.[28]

When the two processes are on different machines, the communications level must implement the network protocols required to move information reliably between machines (Figure 2). These protocols are much simpler than long-haul protocols because congestion and routing control are not needed, packets cannot be received out of order, fewer error types are possible, and errors are less common.[8]

The READ and WRITE operations become ambiguous unless both a reader process and a writer process are connected to a pipe. Should a writer be blocked from entering information until the reader opens its end? What happens if either the reader or writer breaks its connection? Questions like these are dealt with by a *connection protocol.* A simple connection protocol, called *rendezvous on open and close,* has the following properties:

- The open-for-reading and the open-for-writing requests may be called at different times, but both returns are simultaneous.
- The CLOSE operation, executed by the reader, shuts both ends of the pipe; when executed by the writer, the operation is deferred until the reader empties the pipe.

A pipe capability can be stored in a file or a message and passed to another machine over an existing open pipe or by broadcast. A pipe capability can also be listed in a directory making the pipe a global object. In this case, it is like a FIFO file in System-5 Unix.[29]

The communications level also contains a broadcast operation to permit levels 10, 11, and 12 to request mapping information from their counterparts on other machines. For example, if the file level on one machine cannot locally open the file named by a given capability, it can broadcast that capability to the file levels of other machines; the machine actually holding the file responds with enough information to allow the broadcaster to complete its pending OPEN operation.

Table 4. Specification of communication level interface (level 9).

FORM OF CALL	EFFECT
pipe_cap := CREATE_PIPE ()	Creates a new empty pipe and returns a capability for it. (If the caller is a user process, it can store this capability in a directory entry and make the pipe available throughout the system.)
DESTROY_PIPE (pipe_cap)	Destroys the given pipe (undoes a create-pipe operation).
op_pipe_cap := OPEN_PIPE (pipe_cap, rw)	Opens the pipe named by the pipe capability by allocating storage and setting up a descriptor block. Initially, the pipe is empty. If rw = write (rw can be read or write) the open-pipe capability has its write permission set and can be used only by the process at the input end of the pipe. If rw = read, the open-pipe capability has its read permission set and can be used only by the process at the output end of the pipe. Does not return until both reader and writer have requested connections. (Fails if the pipe is already open for writing when rw = write or for reading when rw = read.) If both sender and receiver are on the same machine, the open-pipe descriptor block will indicate that shared memory can be used for the pipe; otherwise a network protocol must be used.
READ_PIPE WRITE_PIPE CLOSE_PIPE	These have the same effects as the READ, WRITE, and CLOSE operations (described later in the section on stream I/O).
op_pipe_cap := BROADCAST (msg)	Broadcasts a message to all type managers in the network that manage objects of the same type as the calling local type manager. Returns an open pipe capability for reading responses.

Files: level 10. Level 10 implements a long-term store for files, named strings of bits of known, but arbitrary length that are potentially accessible from all machines in the network. Table 5 summarizes file operations.

To establish a connection with a file, a process must present a file capability to the OPEN_FILE operation, which will find the file in secondary storage and allocate buffers for transmissions between the file and the caller. The transmissions themselves are requested by READ_FILE and WRITE_FILE operations. Each READ operation copies a segment of information from the file to the caller's virtual memory and advances a read pointer by the length of the segment. Each WRITE operation appends a segment from the caller's virtual memory to the end of the file.

In a multimachine system, the file level must deal with the problem of nonlocal files. When a process on one machine requests to open a file stored on another machine, there are two feasible alternatives:

- Remote Open: Open a pair of pipes to level 10 on the file's home machine; READ and WRITE requests are relayed via the forward pipe for remote execution; results are passed back over the reverse pipe. An example is the Berkeley Cocanet system.[30]
- File Migration: Move the file from its current machine to the machine on which the file is being opened; thereafter all READ and WRITE operations are local. An example is the Purdue Stork file system.[31]

The open-connection descriptor block for a file, which is addressed by an open-file capability, indicates whether READ and WRITE operations can be performed locally or must interact with a surrogate process on another machine. In the latter case, the required open-pipe capability is implanted in the descriptor block by the open-file command.

Figure 3 illustrates the types of capabilities generated and used during a typical file-editing session.

One important improvement to the basic file system is to allow multiple readers and writers by building into READ and WRITE operations a solution to the "readers and writers" synchronization problem.[32] Another is to use a version control system to automatically retain different revisions of a file; the file system can then provide access to the older versions when needed.[33]

Devices: level 11. The devices level implements a common interface to a wide range of external I/O devices, including terminal displays and keyboards, printers, plotters, time-of-day

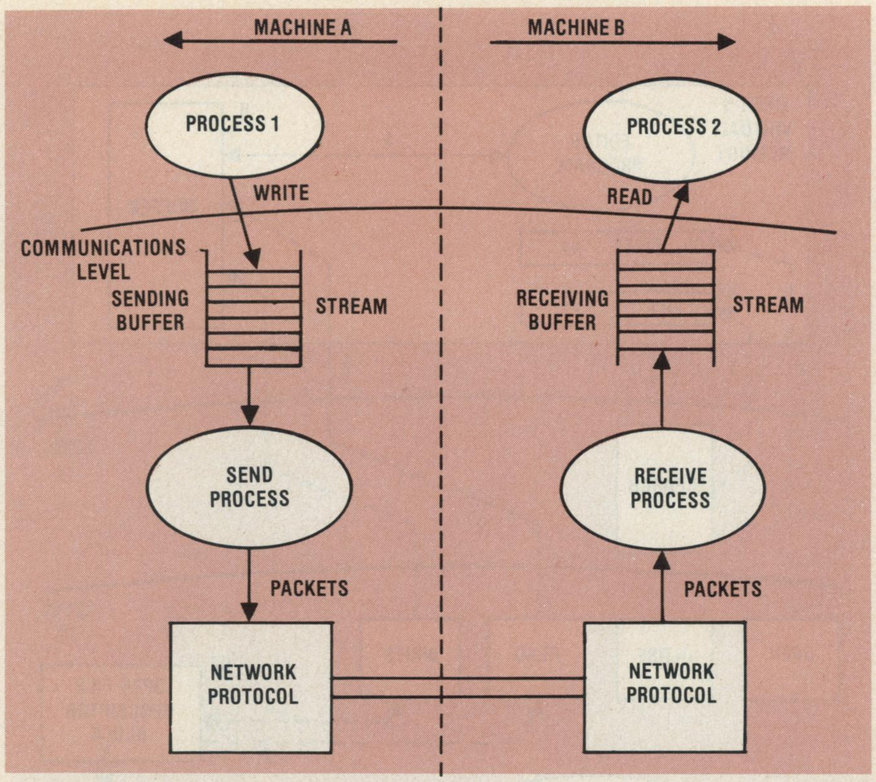


Figure 2. A network protocol must be used when two processes connected by a pipe are on different machines. The WRITE requests of the sender append segments to a stream awaiting transmission. The sender process transmits the stream as a sequence of packets, which are converted back into a stream and placed in the receiving buffer. Each READ request of the receiver waits until the requested amount of data is in the buffer, then returns it.

Table 5. Specification of the files level interface (level 10).

FORM OF CALL	EFFECT
file_cap := CREATE_FILE ()	Creates a new empty file and returns a capability for it. (If the caller is a user process, it can store this capability in a directory entry and make the file available throughout the system.)
DESTROY_FILE (file_cap)	Destroys the given file (undoes a create-file operation).
op_file_cap := OPEN_FILE (file_cap, rw)	Opens the file named by the file capability by allocating storage for buffers and setting up a descriptor block. The value of rw (read, write, or both) is put in the access field of the open-file capability. The read pointer is set to zero and the write pointer to the file's length. (Fails if the file is already open.)
READ_FILE WRITE_FILE CLOSE_FILE	These have the same effects as the READ, WRITE, and CLOSE operations (described later in the section on stream I/O).
REWIND (op_file_cap)	Resets read pointer to zero.
ERASE (op_file_cap)	Sets file length and write pointer to zero; releases secondary storage blocks occupied by the file.

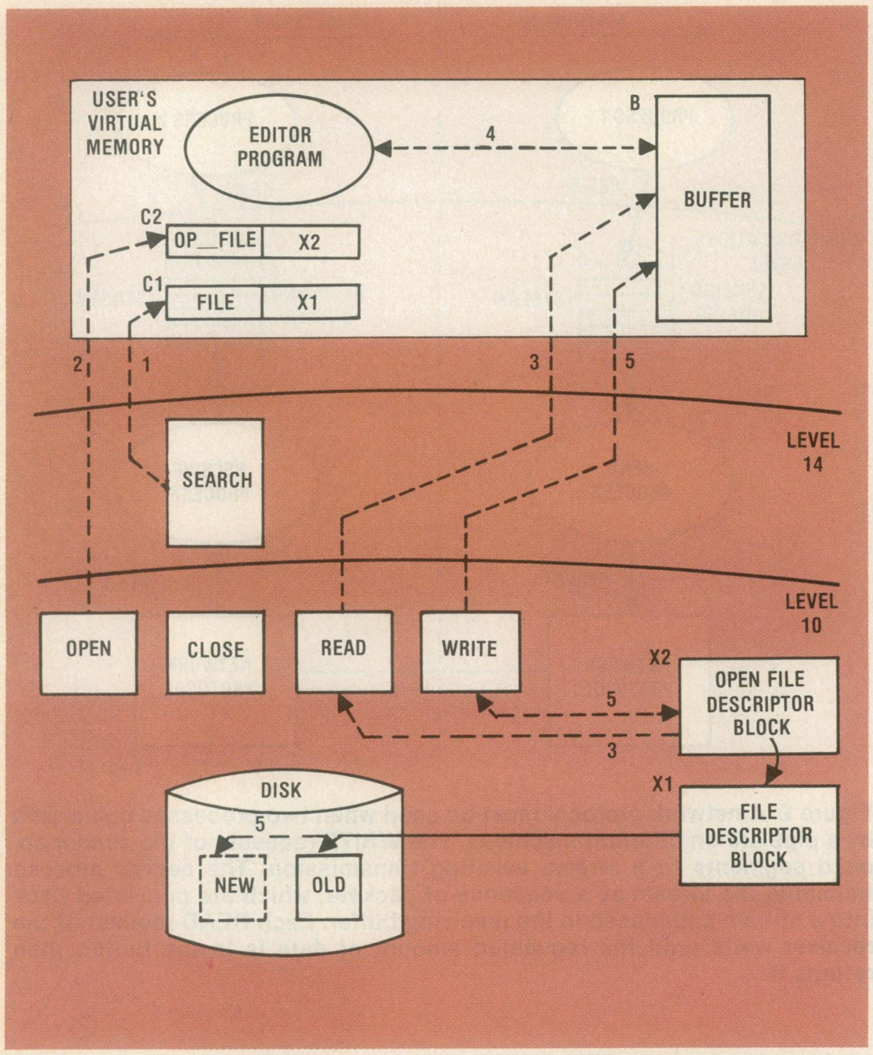


Figure 3. The steps of an editing session generate and use various capabilities. (1) Convert external name string to capability c1 by a directory-search command. (2) Open file for reading and writing by command c2 : = OPEN_FILE (c1, rw). (3) Copy file into buffer by the command READ_FILE (c2, b, all). (4) Edit contents of buffer. (5) Replace older version of file by pair of commands "ERASE (c2); WRITE_FILE (c2, b, l)," where *l* is the length of buffer. (6) Close file by command CLOSE_FILE (c2).

Table 6. Specification of devices level interface (level 11).

FORM OF CALL	EFFECT
dev_cap : = CREATE_DEV (type, address)	Returns a capability for a device of the given type at the given address. The access code of the returned capability will not include "W" if the device is read-only or "R" if the device is write-only.
DESTROY_DEV (dev_cap)	Detaches the given device from the system (undoes a create-device operation).
op_dev_cap : = OPEN_DEV (dev_cap, rw)	Opens the device named by the device capability by allocating storage for buffers and setting up a descriptor block. The value in the access field of the open-device capability is the logical AND of rw and the access code of the device capability. (Fails if the device is already open.)
READ_DEV WRITE_DEV CLOSE_DEV	These have the same effects as the READ, WRITE, and CLOSE operations (described in the section on stream I/O).

clock, and optical readers. The interface attempts to hide differences in devices by making input devices appear as sources of data streams and output devices as sinks. Obviously, the differences cannot be completely hidden—cursor-positioning commands must be embedded in the data stream sent to a graphics display, for example—but a surprising degree of uniformity is possible.

Corresponding to each device is a *device driver* program that translates commands at the interface into instructions for operating that device. A considerable amount of effort may be required to construct a reliable, robust device driver. When a new device is attached to the system, its physical address is stored in a special file accessible to device drivers.

Table 6 summarizes the interface for external devices.

Stream I/O: level 12. An important principle adopted in the hypothetical operating system described here is I/O independence. At levels 9, 10, and 11, the same fundamental operations (namely OPEN, CLOSE, READ, and WRITE) are defined for pipes, files, and devices. Although writing a block of data to a disk calls for a sequence of events quite different from that needed to supply the same data to the laser printer or to the input of another program, the author of a program does not need to be concerned with those differences. All READ and WRITE statements in a program can refer to I/O ports, which are attached to particular files, pipes, or devices only when the program is executed.

This strategy, an instance of delayed binding, can greatly increase the versatility of a program. A library program (such as the pattern-finding "grep" program in Unix) can take its input from a file or directly from a terminal and can send its output to another file, to a terminal, or to a printer. Without delayed binding, each program would have to be written to handle each possible combination of source and destination.

A common model of data must be used for pipes, files, and devices. The

Abstraction levels in practical operating systems

The level structure of the hypothetical operating system described in this article reflects the "uses-relation": All the functions that a given level of software uses must be on the same or lower levels. The resulting hierarchy of functions aids in understanding the software, verifying it, limiting the effects of modifications to it, and incrementally testing it. The first operating system explicitly incorporating levels was Dijkstra's T.H.E. operating system, built around 1968.

Levels can be a powerful descriptive tool even for operating systems with code not strictly structured by levels. The following paragraphs illustrate this with sample functions in existing operating systems of levels 8 through 15 (listed in Table 1 in the article).

Level 8: Capabilities. The idea of a hardware-recognizable, unique virtual address, or *capability,* proposed by Dennis and Van Horn in 1966, was implemented with special hardware in the Plessey System 250 in 1972, the Carnegie-Mellon Hydra in 1975 and Star OS in 1979, the Cambridge CAP system in 1980, and the Intel i Max system for the 432 microcomputer in 1981. It was implemented with microcode extensions in IBM's System 38 in 1980.

Although most other operating systems are not capability-based, they implement many ideas of capability addressing in the levels that interact with user programs. For example, a system call for opening a file usually returns a pointer to a file descriptor block; the pointer may be passed to routines for reading and writing the file. The routine for writing, for instance, checks whether the pointer passed to it is indeed for a file opened for writing. In a capability-based system, this check would be performed automatically by hardware.

Level 9: Pipes. Advances in communications research have spawned a large number of networks that provide reliable, low-cost, high-speed computer communication over a wide variety of carriers. Sample networks in use today are long-haul networks, such as DARPA's Arpanet and GTE's Telenet, and local networks such as Digital Equipment Corporation's Decnet, IBM's SNA, and Xerox's Ethernet.

Networks and communication channels are relatively recent additions to operating systems. They have usually been added as utility programs outside the kernel, resulting in a set of loosely coupled machines and the network's being visible. When the network functions are integrated into the kernel, they can be made to look like interprocess pipes, and the network becomes invisible.

Level 10: Files. All operating systems offer files for long-term storage. In addition to the simple sequential file organization presented here, most commercial systems provide record-oriented files.

Level 11: Devices. A wide variety of I/O devices is available. Most operating systems are equipped with a set of interface programs, called "drivers," for standard devices like keyboards, displays, printers, and tape drives. Most systems allow programmers of special subsystems to add drivers for the more esoteric devices such as TV cameras or robot arms.

Level 12: Stream I/O. If all standard devices and programs transmit data in the form of byte streams, considerable flexibility can be achieved in forming interconnections among programs. These ideas were first tested in Multics in 1968 and are an important aspect of the Unix operating system (1974).

Level 13: User processes. Most modern operating systems allow users to construct commands composed of several programs. The operating system implements such commands by spawning a process for each component program, even for single-user machines such as personal computers. A process is a simulated virtual machine containing an executable program.

Level 14: Directories. Directories were used as catalogs of files in the MIT Compatible Time Sharing System around 1963 with one directory per user. In 1965, CTSS was modified to permit subdirectories. Many later systems also used hierarchical directories—for example, Multics in 1968, Unix in 1972, and VMS in 1978. In Unix, directories can contain entries not only for other directories and files, but for devices and pipes. In capability-based systems, such as the Cambridge CAP, directories can contain entries for any object. In Locus, the Unix directory structure is made to span all the machines in the network, which makes files addressable in the same way on every machine and provides "location transparency," which means that a file's location cannot be determined from its name.

Many commercial and hobby systems today have the primitive, one-level directory structure type used in CTSS.

Level 15: Shell. The shell is the interactive command interpreter program that listens to the user's terminal. The term "shell" was first used in Multics, but the idea of treating commands as statements in a very high level programming language goes back to the early timesharing systems like the Manchester Atlas in 1959. The command interpreters of most batch-processing systems implement primitive "job control languages" that are often difficult to use.

simplest possibility is the stream model in which these objects are media for holding streams of bits. Corresponding to each of these objects is a pair of pointers, *r* for reading and *w* for writing; *r* counts the number of bytes read thus far and *w* counts those written thus far. Each READ request begins at position *r* and advances *r* by the number of bytes read. Similarly, each WRITE request begins at position *w* and advances *w* by the number of bytes written.

The blocks of data moved by READ or WRITE requests are called segments; *seg*(x,n) denotes a contiguous sequence of *n* bytes beginning at position *x* in a given data stream. The exact interpretation of a READ (or WRITE) request depends on whether the segment comes from a pipe, file, or device. For example, a READ request can be applied only at the output end of a pipe, and the reader must wait until the writer has supplied enough data to fill the request. An output-only device, such as a laser printer, cannot be read and an input-only device, such as a terminal keyboard, cannot be written.

The OPEN operation of level 12 returns an op_T_cap, corresponding to a given T_cap for T, a pipe, file, or device. The op_T_cap represents an active connection through which data may be passed efficiently to and from the object. The request segment in READ and WRITE operations moves across such a connection. The CLOSE operation breaks the connection.

Because the stream model has already been incorporated into the pipes, files, and devices levels, the only new mechanism involves a method of switching from a level 12 operation to its counterpart in the level for the type of object connected to a port. For example, OPEN (T_cap,rw) means

```
CASE T OF
  pipe: RETURN
  OPEN_PIPE (T_cap, rw);
  file: RETURN
  OPEN_FILE(T_cap, rw);
  dev: RETURN
  OPEN_DEV (T_cap, rw);
  ELSE: error;
END CASE
```

Table 7 summarizes the interpretation of OPEN, CLOSE, READ, and WRITE operations for the three kinds of I/O objects.

The stream model is not used in every operating system. For example, in Multics, because segments are explicit components of virtual memory, a separate concept of file is not needed because segments are retained indefinitely until deleted by their owners.[3] In Multics, the four operations of Table 7 are implicit. The first time a process refers to a segment, a "missing-binding" interrupt causes the operating system to load and bind that segment to the process. The process can thereafter read or write the segment using the ordinary virtual-addressing mechanism. Certain segments of the address space are permanently bound to devices, so reading or writing those segments is equivalent to reading or writing the device. The concept of pipe is missing, but the interprocess communication mechanism allows a data stream to be transmitted from one process to another.

User processes: level 13. A user process is a virtual machine containing a program in execution. It consists of a

Table 7. Semantics of I/O operations on objects (level 12).

FORM OF CALL	PIPE	FILE	DEVICE
op T cap := OPEN (T cap, rw)	Verify that T cap refers to an unopened pipe. Use OPEN PIPE (T cap, rw) to initialize an open-pipe descriptor block in which r=w=0; return the op pipe cap.	Verify that T_cap refers to an unopened file. Use OPEN FILE (T cap, rw) to initialize an open-file descriptor block in which r=0 and w=l (file length); return the op file_cap.	Verify that T cap refers to an unopened device. Use OPEN DEV (T cap, rw) to initialize an open-device descriptor block in which r=0 or w=0, according to whether the device is input or output; return the op dev cap.
READ (op T cap, a, n)	Wait until r+n≤w. Invoke READ PIPE (o pipe cap, a,n) to copy seg (r,n) to seg (a,n) and advance r to r +n. If n=all, return immediately with whatever is in the pipe, seg (r,w−r)	Set m=min[l−r,n]. Invoke READ FILE (op file cap, a,m)to copy seg (r,m) to seg (a,m). If n=all, return immediately with whatever is in the file, seg (r,w−r).	Invoke READ DEV (op dev cap, a, m) as for file. If n=all, return immediately with whatever input is available, seg (r,w−r). (No effect for output device.)
WRITE(op T cap, a, n)	Invoke WRITE PIPE(op pipe cap, a, n) to copy seg (a,n) to seg (w,n) and advance w to w+n. (May awaken waiting reader).	Invoke WRITE FILE(op file cap,a, n) as for pipe, plus advance l to l+n.	Invoke WRITE_DEV(op dev_cap, a, n) as for pipe. (No effect for input device.)
CLOSE(op T cap)	If pipe contains a waiting reader, return to that reader the remaining segment in the pipe. Invoke CLOSE PIPE (op pipe cap) to deallocate the open-pipe descriptor block.	Invoke CLOSE FILE (op file cap) to deallocate the open-file descriptor block.	Invoke CLOSE DEV(op dev cap) to deallocate the open-device descriptor block.

primitive process, a virtual memory, a list of arguments passed as parameters, a list of ports, and a context. Each port represents a capability for an open pipe, file, or device. Context, a set of variables characterizing the environment in which the process operates, includes the current working directory, the command directory, a link to the parent process, a linked list of spawned processes, and a signal variable that counts the number of spawned processes with an incomplete execution. Figure 4 illustrates the format of a user-process descriptor block.

A new user process is created by a FORK operation. The creator is called the "parent" and the new process a "child." A parent can exercise control over its children by resuming, suspending, or killing them. A parent can stop and wait for its children to complete their tasks by a join operation, and a child can signal its completion by an exit operation (Table 8).

The OPEN operation that appears in Table 8 hides the level 12 OPEN from higher levels and allows level 13 to store copies of all open-object capabilities in the PORTS table. When a process terminates, level 13 can assure that all open objects are closed by invoking the level 12 CLOSE operation for each entry in the PORTS table.

Directories: level 14. Level 14 is responsible for managing a hierarchy of directories containing capabilities for sharable objects. In our hypothetical system, these capabilities are pipes, files, devices, directories, and user processes; capabilities for open pipes, files, and devices are not sharable and cannot appear in directories. A hierarchy arises because a directory can contain capabilities for subordinate directories.

A directory is a table that matches an external name, stored as a string of characters, with an access code and a capability. In a tree of directories (Figure 5), the concatenated sequence of external names from the root to a given object serves as a unique, systemwide external name for that ob-

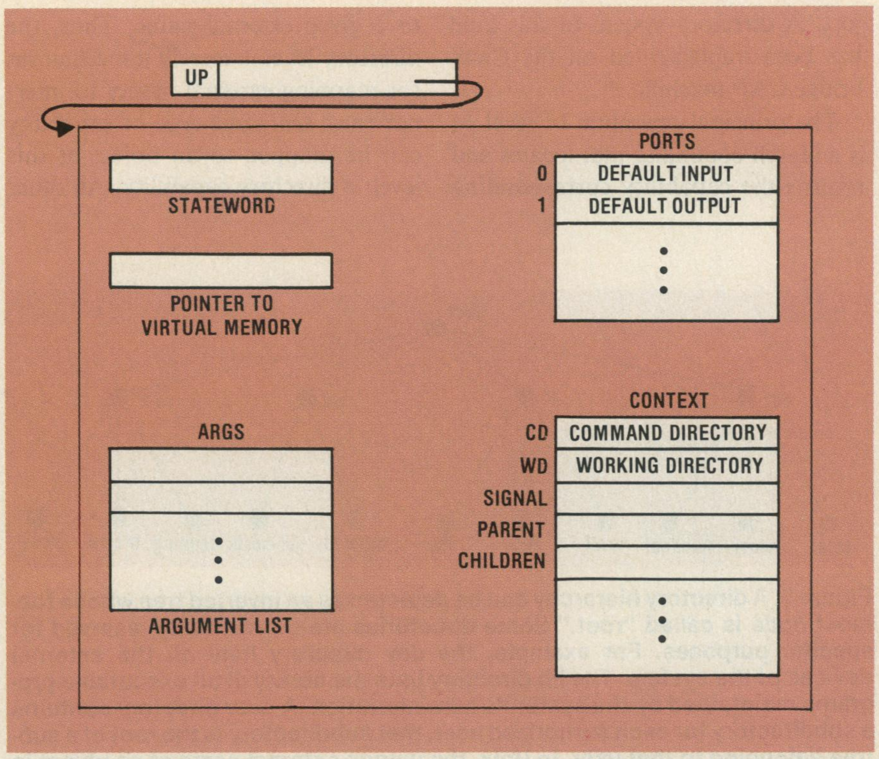


Figure 4. A user process is a virtual machine created to execute a given program. It contains a primitive process, a virtual memory holding the given program, a list of arguments supplied at the time of call, a list of ports, and a set of context variables. By convention, PORTS[0] is the default input and PORTS[1] is the default output; these two ports are bound to pipes, files, or devices when the process is created. The process can open other ports as well after it begins execution.

Table 8. Specification of user-process operations (level 13).

FORM OF CALL	EFFECT
up_cap := (FORK(file_cap, params, in, out)	Allocates a user process descriptor block. Creates a suspended primitive process and stores its index in a new user-process capability. Creates a virtual memory and loads the executable file denoted by file_cap. Copies the parameters into the ARGS list. Verifies that in and out are capabilities for pipes, files, or devices; if so, opens in for reading and puts the open-capability in PORTS [0], and opens out for writing and puts the open-capability in PORTS [1].
JOIN (A)	Waits until caller's context-variable signal is *A*, then returns.
KILL (up_cap)	Terminates the designated user process, but only if it is a child of the caller. This entails destroying the primitive process and virtual memory, closing open pipes, files, or devices connected to ports, releasing the storage held by the descriptor block, and removing the deleted process from the list of the caller's children.
EXIT	Terminates the caller process and adds 1 to the signal variable of the parent process.
SUSPEND (up_cap)	Puts the primitive process contained within the user process "up_cap" into the suspended state, but only if the caller is the parent of process "up_cap."
RESUME (up_cap)	Puts the primitive process contained within the user process "up_cap" into the ready state, but only if the caller is the parent of process "up_cap."
op_T_cap := OPEN(T_cap, rw)	Invokes the OPEN command in level 12, stores a copy of the result in the next available position in the PORTS table, and returns the result to the caller (T is pipe, file, or device).

ject. A directory system of this kind has been implemented on the Cambridge CAP machine.[10]

The principal operation of level 14 is a search command that locates and returns the capability corresponding to a given external name. Thus, the directory level is merely a mechanism for mapping external names to internal ones. Only one type of capability can be mapped to an object at this level: a directory capability. All other capabilities must be presented to their respective levels for interpretation. Information about object attributes, such as ownership or time of last use, is not kept in directories but rather in the object descriptor blocks within the object manager levels.

The requirement for systemwide unique names implies that the directory level must also ensure that portions of the directory hierarchy resident on each machine are consistent. The methods for replication in a distributed database system are a good way to guarantee this consistency.[34] To control the number of update messages in a large system, the full directory database may be kept on only a small subset of machines (for example, two or three) implementing a stable store. A workstation or other local system can store copies of the views of the directory database being accessed after the accessing user logs in. Operations that modify an entry in a directory must send updates to the stable-store machines, which relay them to affected workstations.

Specifications of the directory level's principal operations are given in Table 9. These operations allow higher level programs to create objects and store capabilities for them in directories. The table is not a complete specification of a directory manager, however; for example, it contains no command to change the name and access fields of a directory entry.

The ATTACH operation is used to create a new entry in a directory. The access field of the capability returned by a search operation will be the conjunction of the entry's access code and the access field already in the capability.

When a directory needs to be attached to another directory, the ATTACH operation must also define the parent of the newly attached directory. The operation fails if a parent is already defined (Figure 6). The DETACH operation only removes entries from directories but has no effect on the object to which a capability points. To destroy an object, the DESTROY operation of the appropriate level must be used. To minimize inadver-

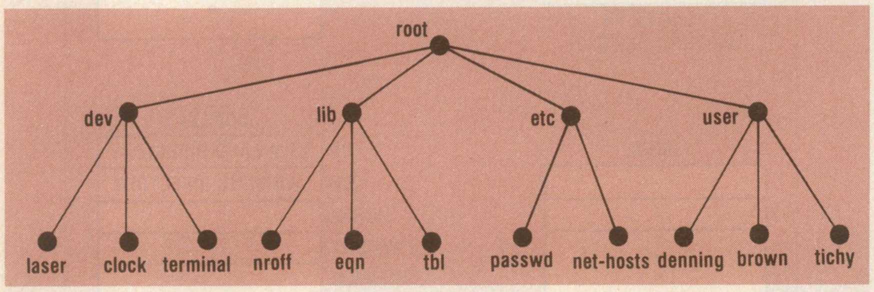


Figure 5. A directory hierarchy can be depicted as an inverted tree whose topmost node is called "root." Some directories are permanently reserved for specific purposes. For example, the dev directory lists all the external devices of the system. The lib directory lists the library of all executable programs maintained by the system's administration. A user directory contains a subdirectory for each authorized user; that subdirectory is the root of a subtree belonging to that user. In Unix, the unique external name of an object is formed by concatenating the external names along the path from the root, separated by an "/" and omitting root. Thus, the laser printer's external name is "/dev/laser."

Table 9: Specification of a directory manager interface (level 14).

FORM OF CALL	EFFECT
dir_cap := CREATE_DIR (access)	Allocates an empty directory. Returns a capability with its permission bits set to the given access code. (This directory is not attached to the directory tree.)
DESTROY_DIR (dir_cap)	Destroys (removes) the given directory. (Fails if the directory is not empty.)
ATTACH (obj_cap, dir_cap, name, access)	Makes an entry called name in the given directory (dir_cap); stores in it the given object-capability (obj_cap) and the given access code. If obj_cap denotes a directory, sets its parent entry from the self entry of the directory dir_cap. Notifies the directory stable store of the change. (Fails if the name already exists in the directory dir_cap, if the directory dir_cap is not attached, or if obj_cap denotes an already attached directory.)
DETACH (dir_cap, name)	Removes the entry of the given name from the given directory. Notifies the directory stable store of the change. (Fails if the name does not exist in the given directory or if the given directory is not empty.)
obj_cap := SEARCH (dir_cap, name)	Finds the entry of the given name in the given directory and returns a copy of the associated capability. Sets the access field in the returned capability to the minimum privilege enabled by the access fields of the directory entry and of the capability. (Fails if the name does not exist in the given directory.)
seg := LIST (dir_cap)	In a segment of the caller's virtual memory, returns a copy of the contents of the directory. (A user-level program can interrogate the other levels for other information about the objects listed in the directory, such as the date of last change.

tent deletions, the operation to destroy a directory fails if applied to a nonempty directory.

The ATTACH and DETACH operations must notify the stable store so that changes become effective throughout the system. By maintaining two conditions, this process can be made simple: (1) an empty directory must first be attached to the global directory tree before entries are made in it, and (2) a directory must be empty before being detached. A more complicated notification mechanism will be needed if a process is allowed to construct a directory subtree before its root is attached to the global directory tree.

Shell: level 15. Most system users spend a great deal of time executing existing programs, not writing new ones. When a user logs in, the operating system creates a user process containing a copy of the shell program with its default input connected to the user's keyboard and its default output connected to the user's display. The shell is the program that listens to the user's terminal and interprets the input as commands to invoke existing programs in specified combinations and with specified inputs.

The shell scans each complete line of input to pick out the names of programs to be invoked and the values of arguments to be passed to them. For each program called in this way, the shell creates a user process. The user processes are connected according to the data flow specified in the command line.

Operations of substantial complexity can be programmed in the command language of the Unix shell. For example, the operations that format and then print a file named "text" can be set in motion by the command line:

```
tbl < text | eqn | lptroff > output
```

The first program is *tbl*, which scans the data on its input stream and replaces descriptions of tables of information with the necessary formatting commands. The "<" symbol indicates that *tbl* is to take its input from the file *text*. The output of *tbl* is directed by a pipe (the " | " symbol) to the input of *eqn*, which replaces descriptions of equations with the necessary formatting commands. The output of *eqn* is then piped to *lptroff*, which generates the commands for the laser printer. Finally, > indicates that the output of *lptroff* is to be placed in a file named *output*. If "> output" is replaced with " | laser," the data is instead sent directly to the laser printer.

After the components of a command line are identified, the shell obtains capabilities for them by a series of commands:

```
c1: = SEARCH(CD, "tbl");
c2: = SEARCH(WD, "text");
c3: = CREATE_PIPE();
c4: = SEARCH(CD, "eqn");
c5: = CREATE_PIPE();
c6: = SEARCH(CD, "lptroff");
c7: = CREATE_FILE();
ATTACH(c7, WD, "output", all);
```

The variable CD holds a capability for a commands directory and WD holds a capability for the current working directory. Both CD and WD are part of the shell's context (Figure 4).

The shell then creates and resumes user processes that execute the three components of the pipeline and awaits their completion:

```
RESUME (FORK (c1, -, c2, c3));
RESUME (FORK (c4, -, c3, c5));
RESUME (FORK (c6, -, c5, c7));
JOIN (3);
```

After the JOIN returns, the shell can kill these processes and acknowledge completion of the entire command to the user through a "prompt" character.

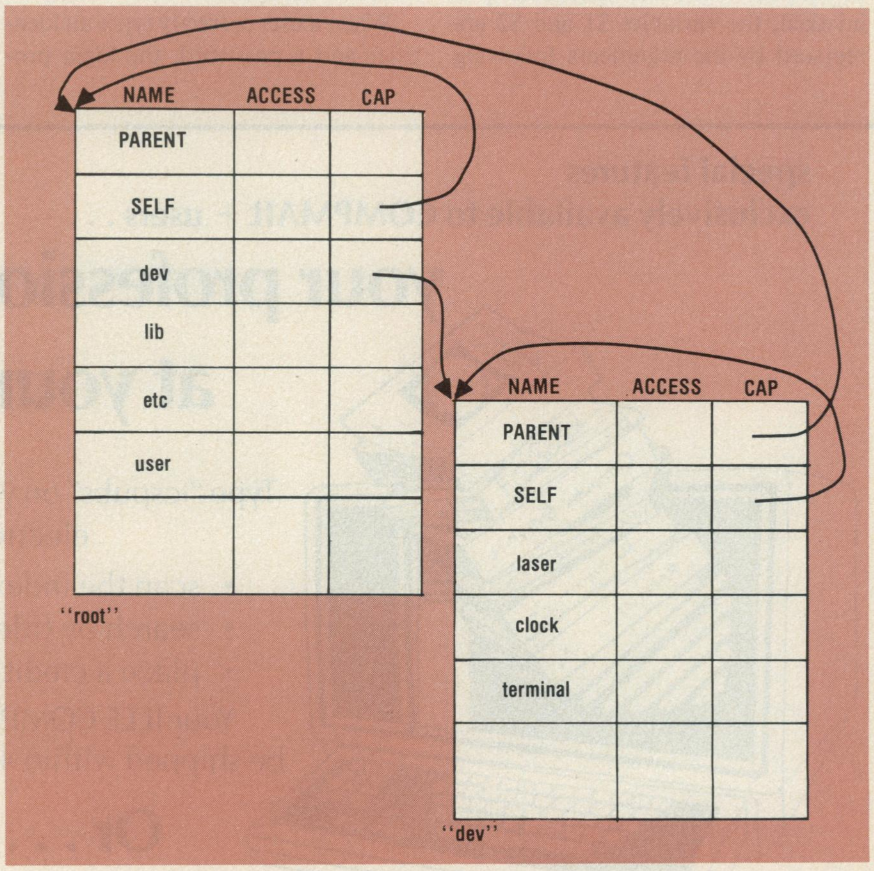


Figure 6. A directory is a table matching an external name string with an access code and a capability. Every directory contains a capability pointing to its immediate parent and a capability pointing to itself; the self-capability can be used to fill in the parent entry in a new subordinate directory. Because directories are at a higher level than files, the file system can be used to store directories. A directory containing only the self and parent entries is considered empty.

If the specification "< text" is omitted, the shell connects *tbl* to the default input, which is the same as its own, namely the terminal keyboard. In this case, the second search command is omitted and the first fork operation is

FORK (c1, -, PORTS[0], c3)

Similarly, if "> output" is omitted, the shell connects *lptroff* to the default output, the shell's PORTS[1].

If an elaborate command line is to be performed often, typing it can become tedious. Unix encourages users to store complicated commands in executable files called *shell-scripts* that become simpler commands. A file named *lp* might be created with the contents

tbl < $1 eqn lptroff >$2

where the names of input and output files have been replaced by variables $1 and $2. When the command lp is invoked, the variables $1 and $2 are replaced by the arguments following the command. For example, typing

lp text output

would substitute text for $1 and output for $2 and so would have exactly the same effect as the original command line.

System initialization

One small but essential piece of an operating system has not been discussed—the method of starting up the system. The start-up procedure, called a bootstrap sequence, begins with a very short program copied into the low end of main memory from a permanent ROM. This program loads a longer program from the disk, which then takes control and loads the operating system itself. Finally, the operating system creates a special login process connected to each terminal of the system.

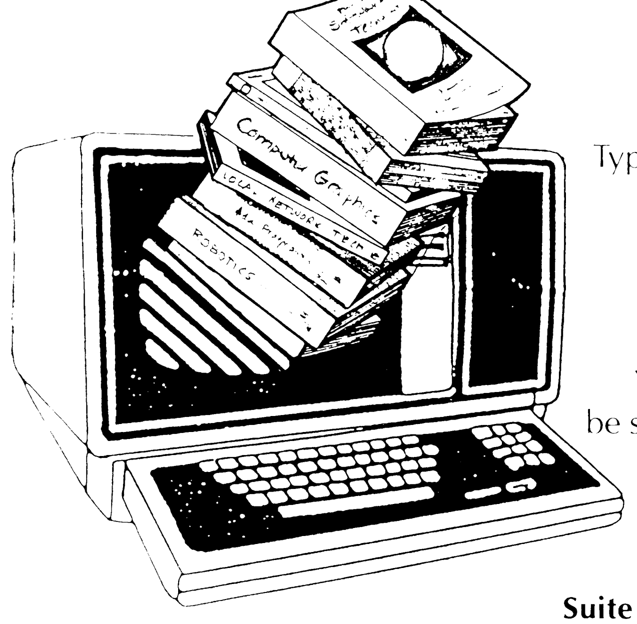
When a user correctly types an identifier and a password, the login process will create a shell process connected to the same terminal. When the user types a logout command, the shell process will exit and the login process will resume.

We have used the levels model to describe the functions of contemporary multimachine operating systems and how it is possible to systematically hide the physical locations of all sharable objects, yet be able to locate them quickly when given a name in the directory hierarchy.

The directory function can be generalized from its traditional role by storing capabilities, rather than file identifiers, in directory entries. No user machine has to have a full, local copy of the directory structure; it needs only to encache the view with which it is currently working. The full structure is maintained by a small group of machines implementing a stable store.

special features
exclusively available to COMPMAIL+ users . . .

your professional library at your fingertips.



Type "›cspubs" on your COMPMAIL+ electronic mail system and . . .

- scan the index of over 400 titles
- search by title or topic
- place a credit card order

Your IEEE COMPUTER SOCIETY books will be shipped within 48 hours of downloading!

Or . . . write for more information or a free catalog to:
IEEE Computer Society, 1109 Spring Street, Suite 300, Dept. CM884, Silver Spring, MD 20910.

IEEE COMPUTER SOCIETY

1884 · 1984 THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

The model can deal with heterogeneous systems consisting of general-purpose user machines, such as workstations, and special-purpose machines, such as stable stores, file servers, and supercomputers. Only the user machines need a full operating system; the special-purpose machines require only a simple operating system capable of managing local tasks and communicating on the network.

The levels model is based on the same principle found in nature to organize many scales of space and time. At each level of abstraction are well-defined rules of interaction for the objects visible at that level; the rules can be understood without detailed knowledge of the smaller elements making up those objects. The many parts of an operating system cannot be fully understood without keeping this principle in mind. □

Acknowledgments

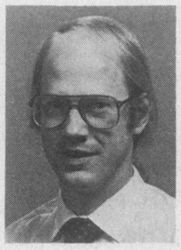
We are grateful to many colleagues for their advice and counsel on the formulation and refinement of this model: namely, J. Dennis, E. Dijkstra, N. Habermann, B. Randell, and M. Wilkes for early inspirations about hierarchical structure; K. Levitt, M. Meliar-Smith, and P. Neumann for numerous discussions about PSOS; D. Cheriton, for demonstrating these ideas in the Thoth system, D. Comer, for demonstrating these ideas in the Xinu system, and G. Popek, for demonstrating these ideas in the Locus system; D. Schrader for advice on distributed databases and help with the document; D. Denning for advice on access controls, capability systems, and verification; A. Birrell, B. Lampson, R. Needham, and M. Schroeder for advice on "server models" of distributed systems; and D. Farber, A. Hearn, T. Korb, and L. Landweber for advice on networks through the CSNET Project.

We are also grateful to the National Aeronautics and Space Administration, which supported part of this work under contract NAS2-11530 at RIACS, and to the National Science Foundation, which supported part of this work through grant MCS-8109513 at Purdue University.

References

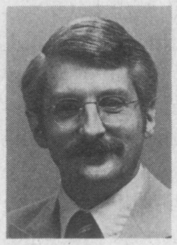
1. P. Denning, "Third Generation Computer Systems," *Computing Surveys*, Vol. 3, No. 4, Dec. 1971, pp. 175-212.
2. P. Denning, "Fault-Tolerant Operating Systems," *Computing Surveys*, Vol. 8, No. 4, Dec. 1976, pp. 359-389.
3. E. I. Organick, *The Multics System: An Examination of Its Structure*, The MIT Press, Cambridge, Mass., 1972.
4. R. P. Goldberg, "Survey of Virtual Machine Research," *Computer*, Vol. 7, No. 6, June 1974, pp. 34-46.
5. *IBM Virtual Machine Facility/370: Introduction*, tech. report GC20-1800-1, IBM, Aug. 1973.
6. D. M. Ritchie and K. L. Thompson, "The Unix Time-Sharing System," *Comm. ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.
7. B. W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, N. J., 1984.
8. G. B. Popek et al., "LOCUS: A Network Transparent, High Reliability Distributed System," *Proc. Eighth Symp. Operating Systems Principles*, Dec. 1981, pp. 169-177.
9. A. Barak and A. Litman, "MOS: A Multicomputer Distributed Operating System," *Software Practice and Experience* (to be published).
10. M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operating System*, Elsevier/North-Holland Publishing Co., New York, 1979.
11. J. K. Ousterhout et al., "Medusa: An Experiment in Distributed Operating System Structure," *Comm. ACM*, Vol. 23, No. 2, Feb. 1980, pp. 92-105.
12. A. D. Birrell et al., "Grapevine: An Exercise in Distributed Computing," *Comm. ACM*, Vol. 25, No. 4, Apr. 1982, pp. 260-274.
13. D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, Vol. 1, No. 2, Apr. 1984, pp. 19-42.
14. D. R. Cheriton, *The Thoth System: Multi-process Structuring and Portability*, Elsevier Science, New York, 1982.
15. P. G. Neumann et al., *A Provably Secure Operating System, Its Applications, and Proofs*, tech. report CSL-116, 2nd ed., SRI International, Menlo Park, Calif., May 7, 1980.
16. J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, Vol. 9, No. 3, Mar. 1966, pp. 143-155.
17. E. W. Dijkstra, "The Structure of the THE-Multiprogramming System," *Comm. ACM*, Vol. 11, No. 5, May 1968, pp. 341-346.
18. A. Habermann, A. L. F. Nico, and L. W. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *Comm. ACM*, Vol. 19, No. 5, May 1976, pp. 266-272.
19. P. J. Denning, T. D. Dennis, and J. A. Brumfield, "Low Contention Semaphores and Ready Lists," *Comm. ACM*, Vol. 24, No. 10, Oct. 1981, pp. 687-699.
20. P. J. Denning, "Virtual Memory," *Computing Surveys*, Vol. 2, No. 3, Sept. 1970, pp. 154-216.
21. D. R. Boggs, *Internet Broadcasting*, tech. report CSL-83-3, Xerox, Palo Alto Research Center, Palo Alto, Calif. Oct. 1983.
22. E. Organick, *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, 1983.
23. D. Comer, *Operating System Design: The XINU Approach*, Prentice-Hall, Englewood Cliffs, N. J., 1984.
24. A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, N. J., 1981.
25. W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill, 1981.
26. R. S. Fabry, "Capability-Based Addressing," *Comm. ACM*, Vol. 17, No. 7, July 1974, pp. 403-412.

27. H. M. Levy, *Capability-Based Computer Systems*, Digital Press, Bedford, Mass., 1984.
28. H. P. Brinch, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N. J., 1973.
29. "UNIX, System User's Manual: System V," Issue 1, Bell Laboratories, Western Electric, Jan. 1983, pp. 301-905.
30. L. A. Rowe and K. P. Birman, "A Local Network Based on the UNIX Operating System," *IEEE Trans. Software Engineering*, Vol. SE-8, No. 2, Mar. 1982, pp. 137-146.
31. J. F. Paris and W. F. Tichy, *Stork: An Experimental Migrating File System for Computer Networks*, tech. report CSD-TR-411, Purdue University, West Lafayette, Ind., Feb. 1983.
32. R. C. Holt et al., *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, Reading, Mass., 1978.
33. W. F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proc. Sixth Int'l Conf. Software Engineering*, Sept. 1983, pp. 58-67.
34. P. G. Selinger, "Replicated Data," in *Distributed Data Bases*, F. Poole, ed., Cambridge University Press, Cambridge, 1980, pp. 223-231.



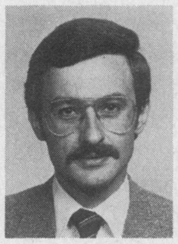
Robert L. Brown is a staff scientist at the Research Institute for Advanced Computer Science at the NASA Ames Research Center. He received a bachelor's degree in mathematics from the Ohio Wesleyan University in 1975 and is close to completing work for a PhD in computer science from Purdue University. His work emphasizes the modeling of a coherent distributed system.

Questions about this article can be directed to Peter Denning, RIACS, NASA Ames Research Center, MS 230-5, Moffet Field, CA 94035.



Peter J. Denning is the director of the Research Institute for Advanced Computer Science (RIACS) at the NASA Ames Research Center in Mountain View, California. Previously, Denning was head of the Computer Sciences Department at Purdue University and assistant professor of electrical engineering at Princeton University. He received a PhD and an SM from MIT's Electrical Engineering Department in 1968 and 1965, respectively, and a BEE from Manhattan College in 1964. Denning's primary research interests are computer systems architecture, operating systems, and performance modeling. He has published over 120 papers and articles in these areas since 1967.

Denning is a past president of the ACM and has held many editorial positions including editor-in-chief of *Communications of the ACM* and editor-in-chief of ACM's *Computing Surveys*. He is a member of the New York Academy of Sciences, Sigma Xi, and IFIP Working Group 7.3 on Computer Performance Modeling.



Walter F. Tichy is assistant professor in the Department of Computer Science at Purdue University. His research interests include operating systems and programming environments, and he is currently investigating distributed, location-transparent, and migrating file systems for heterogeneous computer networks. In programming environments, he is studying version control tools, delta methods, and module interconnection languages.

Tichy received his BS in mathematics and computer science from the Technical University in Munich, Germany, in 1974, and his MS and PhD in computer science from Carnegie-Mellon University in Pittsburgh, Pennsylvania, in 1976 and 1980, respectively. Tichy is a member of the IEEE, ACM, and Sigma Xi.

AI
EMPLOYMENT OPPORTUNITIES

Advanced Information & Decision Systems (AI&DS) invites qualified candidates to apply for research and programming positions in artificial intelligence R&D. AI&DS is a growing employee-owned company of 63 highly talented people working to develop and implement state-of-the-art solutions to real-world problems in areas such as artificial intelligence, software engineering, image understanding, information management, estimation and control, decision theory, signal processing, and cognitive science. Throughout 1984, positions will be available on projects involving:

- **Knowledge-Based and Expert Systems**
  - General Architectures
  - Knowledge Acquisition
- **Hypothesis Formation**
  - Image and Signal Understanding
  - Image Processing
  - Information Integration
- **Planning and Control**
- **Decision-Making Aids**
- **Robotics**
- **Distributed AI**
- **Intelligent User Interfaces**
- **Natural Language Understanding**
- **Software Development Aids**
- **Information Management**
  - Information Retrieval
  - Database Management
- **Systems Support**
  - UNIX Systems Programming
  - LISP Programming

Located in the beautiful San Francisco Bay Area, AI&DS offers a pleasant and supportive work environment, with expanding hardware facilities that include VAXs, Symbolics 3600s, SUNs, and a Vicom image processor linked by Ethernets and connected to the ARPANET.

To find out more about AI&DS and the positions listed above, please phone Mary Margaret Morton or Clif McCormick at (415) 941-3912, or send your resume to:

Advanced Information & Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, California 94040-1270

AI&DS

An equal opportunity employer
U.S. citizenship required for many projects