Operating Systems

Peter J. Denning, James J. Hunt, and Walter F. Tichy

August 17, 1999

1 Introduction

Early operating systems were control programs a few thousand bytes long that scheduled jobs, drove peripheral devices, and kept track of system usage for billing purposes. Modern operating systems are much larger, ranging from hundreds of thousands of bytes for personal computers (e.g., MS/DOS, Xenix) to tens of millions of bytes for mainframes (e.g., Honeywell's Multics, IBM's MVS, AT&T's UNIX) and hundreds of millions of bytes for some servers (Microsoft's NT). In addition to managing processors, memory, and dozens of input/output devices, modern operating systems also provide numerous services such as Internet communications, Web communications, inter-process communications, file and directory systems, data transfer over local networks, and command languages and graphical user interfaces for invoking and controlling programs. These high-level services hide the primitive facilities of the base computer, such as interrupts, status registers, and device interfaces, from the user. The operating system builds its high-level services by wrapping the low-level hardware facilities in layers of software, resulting in a powerful virtual machine that is much easier to use than the basic hardware. Thus an operating system provides two classes of functions: orderly allocation of computing resources among processes contending for them, and an extended machine that provides a powerful programming environment. These two classes are not independent; poor structure can make resource allocation a nightmare. The microkernel architecture to be discussed below is a good structure that enables efficient resource allocation and powerful programming environments.

2 Historical Development of Operating Systems

Most operating systems for mainframes and servers are descendants of third-generation systems, such as Honeywell Multics, IBM VMS, 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[8, 9]. These

concepts all helped improve system throughput and utilization and insulate programming from the details of a particular machine.

During the 1960s, many projects were established to construct timesharing systems and test 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 processor (later renamed the Honeywell 6180) [20]. Multics embraced every important system concept of the day: processes, interprocess communication, segmented virtual memory, page replacement, linking new libraries to a computation on demand, automatic multiprogrammed load control, access control, protection rings, security kernel, hierarchical file system, device independence, I/O redirection, and a high-level language shell.

Perhaps the most influential current operating system is Unix. Originally developed at AT&T Bell Laboratories for DEC PDP computers, Unix distilled the most useful features of Multics into a kernel that fit into the small memory of a minicomputer. Unix retained its predecessor's processes, hierarchical file system, device independence, I/O redirection, and a high-level language shell. Though the first version of Unix did not have virtual memory, most later versions did. It introduced an innovation, the pipe, which enables programs to be strung together by directing (piping) the output of one program into the input of the next. Arbitrarily long pipelines can be constructed from simpler programs to solve complicated problems. Unix offered a large library of utility programs that were well integrated with the command language. The most important innovation introduced with UNIX is the use of a high level programming language (in this case C) for the vast majority of kernel programming. This allowed UNIX to be transported to a wide variety of processors from mainframes to personal computers[24, 17].

In the 1980s, a new genre of operating systems were developed for personal computers, including MS-DOS, PC-DOS, Apple-DOS, CP/M, Coherent, and Xenix. All these systems were of limited function, being initially designed for 8- and 16-bit microprocessor chips with small memories. In many respects, the growth path of personal computer operating systems in the 1980s recapitulates that of mainframes in the early 1960s. For example, multiprogrammed operating systems for microcomputers appeared late in the 1980s in the forms of multiple background tasks. Simultaneously active application programs Multiprocessor operating systems soon followed, e.g., Windows NT and OS/2. This repetition parallels that of the hardware development. Mainframes were initially CPU limited, then memory limited, and finally IO limited. Micro computers went through these same development stages. Processor speeds and memories of personal computers are now sufficient to support full-fledged operating systems.

With multi-processors and computer networks in the early 1980s, operating systems began to manage the resources of multiple computers at once. An

early example is StarOS, an operating system for the CM* (pronounced "CM star") machine, a multicomputer consisting of several dozen individual computers linked by a special network; StarOS supported the "task force," a group of processes cooperating in a distributed computation[16]. Medusa, another operating system for CM*, was composed of several "utilities," each of which implemented a particular abstraction such as a file system; there was no central control[19].

Xerox's Grapevine, a distributed database and message delivery system, contained special name servers that located users, groups, and other services when given their symbolic names. Because Grapevine had no central control, it could survive failures of the name server machines[2].

Established, single-machine operating systems such as Unix and DEC's VMS evolved to accommodate networks of computers. Such operating systems typically support standards for accessing files on remote servers from any machine in a network. Locus[22] and the Apollo DomainOS are early examples of operating systems providing a directory hierarchy that spans an entire network. Sun's Network File System (NFS) was one of the first open, and hence widely available, Unix-based network file systems[27]. Carnegie-Mellon's Andrew system provides a Unix-based network file system that spans thousands of computers around the campus; it allows users to access files without having to know their locations and it improves performance by caching whole files at individual nodes in the network [15]. The Mach operating system, also developed at Carnegie-Mellon University, handles a variety of distributed system operations including a uniform file name space, a virtual shared computational memory, and multiprocessing; it is compatible with Unix [1, 23]. Many of these systems rely on a remote procedure call facility within the operating system to support operations distributed among many machines.

The merging of mainframe functions into personal-computer operating systems was complete by the end of the 1990s. The Mach kernel was incorporated in the NeXTStep operating system, which was acquired by Apple Computer in 1997 and will be offered in Apple's MacOS X. Linux, NetBDS, and FreeBSD, free software version of Unix, were widely available for personal computers and network servers.

Many organizations have found that distributed systems can have unacceptably high costs for system administration – e.g., the separate installation of a new version of software into hundreds of workstations. For this reason, large networks of computers require simplified, central management. Network management functions can be found in server operating systems such as Sun Microsystem's Solaris, Microsoft's NT, and the free Linux operating system.

Personal computers, workstations, and networks do not constitute the entire universe of machines needing operating systems. Demands for high-performance computing have led to massively parallel computers containing thousands and more processors. Operating systems for these machines must provide a single system view, meaning that thousands of processors or computing nodes in a

cluster can be operated and programmed as a single resource. Operating systems must support extremely fast synchronization and communication. Each processor may have its own devices attached, and hence the operating system must control thousands of I/O channels at once. Virtual memory and time-sharing must be extended to accommodate massive parallelism. In parallel computers with shared memory, programs run in a uniform address space without seeing processor/memory boundaries. In distributed memory computers, a programmer must distinguish local memory which is accessed through normal instructions, and remote memory, which is accessed by sending messages—but some operating systems are seeking to hide even this within a virtual memory. Processes need not run on a fixed processor, but may migrate through a network. Perhaps the most important challenge is that the programming environment should permit parallel programs to be written with only modest effort beyond that required for sequential ones[10]. Current operating systems research addresses these problems.

3 A Model Operating System

3.1 Overview

Over the years operating system designers have tended to use just two strategies for organizing the software: the monolith and the kernel architectures. The monolith results from a design strategy based on defining modules for operating system functions; any module can call any other provided it follows the interface specifications. All the modules must be linked together to create the operating system executable file and that file must be completely loaded into the computer's memory. Every module operates in supervisor mode so that it can have access to the hardware resources of the computer; application programs run in user mode, which has no such access. When an application program requires access to such a resource, say a scanner, it must make a supervisor call to invoke a module to perform that access on its behalf. The supervisor call instruction switches the computer from user mode to supervisor mode and forces entry to one of the operating system modules.

Monolithic operating systems can become extremely large and unwieldy. The monolith accumulates all the software that might ever be needed on all platforms, while the computer on which it is actually running needs only a fraction of it. The operating system becomes difficult to adapt to different hardware configurations, difficult to extend and contract. Microsoft Windows 98, Windows NT, and Apple MacOS, all illustrate this trend. Both require 16-32 MB of RAM, and their disk files run upwards of 100 MB. Both systems are well known for numerous bugs and frequent crashes, problems that seem only to grow worse with each new (and larger) release.

The kernel architecture avoids these problems by careful design of the mod-

ules. The kernel is designed as a small set of modules that must run in supervisor mode; every other operating system function (and application) is designed as an extension that can be invoked as needed, does not have to be memory resident, and does not have to operate in supervisor mode. A typical kernel implements interrupts, low-level I/O, processes, semaphores, virtual memory, and interprocess communication. Everything else is treated as an extension – files, directories, network services, user interfaces. Since the number of sensitive kernel functions is small, these systems tend to be much more bug free and stable than their monolith counterparts. An error occurring in one of the extended functions may crash that function, but it is very unusual to crash the kernel. The term *microkernel* is now frequently used to call attention to compact kernels.

The principle of levels can be used to structure either a monolithic or kernelized system. Software is built up as a hierarchy of levels, each one constructed on the ones below it. The programs at a given level depend only on functions provided by lower levels. The bare hardware is the lowest layer, the application programs the highest. The microkernel, layered directly over the hardware, consists of several levels itself. The levels structure gives a systematic way of building the software, testing it, and proving it correct.

Each level adds new functions or operations and hides selected functions at lower levels. The functions visible at a given level form the instruction set of an abstract or virtual machine. Hence, a program written at a given level can invoke visible operations at lower levels, but not operations on higher levels.

The first operating system constructed as a hierarchy of levels was Dijkstra's THE of 1968 [11]. The idea has been extended to generate families of operating systems for related machines [13] and to increase the portability of an operating system kernel [5]. The Provably Secure Operating System (PSOS) is the first complete level-structured system reported and formally proved correct in the open literature [18]. XINU demonstrates that the level-structure principle can lead to exceptionally compact microkernels that can fit into the small memory of a microchip [6]. A distributed operating system design with ten levels appears in [4]. Mach, Chorus, and Amoeba are three microkernel architectures that support multiprocessors; for a comparison see [30].

3.2 The Microkernel

Table 1 shows the minimal set of abstractions in a microkernel. The microkernel contains no file or directory system, offers no remote procedure call, and may even require some of the memory management to be performed in user mode. The functions in the microkernel are included there because they are difficult or inefficient to provide elsewhere.

Level 1 dispatches interrupts and manages access to peripheral devices. The interrupt dispatcher receives signals from condition detectors and responds by immediately transferring control of the CPU to a corresponding condition-

LEVEL	NAME	OBJECTS	EXAMPLE OPERATIONS
5	Processes	Process	Fork, suspend, resume, join, signal, exit, kill
4	Inter-process communication	Message, port	Send, receive, transmit
3	Memory management	Address, segment	Create, destroy, map
2	Threads	Thread, ready list, semaphore,	Fork, suspend, resume, wait, signal, kill
1	Low-level IO	Device, device driver	read, write

Table 1: Microkernel layers

handler routine. Condition handlers are defined at each level that must respond to conditions associated with that level; for example, the memory manager responds to missing-page faults and the IPC level to interrupts signaling the arrival of packets from the network. The entry points of condition handler procedures are stored in the interrupt vector, a list used by the interrupt dispatcher to locate them. Device driver programs manage operations such as positioning the head of a disk drive and transferring blocks of data. Software at a higher level determines the address of the data on the disk and places requests 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 2 implements threads. A thread is a single flow of control in some address space. It is an abstraction for the instruction trace of a CPU executing program code. Each thread operates in a context that includes a program stack, the CPU register contents,interrupt masks, and any other flags or state information needed by the CPU to continue running the thread. This level provides a context switch operation, which transfers a processor's attention from one thread to another by saving the context of the first and loading the context of the second. A scheduler in this level selects the next thread to run from a "ready list" of available threads. The scheduler usually accommodates machines with more than one processor available. This level provides the clock interrupt handler, which forces the scheduler to switch threads at regular intervals. This level also provides semaphores, the special variables used to cause one thread to stop and wait until another thread has signaled the completion of a task. Threads are analogous to "system processes" in PSOS and "lightweight processes" in Locus.

Level 3 manages the computer's main memory, or RAM (random access memory). The memory manager does two jobs: (1) it enforces separation of address spaces, and (2) it moves blocks of data between RAM and secondary storage media. An address space is the set of addresses that a CPU can gener-

ate. The simplest form of address-space separation is a partition of the RAM into disjoint regions, each delineated by a base-and-bound register; an address generated by the CPU is taken to be relative to the base register and must not exceed the bound. A group of threads can be created in the same address space; they cannot be protected from one another, but they can communicate very easily through common variables. Virtual memory distinguishes between address space (CPU addresses) and memory space (RAM addresses). With each address space it associates a mapping table that converts a CPU address into a RAM address. Virtual memory permits a CPU to access only the memory locations visible through the mapping table, thereby enforcing separation (partition) among address spaces. The mapping table stores access flags so that the CPU can be further restricted by denying read or write access to individual objects. Virtual memory can be configured to automatically move blocks of data between RAM and disk, thereby automating swapping of data between memory levels and relieving the programmer of any necessity to compile different versions of the program for different RAM sizes [7]. Automation of swapping is achieved by adding a "presence bit" to the mapping table. When the CPU generates the address of an object marked as not present, the mapping hardware generates an address-snag interrupt that invokes the missing-object interrupt routine in Level 3. That routine locates the missing block in the secondary store, frees space for it in the RAM, and moves the missing block in.

Whether to use virtual memory or simple memory partition is a design tradeoff between system complexity and protection. For applications that run a small number of mutually-trusting threads and where the entire application can fit into a memory partition, virtual memory would be unnecessary. These include digital organizers and embedded applications. For applications whose memory demands are unpredictable or for which a high degree of protection is required, virtual memory would be the best approach. To facilitate the tradeoff, some systems do not include all the virtual memory functions in their microkernel, placing them instead in a module outside the kernel. The kernel architecture of the Mach operating system is like this; it provides only an interface for reading and writing blocks of main memory and lets a user process determine which blocks to replace. The JavaOS, a small operating system for embedded systems, goes further: it leaves memory management up to the runtime system of the language.

Level 4 implements inter-process communication (IPC). Threads that do not share the same address space and threads running on different computers in a network exchange messages with IPC. Messages are exchanged via ports, which are buffers that retain messages in transit. A message contains the sender port number, the receiver port number, and a data field, enabling the receiver to identify the sender and send a reply.

Level 5 supplies the full-blown process, a program in execution on a virtual computer. A process consists of one or more threads, an address space, one or more input ports for incoming messages, and output ports for outgoing mes-

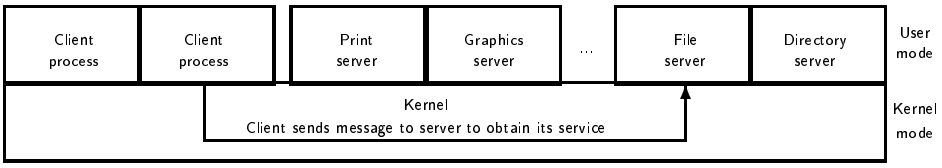


Figure 1: The client-server model on a single machine.

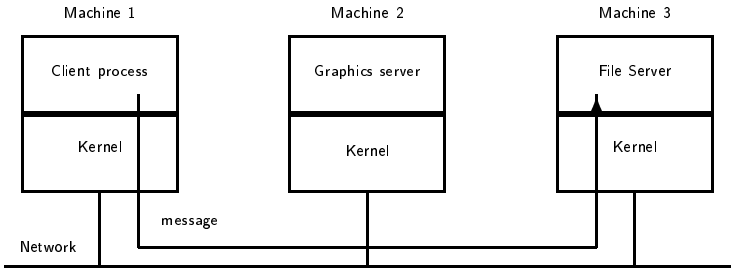


Figure 2: The client-server model in a distributed system.

sages. The threads of a process share access to the address space and the ports. A process can create offspring (child) processes. This level tracks all processes in a tree of parent-child relationships and restricts certain actions accordingly; for example, a parent can force the termination of only its children and cannot terminate unless all its children have terminated.

3.3 User-Level Servers

The remaining levels of the operating system are structured according to the client-server model. In this model certain processes are designated as servers because they perform functions for other processes, called clients, on request. Clients and servers use IPC to exchange requests and responses. Figure 1 shows how clients send messages through the kernel to servers on the same machine; figure 2 illustrates the distributed case.

When a server process is the only server on a machine, the machine itself is often called the server; a common example is a file server. Some servers, such as printer servers, directory servers, mail servers, authentication servers, and web servers, can run alone on a machine or can be part of a group of services offered from a single machine. With this architecture it is possible to run several versions of a server at once—for example, a Unix file server and a Windows NT file server. Because all these servers sit on top of the same microkernel, any client can communicate with any server via the same IPC mechanism.

Although clients and servers communicate with messages, most systems al-

LEVEL	NAME	OBJECTS	EXAMPLE OPERATIONS
9	Graphics server	Window, input event	resize, move, draw, receive, send
8	Shell	application program	Statements in shell language
7	Directories	Directory	Create, delete, enter, remove, search, get
6	Streams	File, device, pipe	Create, delete, open, close, read, write

Table 2: Layers in the user services

low programs to invoke services as procedure calls. This means that all services, local and remote, are called in the same way, and no programmer is required to know which is which. The mechanism for this, called remote procedure call (RPC), is implemented completely by compilers and does not have to be a component of the IPC level. When it encounters a procedure call P(parameters) in a program, the compiler substitutes a call on a "stub" of the form RPC(P,parameters). The stub either calls the procedure P locally, or it sends a message with a copy of the parameters to the remote server on which P resides; that server calls P(parameters) locally and returns then result to the calling stub, which then returns it to its caller.

Level 6 implements a common interface to *information objects*. Info-objects produce, consume, store, or transmit streams (sequences) of bytes. There are three types of info-objects: devices, files, and pipes. Devices are external equipment that either produce or consume streams of bytes; a keyboard is an example of an input (stream-producing) device and a display is an example of an output (stream-consuming) device. A file stores a stream of bytes for an indefinite time. A pipe conveys a stream of bytes from a sender process to a receiver. The importance of treating devices, files, and pipes as info-objects is that they can all be accessed by the same interface: OPEN, CLOSE, READ, and WRITE. OPEN(object) establishes an efficient and fast communication pathway between a process and the object; CLOSE shuts it down. READ(openobject) transfers bytes from the open object to the calling process and WRITE(openobject) transfers bytes from the calling process to the open object. The stream manager calls microkernel services as needed to implement the requested operation. It uses low-level I/O to perform the actual data transfers between address space and secondary storage or external devices. It uses the IPC level to transmit pipe-streams. This architecture makes the three kinds of objects interchangeable. A programmer can say READ(openobject) without knowing in advance whether the object is an open device, a file, or a pipe.

Level 7 manages a hierarchy of directories that catalogs the hardware and software objects to which access must be controlled throughout the network:

files, devices, pipes, ports, processes, and other directories. A directory is a table that matches *external names* of objects to *internal names*. An external name is a string of characters chosen by a user; an internal name is a binary code chosen by the system. Internal names can be guaranteed to be unique for all time, which enables users to share objects without prior arrangements on what to name them. Internal names are generated for every sharable object—file, pipe, device, directory, process, or port. The directory allows the user to deal solely with recognizable, meaningful strings as object names without having to import system names into an address space (where they can be misused or damaged). Pathnames in directory hierarchies can be used to uniquely identify objects.

The directory level is responsible only to record the associations between the external and internal names; other levels manage the objects themselves. Thus, when a directory of devices is searched for the string "laser," the result returned is merely an internal name for the laser printer. The internal name must be passed to a program at level 6 (streams), which handles the actual transmission to that printer.

Level 8 provides command language interfaces called shells. The shell derives its name from a metaphor: it is the layer of software that separates the user from the rest of the machine. The user expresses a command to the shell which, in turn, invokes low-level and kernel service as appropriate to implement the command. The shell is in essence a parser that interprets commands in the syntax of the command language; it creates processes and pipes and connects them with files and devices as needed to carry out the command. A graphical user interface uses point-and-click facilities (windows, icons, mouse, menus) to accomplish the same objective.

At level 9, a graphics server provides user programs with the ability to present pictorial information to the user and coordinate that information with input from user devices such as a keyboard, a mouse, or a joystick. The server enables several programs to share screen space without interfering with one another. Standard libraries have been developed to provide a large number of graphical user interface elements such as buttons, sliders, menus, check-boxes, tree displays, dialog boxes, dials, meters, and table displays—all providing intuitive controls for applications. The distinction between the graphical server and the graphical user interface is that the server provides graphical control and display for applications while the interface provides standardized components for a uniform appearance and behavior of interface elements. These interface elements are displayed by the graphical server.

4 General Comments on Operating System Architecture

4.1 Level 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; input is communicated directly to the lower-level operation without any intermediate level's involvement; and output is returned directly to the caller. The level structure can be completely enforced by a compiler, which inserts procedure calls or expands functions in-line[13]. A well-documented example of its use is XINU, a distributed operating system for microcomputers[6].

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[29]. In the ISO model, data input to a remote operation are passed down through all the layers on the sending machine and back up through all the layers on the receiving machine; return data follow the reverse path. Because each layer adds delay to a data transmission, whether or not that layer's function is required for the transaction, long-haul network protocols are likely to be inefficient in a local network [22]. 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 functions' presence in the system.

4.2 Names

Naming of objects is a very important design problem in operating systems. The system of object names has two principal requirements. (1) It must allow individual users to choose local names as character strings that make sense to them. (2) It must also allow any two users to share an object even though they have no prior agreement on the name each will use for the object. These requirements can be met by allowing objects to have two names: the user-assigned (external) name and a system-assigned (internal) name. Although users can reuse external names at will, internal names must be unique in both space and time: in space because the object name may be passed to anyone anywhere; and in time because object names cannot be reused lest someone access the wrong object.

A two-level mapping scheme is used to convert an external name to an object location. The first level converts user-defined character strings into internal system names; the second level converts system names to object locations. In

the architecture described below, the directories implement the first level and the individual object managers implement the second.

The simplest internal names are bit-strings, called *handles*, generated by the operating system when an object is created. Whenever a program requests creation of an object such as a process, port, pipe, or file, the operating system returns an internal name for that object. The internal name is used to identify the object in subsequent operations. In its simplest form, a handle is a pointer to the object or an index into a table of objects managed by a particular level.

On a network comprises many machines, two extensions to handles are necessary. First, handles are extended by adding extra bits to hold the identifier of the creating machine; this makes handles unique throughout the network. Second, the mapping from handles to object locations must be augmented with search rules to help find objects that reside on other machines: object managers must poll other machines during searches for objects. To speed up multiple accesses to the same object, an object manager can maintain a cache which notes the locations of recently requested objects. Policies of moving or replicating objects to requesting machines and updating caches were explored in the Purdue Ibis[26] and Carnegie-Mellon Andrew[15] file systems, as well as in the Xerox Grapevine system[2].

These simple handles are good for object sharing but inadequate for access control. Nothing prevents anyone from passing a handle for an object to the wrong type manager, or from attempting to overwrite a read-only object. To overcome these limitations, some systems rely on *capabilities* rather than handles. A capability is a handle augmented with type and access codes. These codes can be checked by an object manager to make sure that it performs only the operations allowed by the access code only on its type of object. Fabry [12] advocated capabilities as the most efficient solution to the two naming requirements stated earlier.

Capabilities were explored, among others, in the Carnegie-Mellon Hydra system[33], the Cambridge CAP system[32], and the Intel iMax system[21]. The Amoeba system[30] provides encrypted capabilities for both kernel and user-space objects. Amoeba capabilities can be passed safely across machine boundaries and stored in arbitrary data structures. Objects referred to by capabilities are located by broadcast, with the result cached fur future use.

4.3 Heterogeneous Systems

The systems discussed above deal with many computers on a network by running the same operating system on each machine, an approach often called *homogeneous distributed computing*. In such an environment, sharing information and moving objects among the machines is straightforward.

The open system philosophy, now practiced by many manufacturers of hardware and software, aims for networks whose components can be supplied by different vendors and which will work together anyway because those vendors

FORM OF CALL	EFFECT
dev_handle := INSTALL_DEV(spec)	Add the specification of a new device to the device table.
REMOVE_DEV(dev_handle)	Free up the entry occupied by the given device in the device table.
READ_SEG(mem_addr, dev_handle, dev_addr, size)	Copy *size* bytes from the device, starting at address *dev_addr*, to the segment of memory starting at base *mem_addr*.
WRITE_SEG(mem_addr, dev_handle, dev_addr, size)	Copy *size* bytes from the segment of memory at base *mem_addr* to the device starting at address *dev_addr*.

Table 3: Specification of low-level I/O (level 1)

follow basic standards. These are often called *heterogeneous distributed computing systems* because they may not have the same operating system or internal understanding of formats and structures. To make such an environment work, all machines will have to use a standard Interprocess Communication system (e.g., TCP/IP). They may require translating filters to convert formats and structures as they are sent between machines with different operating systems.

5 A Closer Look

Let us now look a little more deeply into the operations and assumptions of the operating system levels outlined above.

5.1 Low-level I/O: level 1

This level offers simple transfers of blocks of information between devices and the main store (RAM). It hides such details as device startup, parameter passing, device register, device controller management, and interrupts.

Each device has a detailed specification that includes its hardware address, speed and bandwidth parameters, error codes, command codes, and driver. A device driver is a program that interprets the command codes available to the system; for example, the disk driver instructs the disk controller to move the disk arm to a particular cylinder in response to the disk command "seek(diskaddress)". All the device specifications are stored in a device table in a main-memory segment. The operations INSTALL_DEV and REMOVE_DEV are used to add and remove entries from this table.

A device-to-memory transfer is initiated by a READ_SEG operation. It copies a number of bytes given by a size parameter from the device to a segment of memory. Similarly, a WRITE_SEG operation carries out a memory-to-device transfer by copying a specified number of bytes from a memory segment to the device. Many systems support block devices where the basic unit of transfer is a block of data (typically between 512 and 4096 bytes) instead of a byte.

Once the system is booted, new devices can be added by copying their specifications from any storage medium for which there is a driver. During the boot sequence, an initial set of drivers must be loaded as part of the loading of the operating system executable file. This initial set may be determined by probing for known devices, then loading just the drivers corresponding to the devices that are available.

5.2 Threads: level 2

A thread, or primitive process, is described by its *context*, i.e., its stack and the register state. The register state, known also as its stateword, holds the contents of *all* processor registers—including not only the general purpose registers holding program data and addresses, but also the program counter, condition codes, interrupt masks, stack pointer, and any other registers that control or delimit the execution of a program. To run a process, its register state must be loaded into a processor's registers. The operation of saving one thread's state and loading another is called *context switch*.

A thread is in one of four states: running, ready, waiting, or suspended. A running thread controls a processor: its register state has been loaded into a processor, and that process is executing instructions from its program. A ready thread is authorized to execute instructions a processor as soon as one is available. A waiting thread is waiting for a semaphore signal and is ineligible for execution until the signal arrives. A suspended thread is waiting for a signal from its parent. The ready and waiting states are represented by lists — all the threads waiting for a processor are linked to the *ready list* and all those waiting for a particular semaphore are linked to that semaphore's *waiting list*. The ready list is commonly organized as a set of queues for each priority level. When a thread is added to the ready list, it may preempt a lower-priority running thread, returning it to the ready list. Each thread has a private semaphore on which only it can wait. A thread is first created in its suspended state and will not become ready until its creator gives the signal.

Thread priorities are determined partly by the users and partly by the operating system. An example user-set priority is the high priority given to threads that must react quickly to time-critical events (such as completion of an operation on a high-speed I/O device). Examples of system-set priorities are assigning a background thread lower priority than a foreground thread, and the demotion of a thread that has executed for a long time, in a system aiming to favor short jobs. The lowest priority of all is assigned to the *idle threads*. They consist of

FORM OF CALL	EFFECT
thread_handle := T_FORK(addr, priority)	Creates a suspended thread at the given *priority* level by allocating a context for it. Sets its program counter to *addr* and returns a handle to the thread.
T_KILL(thread_handle)	Deletes the given thread (Undoes T_FORK).
T_SUSPEND(thread_handle)	Transfers the given thread into the suspended state, removing it from its current state (running, ready, waiting). Remembers its prior state. (No effect if the thread is already suspended.)
T_RESUME(thread_handle)	Returns the thread to its prior state. A new thread is transferred to the ready state. (No effect if the thread is not suspended.)
sem_handle := CREATE_SEM(val)	Creates a semaphore with *val* as initial non-negative integer counter value and an empty waiting list.
DELETE_SEM(sem_handle)	Removes the given semaphore (undoes CREATE_SEM).
WAIT(sem_handle)	Subtracts 1 from the counter of the given semaphore. If the counter is now negative, WAIT suspends the invoking thread, enqueues it on the semaphore's waiting list, and switches to the next ready thread. Otherwise, returns immediately to caller.
SIGNAL(sem_handle)	Adds 1 to the counter of the given semaphore; if the counter is now negative or zero, transfers a a thread from the semaphore's waiting list to the ready list. Returns immediately to caller.

Table 4: Specification of thread level interface (level 2)

infinite loops of no-ops (empty instructions) and run only if there is no other ready thread; they are needed because otherwise a processor will crash if it has no instructions to execute.

Most operating systems allow many more threads to be created than there are processors. To prevent any one of them from monopolizing a processor, operating systems implement *time slicing*, a policy of limiting the maximum period that a thread can run continuously. At the start of a thread's interval of execution, a timer register in the processor is set to a standard value, called the "time quantum;" the timer triggers a clock interrupt when it reaches zero. The clock interrupt handler (part of the threads level) returns the running thread to the ready list, resets the timer to the time quantum, and switches to the next ready thread.

Requests for I/O are the most common events generated by threads. A thread requesting I/O places a request in the work queue of the device driver, signals the driver, and stops to wait on its private semaphore. The request includes the components

```
requesting thread id
request type (read or write)
memory address
device address
size
```

The device driver will cause the requested transfer to take place; when done, it signals the requesting thread via its private semaphore. The interrupt handler that receives the device's completion signal awakens the device driver from the point where it had paused to wait for the device's completion signal.

I/O is not the only example of thread coordination. Other common examples include: (1) several threads must stop and wait while another thread executes the instructions of an operation on a shared object; (2) a producer thread cannot add items to a full buffer and a consumer thread cannot remove items from an empty buffer; (3) threads borrow resource units from a pool and others must wait if the pool is temporarily empty. The semaphore is a single mechanism that provides simple solutions to these coordinations (and many others).

A semaphore consists of a counter and a queue. The counter records the number of signals sent but not yet received; when it is negative, each thread waiting for a signal is listed in the queue. The operation WAIT implements the request to obtain a signal from the semaphore; SIGNAL provides a signal that can release one waiting thread (if any are waiting). The private semaphore is uniquely and permanently assigned to a thread; its implementation can be simplified because it only needs to record whether or not its owner is waiting on it.

In the first example of thread coordination cited above, threads must be allowed access to shared data one at a time. For example, if two teller machines attempt to add deposits to the same account simultaneously, one deposit will be

lost; which is lost and which is recorded depends on the relative speeds of the two tellers. This problem can be prevented by defining a semaphore *mutex* (for "mutual exclusion") with an initial count of 1, and then bracketing the shared access with and WAIT/SIGNAL pair:

```
WAIT(mutex)
    access to shared data
SIGNAL(mutex)
```

Any section of code that must be constrained to be executed by only one thread at a time is called a *critical section*.

In the third example of thread coordination cited above, threads access a critical section that allocates resource units. The number of resources units is the initial value of a semaphore named *pool*:

```
WAIT(pool)
WAIT(mutex)
    get unit number from free list
    return (unit number)
SIGNAL(mutex)
```

A thread returns a unit to the pool:

```
WAIT(mutex)
    add unit number to free list
SIGNAL(mutex)
SIGNAL(pool)
```

One of the original motivations for semaphores was to avoid *busy waiting* — a form of waiting in which the processor loops while testing for a go-ahead condition. Busy waiting can waste a lot of processor time. The WAIT and SIGNAL operations avoids busy waiting.

5.3 Memory Management: level 3

The simplest responsibility of a memory manager is to partition the memory among address spaces so that parallel threads in different address spaces cannot interfere with one another. This can be accomplished through a method called segmentation. A segment of memory is a region of L (for length) contiguous addresses starting at a base address B. The pair of values (B,L) is called a descriptor for the segment and is part of the register state of a processor. The addressing hardware of the processor checks that a process-generated address A does not exceed L and, if not, presents the address B+A to memory.

Memory managers are often asked to do more than simply partition the main memory among disjoint tasks. They are asked to automatically swap segments between a backing store (usually a disk) and the main memory. This kind of

FORM OF CALL	EFFECT
(B,L) := ALLOC(size)	Return the base and length of a free segment of memory of the given size ($L = size$).
FREE((B,L))	Return the segment (B, L) to the pool of free space in memory.

Table 5: Specification of physical memory (level 3)

memory manager is called virtual memory. By creating the illusion that the entire address space fits into main memory, virtual memory frees programmers from managing swaps.

Two mechanisms are needed to accomplish this. First, every address space must be represented as a table containing one entry for each segment of the address space. An entry tells whether the segment is present in main memory, and if so gives its descriptor. The processor addressing hardware presents the segment number to the memory mapping unit (MMU), which looks up the descriptor in the table and computes the memory address as above. If the MMU finds the segment marked as "missing," it generates a missing-segment addressing fault signal. The fault signal invokes the segment-fault handler (part of this level). Second, the fault handler frees up space in main memory for the new segment by copying one or more segments back to disk (and marking them as missing), then swapping in the new segment (and marking it as present), and then allowing the interrupted thread to resume and retry the address. A full discussion of virtual memory appears in a separate article of this Encyclopedia.

5.4 Inter-process Communication: level 4

Inter-process communication (IPC) is used to exchange messages among threads in different address spaces, on the same or different machines. IPC is the base for client/server communication and remote procedure call (RPC).

Senders expect messages to be delivered even if the receiver is not able to accept messages exactly when they are sent. To accomplish this, messages are not sent directly to processes or threads; they are sent to special buffers called *ports*. Each port has a network-wide, unique identification number. A random large number (96 to 128 bits) is good enough to ensure that two ports have distinct numbers.

Table 7 shows the main operations of IPC. A port is created with CREATE_PORT and deleted with DELETE_PORT. ATTACH_PORT opens a connection to an existing port, either for receiving messages form it or sending

FORM OF CALL	EFFECT
vm_handle := CREATE_VM(size, dev_addr, dev_handle)	Create a new virtual memory of given size, initialized to the contents of the file at the given address on the given secondary storage device.
DELETE_VM(vm_handle)	Delete the given virtual memory and free up space it occupied in the secondary storage system.
A := MAP(V, vm_handle)	Translate the virtual address V generated by a processor into an address A in the main store, using the mapping table of the given virtual memory. If the mapping table says that the block containing V is not present, generate a mapping fault (the fault handler will move the missing block into memory, update the table, and retry the MAP operation).

Table 6: Specification of virtual memory (level 3)

messages to it. To complete this operation, the IPC software must identify the machine on which the port is located. On a small, local network, this can be done by broadcasting a message "who has *port_id*?". On a larger network this is done by consulting a *name server*. A name server is a database that records the associations between ports and machines; it provides an interface for registering ports and looking them up. In either case — broadcast or name server — the IPC software caches the result for fast future lookup.

The operation SEND delivers a message to a port and RECEIVE retrieves it. Both operations are synchronous, meaning that SEND blocks until the message has been copied into the port and that RECEIVE blocks until a message is available in the port. Some operating systems also provide asynchronous versions, in which both SEND and RECEIVE return immediately, providing a return code indicating that the operation is incomplete. In this case, the IPC interface needs additional operations that allow sender and receiver to find out if previous operations have completed.

The operation TRANSMIT combines SEND and RECEIVE; it is useful for RPC. A client uses TRANSMIT to send a message to a server port and wait until a reply is returned. The port at which the reply is expected is sent along with the original message.

When messages are sent to a port on another machine in a network, a *network service* is used. A network service speaks several different protocols for exchanging messages with other computers. When sending a message, it breaks

FORM OF CALL	EFFECT
port_id := CREATE_PORT()	Create a new port with a randomly chosen identification number.
DELETE_PORT(port_id)	delete the port given by *port_id*.
port_handle := ATTACH_PORT(port_id, rw)	open a connection to the port *port_id*; initialize it for sending or receiving, depending on the value of *rw*.
DETACH_PORT(port_id)	drop the connection to the port *port_id*.
SEND(sport_id, sbase, ssize)	send message of length *ssize* beginning at *sbase* to port given by *sport_id*. Block until all data have been transferred to the port.
(rbase,rsize) := RECEIVE(rport_id)	block until a message has arrived at port *rport_id*, then return it in segment (*rbase*, *rsize*).
(rbase,rsize) := TRANSMIT(sport_id, sbase, ssize, rport_id)	send a message of length *ssize* beginning at *sbase* to *sport_id* and block until the reply message has been received from *rport_id*.

Table 7: Specification of IPC (level 4)

large messages into smaller packets, encapsulates them into the appropriate protocol wrappers, forwards the packets one by one to the receiver, and retransmits them if they are lost or corrupted. The receiving machine's network service reassembles the message from the fragments, even if they arrive out of order. A network service provides a host of additional functions. For example, it translates data types from one machine's representation to another's, authenticates other network services, acts as a gateway that bridges different networks, and performs simple name lookup. A network service can be run in kernel or user mode, similar to virtual memory management More detail about network protocols, IPC, and RPC can be found in other articles of this Encyclopedia.

5.5 Processes: level 5

A process is a program in execution on a virtual (simulated) machine. It consists of one or more threads, an address space, and communication ports. The threads within the same process form a "team" that must cooperate toward a common computational goal; they share the same address space and cannot be protected

from one another. When it is created, a process has one thread; it can create and control additional threads with the facilities of the Threads Manager (Level 2). A process's creator also passes it a small set of ports for sending and receiving messages; they can be used later to exchange control information and port identifiers for additional communication channels.

The system keeps track of the creator of each process and restricts process control operations accordingly. The creator of a process is called a "parent" and the new process the "child." A parent may suspend, resume, or kill any of its children but no others. These operations apply to all descendants of the affected child process — suspend and resume apply to all threads of a process; kill applies to the process and all its children. The JOIN operation allows a parent to stop and wait until all its children have completed their tasks; each child uses the EXIT operation to tell the parent it has done so.

The operating systems UNIX and MACH separate process creation into two parts: FORK and EXEC. FORK creates a clone of the parent and EXEC performs a context switch to a child program. This approach is quite powerful but can be expensive on multiprocessor systems without shared memory.

5.6 Stream I/O: level 6

Stream I/O is the common interface to the information objects files, devices, and pipes. Each deals with streams of bytes. The common operations — OPEN, CLOSE, READ, and WRITE — are used to open and close data sources and sinks, to read blocks of data from a source, and to write blocks of data into a sink. The common interface supports *I/O independence*, the principle that READ and WRITE operations can be independent of the type of data source or sink. All READ and WRITE statements in a program refer to stream handles, which are attached to files, devices, and pipes when the program is executed.

This strategy 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 keyboard and can send its output to another file, to a window on a display, or to a printer. Without I/O independence, different versions of a program would have to be written for each possible combination of source and sink.

I/O independence works because files, devices, and pipes all rely on the same model of data: streams (sequences) of bytes. 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 those written thus far; r cannot exceed w. 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 stream I/O interface uses "dynamic dispatch" to route a request to the appropriate file, device, or pipe manager. The dispatch vector points to the type managers for each type of info-object (three in this case). To see how

FORM OF CALL	EFFECT
proc_handle := CREATE_PROC(file_handle, port_list)	Allocates a process control block that points to the thread, address space, and ports of the process. Creates a single suspended, thread and a virtual memory containing the executable file denoted by *file_handle*. Attaches to the ports given in *port_list*. Adds the new *proc_handle* to the list of children of its caller.
KILL(proc_handle)	Terminates the given process, but only if it is a child of the caller: delete the threads and virtual memory and detach from its ports; release its process control block; and delete it from the list of existing children of its parent.
EXIT()	Terminates the caller process and deducts 1 from the UNDONE variable of the parent process.
JOIN(m)	Sets caller's UNDONE variable to m, then waits until it reaches 0.
SUSPEND(proc_handle)	Puts the threads of the given process into the suspended state, but only if the process is a child of the caller.
RESUME(proc_handle)	Puts the threads of the given process back into the state they had at the time of the last SUSPEND operation on the process, but only if the given process is a child of the caller.

Table 8: Specification of process operations (level 5)

this works, take the stream READ as an example. READ is provided with an open-object handle, which contains within it a type indicator of the info-object to which it points; using the dispatch vector, READ then passes control to the device manager, file manager, or pipe manager. A knowledgeable programmer can extend the same interface to deal with a new type of stream object by providing the type manager and adding a pointer to it to the dispatch vector.

The stream model is not used in every operating system. Multics illustrated another approach [20]. Multics had a segmented address space that subsumed the file system. Each segment was permanent, just like a file, and had a unique pathname in a directory tree. The first time a process referred to a segment (via its directory pathname), a "linkage" fault interrupts the process, calling in the linker, which loads the missing segment; thereafter, the process can refer to the segment using ordinary virtual addressing. Certain segments of the address space are permanently bound to devices: read (writing) and one of them reads (writes) the associated device. Fabry offered arguments that explain why the shared-handle approach to naming info-objects leads to a simpler I/O system than Multics had [12]

5.6.1 Files

A file server implements a long-term store for files. Files are named sequences of bytes of known (but arbitrary) lengths that persist until deleted and are accessible from all machines in the network. The file server offers a set of file operations (Table 9) that includes the four generic stream operations.

To establish a connection with a file, a process presents a file handle to the OPEN operation. OPEN contacts the file server via a known port. The file server locates the file in its secondary storage and allocates ports for transmissions between itself and the caller; it may even create and assign a dedicated thread to manage the file session. A READ operation copies the requested data from the file server to the client's buffers (assigned when the file was opened), then copies those data to the caller's address space, and finally updates the file's read pointer. A WRITE operation performs similarly in the reverse direction. The SEEK operation can change the read and write pointers to allow for random (non-sequential) file access. Examples of such remote file implementations are the Berkeley Cocanet System[25] and the Network File System (NFS)[27].

Caching is often used to improve performance of file READ and WRITE operations. When the file is opened, a copy is read into the client's buffers; READ operations do not require further transmissions from the file server. However, the cache must be copied back to the file server shortly after any WRITE operation. Examples are Purdue's Ibis[26] and CMU's Andrew[15].

Some file systems also provide improved synchronization and version control. Synchronization control means implementing a solution to the "multiple readers and writers" problem as part of READ and WRITE operations [14, 30]. Version control means to retain previous versions of a file so that older versions can be

FORM OF CALL	EFFECT
file_handle := CREATE_FILE()	Creates an empty file and returns a handle for it. (If the caller is a process, it can store the handle in a directory entry and make the file available throughout the system.)
DELETE(file_handle)	Deletes the given file (undoes the corresponding CREATE_FILE).
ofile_handle := OPEN(file_handle, rw)	Opens the given file by allocating buffer storage and loading the file index table into main memory. The file is enabled for reading, writing, or both, depending on the value of rw. The read pointer r is set to zero, and the write pointer w to the file's length (fl). (Fails if the file is already open).
CLOSE(ofile_handle)	Undoes OPEN.
READ(ofile_handle, buf, n)	Sets $m := min(fl - r, n)$. Copies m bytes from the given file, starting with position r, into segment *buf*. Updates fl to $fl + m$. (Fails if reading is not enabled.)
WRITE(ofile_handle, buf, n)	Copies the first n bytes of segment *buf* into the given file, starting with position w. Sets $fl := max(fl, w + n)$ and $w := fl$ (Fails if writing is not enabled.)
SEEK(ofile_handle, pos, rw)	Stores the value of *pos* into the read pointer r, the write pointer w, or both, depending on the value of rw. (Fails if *pos* is larger than file length fl.)
ERASE(ofile_handle)	Sets file length, read and write pointers to zero; releases secondary storage blocks occupied by the file. (Fails if writing not enabled.)

Table 9: Interface for files (level 6)

retrieved even after the file is "overwritten " [31].

5.6.2 Devices

The devices level implements a common interface to a wide range of external I/O devices, including keyboards, scanners, displays, printers, plotters, and time-of-day clock. 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—for example, cursor-positioning commands must be embedded in the data stream sent to a graphic display—but a substantial degree of uniformity is possible.

Corresponding to each device is a *device driver program* at Level 1. The stream implementation of device access translates between the model of streams and the device access model assumed by the device driver. (Considerable effort is required to construct reliable, robust device drivers; the stream level does not attempt to duplicate that work, but only to interface with it.) Some devices are operated by dedicated servers — print servers are notable examples — and in these cases the translation is nothing more than a remote procedure call to the server.

The stream devices can add semantics appropriate for the device. For example, a READ operation applied to a keyboard may be programmed to return the next full line of input, regardless of its length.

Table 10 summarizes the interface for external devices. It is similar to the interface for files, without the SEEK and ERASE operations.

5.6.3 Pipes

Pipes move a continuous stream of data from a writer process to a reader process on the same or different machines. The most important property of a pipe is that a reader must stop and wait until a writer has put enough data into the pipe to fill the request. The main difference with IPC is that a pipe provides a continuous stream; IPC would treat a stream as a series of chunks, each transmitted separately.

When the reader and writer processes are on the same machine, a pipe between them can be stored in shared memory and the READ and WRITE operations are implemented in the same way as SEND and RECEIVE operations for message queues [3]. When the two processes are on different machines, IPC ports are used.

The semantics of READ and WRITE operations must be defined even if one end of the pipe is not connected. Should a writer be blocked from entering data until the reader opens its end? What happens if either the reader or writer breaks its connection? Such questions are answered by a *connection protocol*. The specifications in the table above use the "rendezvous on open and close" connection protocol:

FORM OF CALL	EFFECT
dev_handle := CREATE_DEV(type, address)	Creates a control block for the stream interface to a device driver or server. Returns a handle referring to this control block. (If the caller is a process, it can store the handle in a directory entry and make the device available throughout the system.)
DELETE(dev_handle)	Releases the given device control block (undoes the corresponding CREATE_DEV).
odev_handle := OPEN(dev_handle, rw)	Opens the given device by allocating buffer storage and performing setup operations. The device is enabled for reading, writing, or both, depending on the value of *rw* and on whether the device is an input or output device or both. (Fails if the device is already open).
CLOSE(odev_handle)	Undoes OPEN.
READ(odev_handle, buf, n)	Reads n bytes into segment base address *buf*, as for files. (No effect for output device.)
WRITE(odev_handle, buf, n)	Sends the n bytes of segment base address *buf* to the device, as for pipe. (No effect for input device.)

Table 10: Interface for devices (level 6)

FORM OF CALL	EFFECT
pipe_handle := CREATE_PIPE()	Creates a new empty pipe and returns a handle for it. (If the caller is a process, it can store the handle in a directory entry and make the pipe available throughout the system.)
DELETE(pipe_handle)	Deletes the given pipe (undoes the corresponding CREATE_PIPE).
opipe_handle := OPEN(pipe_handle, rw)	Opens the given pipe by allocating buffer storage and performing setup operations. Initially, the pipe is empty. If $rw = read$ the pipe is opened for reading (but only if the pipe is not already open for reading). If $rw = write$, it is opened for writing (but only if the pipe is not already open for writing.) If both reader and writer are on the same machine, the pipe can be implemented in shared memory; otherwise it must be implemented using IPC ports.
CLOSE(opipe_handle)	If executed by the reader: close down the pipe and send an error message to a waiting writer. If executed by the writer: wait until reader empties the pipe, then close it down.
READ(opipe_handle, buf, n)	Waits until there are at least n bytes in the open pipe, then moves them from the pipe into the caller's address space at base address *buf*. Awakens waiting writer if the read has freed up enough space in the pipe to accommodate the writer's request. (Fails if the open pipe does not permit reading.)
WRITE(opipe_handle, buf, n)	Copies n bytes from the caller's address space at base address *buf* into the given pipe. Waits if pipe cannot accommodate the requested bytes. May awaken waiting reader. (Fails if the open pipe does not permit writing.)

Table 11: Interface for pipes (level 6)

- The open-for-reading and the open-for-writing request may be called at different times; each returns immediately.
- 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.

5.7 Directories: level 7

Level 7 manages a hierarchy of directories containing handles for sharable objects. In our model, ports, pipes, files, devices, and directories are sharable; but handles for processes, threads, semaphores, virtual memories, and for open pipes, files, devices, are not sharable and cannot appear in directories. A hierarchy arises because a directory can contain handles for subordinate directories.

A directory is a table that matches an external name, stored as a string of characters, with a handle. (An access code for the object is contained in the handle.) The first ("self") entry of a directory contains a copy of the directory's own handle and the second entry is reserved for the handle of the parent directory. These two handles facilitate certain common operations such as copying an entry to the current directory or changing the current directory to the parent. In a tree of directories (Figure 3), the concatenated sequence of external names from the root to a given object serves as a unique, system-wide external name (pathname) for that object. Since directories are at a higher level than files, the file system can be used to store directories.

The principal directory operation is a search command that locates and returns the handle corresponding to a given external name. Thus, the directory level is merely a mechanism for mapping external names to internal ones. Information about attributes of objects, such as ownership, time of last use, or time of creation, is not kept in directories but rather in the object descriptor blocks within the various object-manager levels.

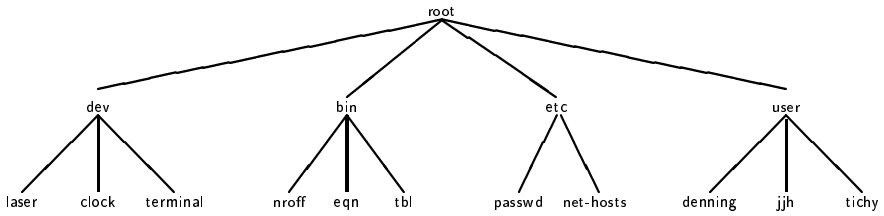
Portions of the directory hierarchy may be replicated across machines. The replication methods must guarantee consistency of the replicated portions; distributed database methods have been used for this purpose [28], as have others [26, 15]. To control the number of update messages in a large system, the full directory database may be kept on only a small number of machines. Other machines can cache the portions of the directory database accessed by their users. Operations that modify an entry in a directory must send updates to the directory-database machines, which relay the updates to other machines.

The specifications for the directory level in the table below are not complete; specific systems will provide other operations as needed.

The CREATE_DIR operation creates an empty, unattached directory. The access codes passed to this operation indicate which classes of processes have the right to search or modify the directory. The ATTACH operation is used to create a new entry in a target directory. When new entry is a directory, this

FORM OF CALL	EFFECT
dir_handle := CREATE_DIR(access)	Allocates an empty directory, sets its access codes to *access*, generates a handle for it, and places a copy of the handle in the first entry. Marks the new directory as unattached to the directory tree.)
DELETE(dir_handle)	Deletes the given directory. (Fails if any entry of the directory is nonempty, other than the self and parent entries, or if the directory is attached to the directory tree.) empty.)
ATTACH(dir_handle, name, obj_handle)	Makes an entry (*name*, *obj_handle*) in the given directory. If *obj_handle* denotes a directory, sets its parent entry to *dir_handle*. Notifies the directory databases of the change. (Fails if *name* already exists in the directory, if the directory *dir_handle* is not attached, if *obj_handle* denotes an already attached directory, of if *obj_handle* denotes a nonsharable object.)
DETACH(dir_handle, name)	Removes the named entry from the given directory. Notifies the directory databases of the change. (Fails if the given name does not exist in the given directory, or if it names a nonempty directory, or if the access codes of the directory prohibit changes.)
obj_handle := SEARCH(dir_handle, name)	Finds the entry of the given name in the given directory and returns a copy of the associated handle. (Fails if the name does not exist in the given directory, or if the access codes of the directory prohibit searching.)
n := COUNT(dir_handle)	Returns the number of entries in the given directory. (Fails if the the access codes of the directory prohibit searching.)
(name, handle) := GET(dir_handle, *i*)	Returns the *i*th entry of the directory. (Fails if there is no such entry or if the entry is blank.)
RENAME(dir_handle, name, newname)	Locates the named entry and replaces the name with the new name. (Fails if the directory access codes prohibit changes.)

Table 12: Specification of a directory manager interface (level 7)



Some directories of the directory tree 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 subdirectories 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 "/" and omitting the root. Thus, the laser printer's external name is "/dev/laser."

Figure 3: A Directory Tree

operation sets that new directory's parent pointer to the target directory. The DETACH operation only removes entries form directories but has no effect on the object to which a handle points; to delete an object, the DELETE operation of the appropriate level must be used. To minimize inadvertent deletions, the operation to delete a directory fails if applied to a directory containing anything else but the self and parent pointers.

The ATTACH and DETACH operations must notify other machines so that changes become effective throughout the system. By maintaining two conditions, this process is unlikely to yield inconsistencies: (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 can construct a directory subtree before attaching its root to the global directory tree. The COUNT and GET operations are used by a formatting program to prepare a summary of the objects listed in a directory.

5.8 Shell: level 8

Most system users spend most of their time executing existing programs, not writing new ones. The shell is the program that listens to the user's console and interprets inputs as commands to invoke existing programs, in specified

combinations with specified inputs. When a user logs in, the operating system creates a 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 scans each complete command line of the 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 process. The processes are connected according to the data flow specified in the command line. Multics was one of the first systems to have a shell [20]; Unix adapted the shell model [24].

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 the file *output*. If "> output" is replaced with "— laser," the data are instead sent directly to the laser printer.

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

```
h1 := SEARCH(CD, "tbl");
h2 := SEARCH(WD, "text");
h3 := CREATE_PIPE();
h4 := SEARCH(CD, "eqn");
h5 := CREATE_PIPE();
h6 := SEARCH(CD, "lptroff");
h7 := CREATE_FILE();
ATTACH(WD, "output", h7);
```

The variable CD holds a handle for a commands directory and WD holds a handle for the current working directory. Both CD and WD are part of the shell's context

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

```
RESUME(CREATE_SHPROC(h1, (OPEN(h2,r), OPEN(h3,w))));
RESUME(CREATE_SHPROC(h4, (OPEN(h3,r), OPEN(h5,w))));
RESUME(CREATE_SHPROC(h6, (OPEN(h5,r), OPEN(h7,w))));
JOIN(3);
```

(CREATE_SHPROC works almost like CREATE_PROC of level 5. It first extracts the ports from the open stream descriptors it its second argument and passes these ports and the first argument, and executable file, to CREATE_PROC.) After the join completes, the shell can kill these processes, close all open objects, and acknowledge completion of the entire command to the user through a "prompt" symbol on the user's display.

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

```
CREATE_SHPROC(h1,(STD_IN, OPEN(h3,w)));
```

where STD_IN is the standard input for a process. Similarly, if "> output" is omitted, the shell connects *lptroff* to the default output, the shell's STD_OUT.

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 *format* 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 *format* is invoked, the variables $1 and $2 are replaced by the arguments following the command name. For example, typing

```
format text output
```

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

5.8.1 Graphics Server: level 9

The graphics server provides a standard way for programs to interact with the user pictorially. The basic abstractions are the window and the event. The window enables a program to display something for the user without interfering with the output of other programs. The event encapsulates information from input devices such as a keyboard, a mouse, or a joystick, and messages sent between windows. The server maintains a policy (focus) for deciding to which window a given event belongs.

A user program can draw into a window regardless of whether it is visible or not. The server tries to maintain the contents of the window. If it loses the content, it can send a redraw event to the user program to request reconstruction of the context.

Each user program normally has an event loop to scan for and react to events. After initializing its windows, the program dedicates a thread to the event loop and to each of the actions linked to events. When an event arrives,

FORM OF CALL	EFFECT
handle := CREATE(display)	Creates a new window on the given display.
DELETE(handle)	Removes the given window and its context from the display and releases its resources.
MOVE(handle, x, y)	Reposition the window at a new location.
RESIZE(handle, width, height)	Change the size of the window.
MOVE(handle, x, y)	Reposition the window at a new location
SHOW(handle)	Make the given window visible by placing it on the display.
HIDE(handle)	Remove the given window and its contents from the screen but maintain its state.
UP(handle)	Reposition the window over the window that most immediately overlaps it.
DOWN(handle)	Reposition the window under the window that it most immediately overlaps.
event := RECEIVE(handle)	Get the next event that was sent to the window from its event queue.
SEND(handle, event)	Send an event to the given window.

Table 13: Graphics Server(level 9)

the loop thread decides what action to take, invokes the corresponding action thread, then waits again. The event loop generally does not terminate until the user program does.

A special program called the window manager may be provided to assist the user. This program makes it easy for the user to manipulate windows on the screen (e.g., resize by dragging a corner, reposition by dragging the title bar), start and end programs, and even provide screen space that is much larger than the actual screen. The user may even be able to choose what window manager to run. For example, the X Window server, offers a wide variety of window managers to choose from (with acronyms including fvwm, twvm, mwm, kde, and gnome).

Window-system "class libraries" provide extensible building blocks for windows. With these libraries, a programmer can construct control windows that make it easier to present information to the user and get the user's feedback. Examples of control-window elements are icons, buttons, sliders, menus, checkboxes, tree displays, and table displays. The *Swing* library in Java is a good example. A windows manager and its class library offer the applications developer a powerful way of organizing user interactions with the application.

6 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 memory from a permanent read-only memory (ROM). This program loads a longer program from 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.

7 Conclusion

We have used the levels model to describe the functions of multi-machine operating systems and how it is possible to systematically hide the physical locations of all sharable objects while being able to locate them quickly when given a name in the directory hierarchy. The directory function is not simply a way of naming files; it is a way of naming any sharable object. No user machine needs to store locally a full copy of the entire directory structure; it needs only save a copy of the view with which it is currently working. The full structure is maintained by a small group of machines implementing a reliable, dependable

storage system.

The model can deal with heterogeneous systems consisting of general purpose user machines, such as workstations, and special purpose machines such as stable storage systems, database servers, file servers, and supercomputers. Only the user machines need a full operating system; the special purpose machines require only a microkernel compatible with the microkernels on the workstations.

The model can also deal with the growing number of real-time systems such as bank customer inquiry systems, web servers, airline reservation systems, transportation monitoring and control systems, manufacturing plant control systems, and hospital patient monitoring systems. The common element in these systems is that each deals with a specified set of external events that trigger system responses, and the system responses must be completed within a specified deadline. The microkernel can be used to field the event signals and trigger the responses; the response programs can be established at the higher levels and use microkernel operations to coordinate their operations.

As computer systems proliferate we will rely more and more on networks of computers. The network will act as a "nervous system" connecting many sensors, actuators, motors, and service nodes. The networks will have to react to sensor input rapidly and effectively. The principles of operating systems, as outlined above, will be used to achieve these ends. Today's operating system principles, understood as software structuring principles, may well become principles for the design of chips and microcomputers — they are that fundamental.

Enterprise computing offers new challenges that can be accommodated within the levels mode. In organizations, the operating system is no longer limited to "managing the flow of work through the network of computers." It is concerned with helping people manage the flow of work in their organization. This form of operating system needs to incorporate a new level, a level that recognizes the distinctions of action in enterprises, a level higher in our hierarchy than the graphics server. The new, enterprise level is higher because it addresses ongoing never-ending coordinated actions among many people; it's also real time as noted in the paragraph above.

The levels mode is powerful because it 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 objects making up those objects. The many parts of an operating system cannot be fully understood without keeping this principle in mind.

References

[1] Accetta, M. et al., "Mach: A New Kernel Foundation for Unix Development," *Proceedings of USENIX 1986 Summer Conference* (Summer 1986), pp. 93–112.

[2] Birrell, A. D. et al., "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM* **25**(4) (April 1982), pp. 260–274.

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

[4] Brown, R. L., Denning, P. J., and Tichy, W. F., "Advanced Operating Systems," IEEE Computer **17**(10) (Oct. 1984), pp. 173–190.

[5] Cheriton, D. R., "The Thoth System: Multi-process Structuring and Portability," Elsevier Science, New York, 1982.

[6] Comer, D., *Operating System Design: The XINU Approach*, Prentice-Hall, Englewood Cliffs, NJ (1984).

[7] Denning, Peter J., "Virtual Memory," *Computing Surveys*, **2**(3) (Sept. 1970), pp. 154-216.

[8] Denning, Peter J., "Third Generation Computer Systems," *Computing Surveys* **3**(4) (December 1971), pp. 175–212.

[9] Denning, Peter J., "Fault-Tolerant Operating Systems," *Computing Surveys* **8**(4) (December 1976), pp. 359–389.

[10] Denning, Peter J., and Tichy, Walter F., "Highly parallel computation," *Science* **250** (30 November 1990), pp. 1217–1222.

[11] Dijkstra, Edsger W., "The Structure of the THE-Multiprogramming System," *Communications of the ACM* **11**(5) (May 1968), pp. 341–346.

[12] Fabry, R. S., "Capability-Based Addressing," *Communications of the ACM* **17**(7) (July 1974), pp. 403–412.

[13] Habermann, A. Nico, Lawrence Flon, and Lee W. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *Communications of the ACM* **19**(5) (May 1976), pp. 266–272.

[14] Holt, R. C., *Concurrent Euclid, Unix, and the Tunis Operating System.* Reading, MA: Addison-Wesley, 1983.

[15] Howard J. H. et al., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, **6**(1) (February 1988), pp. 51–81.

[16] Jones, Anita K. et al., "StarOS, A Multiprocessor Operating System for the Support of Task Forces," *Proceedings of the Seventh Symposium on Operating Systems Principles*, (December 1979), pp. 117–127.

[17] Kernighan, B. W. and R. Pike, *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, N.J., 1984.

[18] Neumann, Peter G., Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson, "A Provably Secure Operating System, its Applications, and Proofs," CSL-116 (2nd edition), SRI International, Menlo Park, CA (May 7, 1980).

[19] Ousterhout, John K. et al., "Medusa: An Experiment in Distributed Operating System Structure," *Communications of the ACM* **23**(2) (February 1980), pp. 92–105.

[20] Organick, E. I., *The Multics System: An Examination of its Structure,* The MIT Press, Cambridge, Mass., 1972.

[21] Organick, E. I., *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, 1983.

[22] Popek, G. et al., "Locus: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, (December 1981), pp. 169–177.

[23] Rashid, R. et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers* **37**(8) (August 1988), pp. 896–908.

[24] Ritchie, D. M. and K. L. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* **17**(7) (July 1974), pp. 365–375.

[25] Rowe, L. A. and Birman, K. P., "A local network based on the UNIX operating system," *IEEE Trans. Software Engineering,* **SE-8**(2) (March 1982), pp. 137–146.

[26] Ruan, Zuwang and Tichy, Walter, "Performance Analysis of File Replication Schemes in Distributed Systems," *Performance Evaluation Review* **15**(1) (May 1987), pp. 205–215.

[27] Sandberg, Russel, et al., "Design and Implementation of the Sun Network Filesystem," *Proceedings of USENIX 1985 Summer Conference* (June 1985), pp. 119–130.

[28] Selinger, P. G.,"Replicated Data," in *Distributed Data Bases*, ed. F. Poole, Cambridge University Press, Cambridge, England (1980), pp. 223-231.

[29] Tanenbaum, Andrew S., *Computer Networks*, 3rd Ed. Prentice-Hall, Englewood Cliffs, NJ (1996).

[30] Tanenbaum, Andrew S., *Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1995).

[31] Tichy, W. F., "RCS—A system for version control", *Software–Practice & Experience*, **15**(7) (July 1985), pp. 637–654.

[32] Wilkes, M. V. and R. M. Needham, *The Cambridge CAP Computer and its Operating System*, Elsevier/North-Holland Publishing Co. (1979).

[33] Wulf, William A., Roy Levin, and Samuel P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill (1981).