AOS research paper

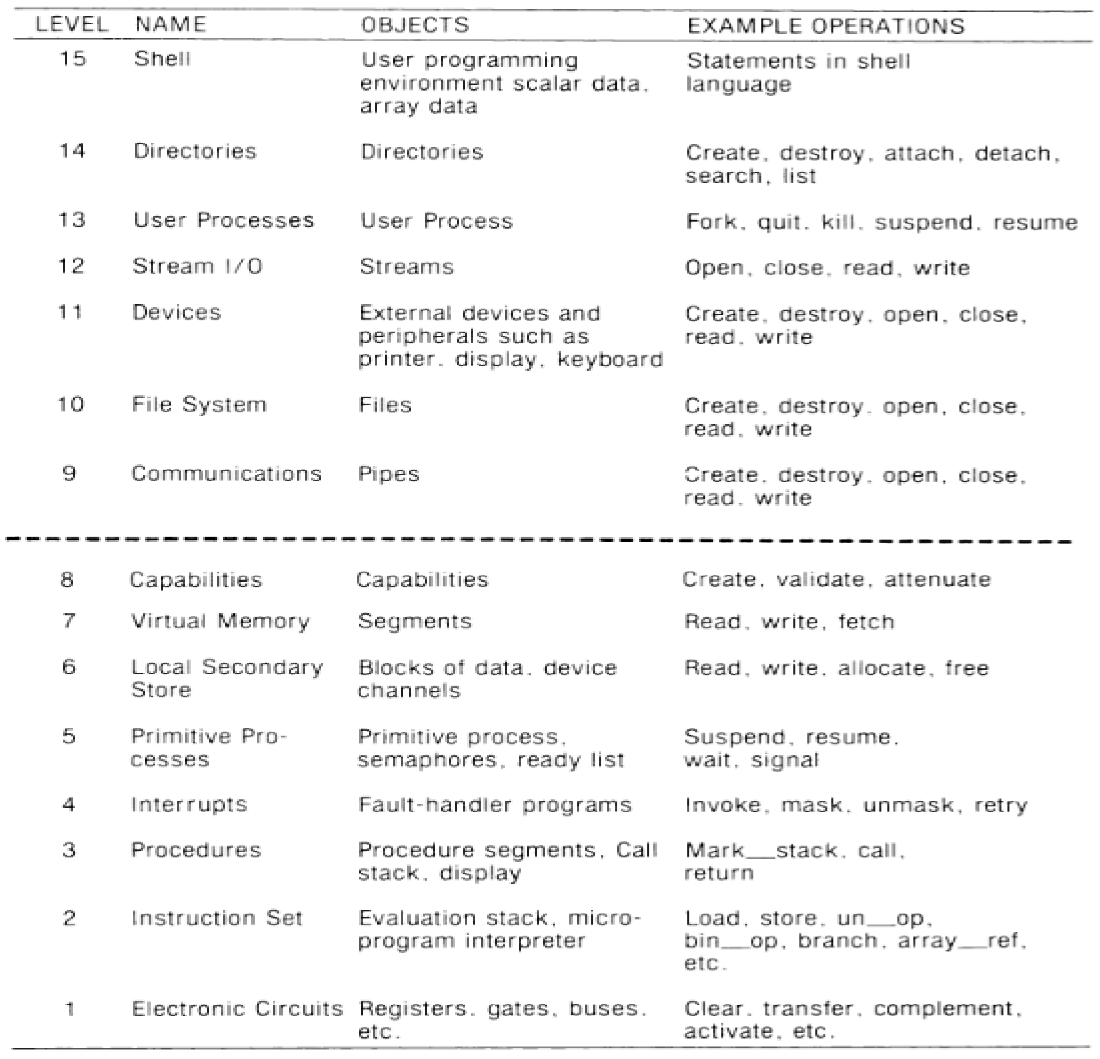
Operating system is "a control program for allocating resources among competing tasks".This is the oldest definition of an OS ,as it describes only a small part of what a modern OS should do so it is also inadequate.

Problem :Main problem faced by the system designer 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.

Earliest Solution is information hiding .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.

The rest of the paper describes a model OS with 14 levels of abstraction of which level 9 to 14 are more important .Operations from level 9 to 14 are called multimachine levels.

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, micro-program interpreter	Load, store, un__op, bin__op, branch, array__ref, etc.
1	Electronic Circuits	Registers, gates, buses, etc.	Clear, transfer, complement, activate, etc.

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:**(important)**

1. Hierarchy: Each level adds new operations to the machine and hides selected operations at lower 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.
2. 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 unauthorized operation to it.

Levels 1 to 8 in Table are called **single machine levels** because their operations are well understood from primitive machines and require little modifications for advanced operating systems.(**not important)**

Level 1 is the electronic circuitry where the objects are gates,memory and registers. operations are clearing gates,memory and registers.

Level 2 adds the processor 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 the process for invoking special procedures when the processor receives an interrupt signal.

(Information about level 5 separately is not given you can read it from the table)

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.

Level 7 is a standard virtual memory.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:(important) 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.**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.

Multimachine levels: 9-14(important): 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.**

Level 9 :Level 9 is explicitly concerned with communication between processes, which can be arranged through a single mechanism called a 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.

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.

When the two processes are on **different machines**, the communications level must implement the network protocols required to move information reliably between machines.

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.

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.

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.**

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.
* **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.

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.
__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. The interface attempts to hide differences in devices by making input devices appear as sources of data streams and output devices as sinks.
Corresponding to each device is a device driver program that translates commands at the interface into instructions for operating that device.When a new device is attached to the system, its physical address is stored in a special file accessible to device drivers.
__Level 12:__**An important principle adopted in the hypothetical operating system described here is I/0 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.
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 bv 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 byREAD or WRITE requests are called **segments.**
Because the stream model(i.e. 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.
__Level 13:__ implements **user processes**,which are virtual machines executing programs. A user process consists of a 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.
A new user process is created by a FORK operation. 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.
__Level 14:__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.

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 systems are a good way to guarantee this consistency. 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 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.

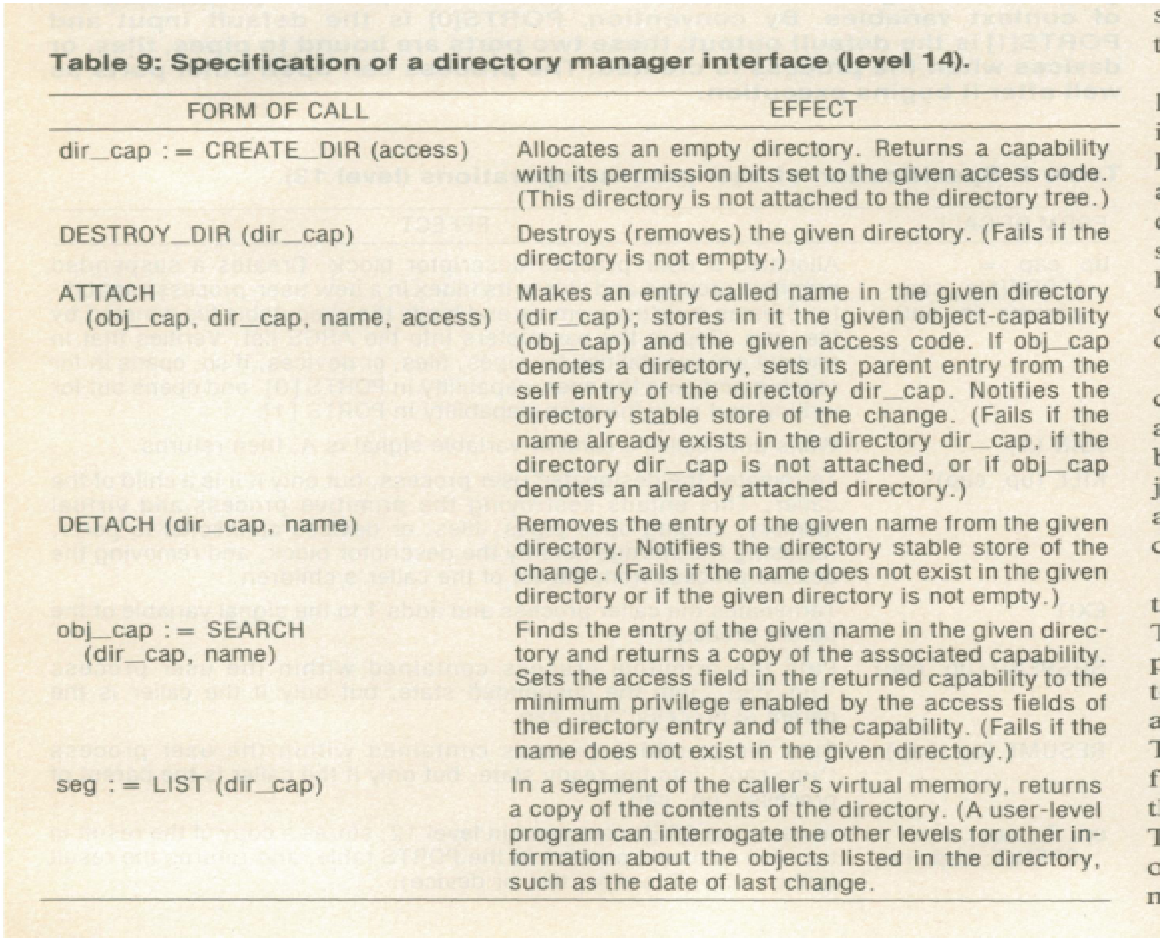


Table 9: Specification of a directory manager interface (level 14).

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.

<u>Shell: level 15:</u>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.