1983

A Model for Coherent Distributed Systems

Robert L. Brown

Peter J. Denning

Walter F. Tichy

Report Number:
83-430

A MODEL FOR COHERENT DISTRIBUTED SYSTEMS

Robert L. Brown[1]
Peter J. Denning[1]
Walter F. Tichy[1]

Computer Sciences Department
Purdue University

Abstract: A model of a complete operating system for multi-machine computer systems is presented. The class of systems modeled is "coherent" because no function depends on the physical location of objects; the class is "fault-tolerant" because the failure of a single machine will not deprive the system of function. The model demonstrates that a hierarchy of object-managing levels can be used to organize the functions of a distributed system, thereby extending the "servers model" studied extensively in distributed system prototypes. The model incorporates new approaches to capabilities, directories, and extended types. It seeks efficiency through simplicity, economy of mechanism, and a verification principle.

January 1983

CSD-TR-430

This paper has been submitted to the 10th Symposium on Operating Systems Principles, to be held in October 1983.

[1] Authors' addresses: Robert L. Brown, Computer Sciences Department, Purdue University, West Lafayette, IN 47907 (net address: rlb@purdue); Peter J. Denning, Computer Sciences Department, Purdue University, West Lafayette, IN 47907 (net address: denning@purdue); Walter F. Tichy, Computer Sciences Department, Purdue University, West Lafayette, IN 47907 (net address: tichy@purdue).

1. INTRODUCTION

In this paper *distributed system* means a collection of computers connected by a local communication network.[2] The computers and network are intended to form a system with personalized working environments, easy communication among users, easy access to special services such as mail and printing, continued operation despite failures of individual computers, and the ability to extend the power of the system in modest steps by adding new computers. A *coherent* distributed system is one in which all system functions are available at any workstation and do not require the user to know the physical locations of objects within the system. A *fault-tolerant* distributed system is one in which major components are replicated so that the failure of a single machine does not deprive the system of function.

Many distributed operating systems do not meet these goals. Some are nothing more than traditional designs with long-haul network software added on, as utilities, to manage communication with other machines. The machines are autonomous and the network is visible in the user environment. Users may have to know the locations of files, to move files among machines using file transfer protocols, to archive files and collect garbage on each machine, to use remote login protocols to gain access to services not available on a given machine, and to use machine names as components of file names and mail addresses. Such systems are fault tolerant but not coherent.

Other distributed operating systems are nothing more than a traditional design adapted by putting important functions on dedicated machines, called "servers," and using the network to simulate procedure calls that invoke them. Examples include name servers, file servers, mail servers, login servers, authentication servers, and printer servers. Such systems hide the machines and the network but the failure of a server can deprive the whole system of an important function. These systems are coherent but not fault tolerant.

[2] The term sometimes denotes an array of microcomputers that work in parallel toward the solution of a common problem (e.g., [illegible]); sometimes computers connected by [illegible] networks; and sometimes multiprocessors sharing memory. These classes of distributed systems are not considered here.

The goals of distributed systems cannot be met without a redesign of the operating system's architecture. Various projects have studied pieces of the problem, notably Medusa [OusS80], StarOS [JonC79], the Xerox Star system [Smil82], Grapevine [BirL82], the Cambridge Ring [WilN79], PSOS [NeuB80], and LOCUS [PopW81]. The Grapevine and LOCUS projects are the most significant among working prototypes in achieving the goals of coherence and fault tolerance without sacrificing performance. The PSOS project is the most general but has not been implemented.

The purpose of this paper is to set forth a model for the architecture of coherent, fault tolerant distributed systems that may contain different types of machines. The model will be presented as a set of functional levels that are managers of the various objects within the system. We will emphasize the conceptual framework rather than details. The new aspects are:

1. An efficient implementation of capabilities for distributed systems.
2. Placement of reliable process-to-process communication in a middle level of the design hierarchy.
3. A global name space for permanent objects, implemented with distributed directories.
4. An efficient approach to data type extension for distributed systems.
5. Integration of a large body of scientific work on systems architecture within a single model.

The "levels model," discussed here, differs from the "servers model" underlying systems like Medusa, StarOS, Cambridge Ring, Xerox Star, and Grapevine. The levels model focuses on the incremental hiding of the network and can accommodate systems containing dedicated servers. It is guided at all levels by two principles:

1. *Efficiency.* Each component should be simple and economical. A program that does not use a given function should experience no overhead from that function's presence in the system. For example: procedure calls should incur cost for validating capabilities only when capabilities are expected; basic system types (e.g., files, directories,

and user processes) should be implemented using standard methods and not type extension; access to local objects should be more efficient than to objects on other machines; and the checks for a given error should not be repeated in many levels of the system.

2. *Verification.* Reasonable steps have been taken to verify that each level of the operating system meets its specifications. Run time checks need be included in system procedures only for conditions that cannot easily be verified a priori. Thus, system procedure calls must verify that expected capabilities are present because the user programs that invoke them may be unverified; but mappings from capability identifiers to objects can be handled directly by system programs without a central mechanism.

These principles are not independent. Hierarchical design makes verification easier; verification reduces the need for run time checking and thereby increases efficiency.

2. OVERVIEW OF A MULTILEVEL OPERATING SYSTEM

The principle of data abstraction -- hiding away the details of managing a class of objects inside a module that has a simple, high-level interface with its users -- was recognized early as an essential tool for maintaining consistency of resource allocation state information. (See [DenV66].) The principle can yield an integrated view of an operating system as a hierarchy of abstract machines from the base hardware to the user interface. The first instance of a multilevel operating system was reported by Dijkstra in 1968 [Dijk68]. The Provably Secure Operating System (PSOS) is the first complete layered system reported and formally proved correct in the open literature [NeuB80].

Table 2.1 is an overview of a 15-level design that will be used in this paper as a basis for discussing coherent distributed systems. This model fits the programming environment principles of UNIX [RitT74] in a framework like PSOS. A brief summary of the levels is given below. For the moment we will assume that the distributed system of interest is a collection of machines running copies of this operating system.

Level	Name	Objects	Example Operations
1	Electronic Circuits	Registers, gates, buses, etc.	clear, transfer, complement, activate, etc.
2	Instruction Set	Evaluation stack, microprogram interpreter, scalar data, array data	load, store, un_op, bin_op, branch, array_ref, etc.
3	Procedures	Procedure segments, Call stack, display	mark, call, return
4	Interrupts	Fault handler programs	invoke, mask, unmask, retry
5	Primitive Processes	Primitive process, semaphores, ready list	suspend, resume, wait, signal
6	Capabilities	Capabilities	create, validate
7	Local Secondary Store	Blocks of data, disk device drivers	read, write, allocate, free
8	Virtual Memory	Segments	read, write, fetch
9	Communications	Channels	open, close, read, write
10	Directories	Directories	create, attach, detach, search, list
11	File System	Files	open, close, read, write
12	Devices	External devices and peripherals such as printer, display, keyboard	open, close, read, write
13	User Processes	User process	fork, quit, kill, suspend, resume
14	Extended Types	Extended type objects (from programming language)	create type-mark, create object, delete object, validate
15	Shell	User programming environment	statements in shell language

TABLE 2.1: An Operating System Design Hierarchy.

Each level in Table 2.1 is the manager for a set of objects of given type; it provides operations for creating, deleting, and changing the states of objects. Levels 1-8 implement hardware-managed objects on each machine. Levels 9-14 implement the principal software objects provided by the operating system; most are sharable among all machines of the system.

The levels must conform to two general rules:

1. *Hierarchy*. Each level adds new operations to the machine and hides more primitive, lower-level operations. The set of operations visible at a given level form the instruction set of an abstract machine that can be used to program operations at that level. Hence, programs can invoke visible operations of lower levels but no operations of higher levels.

2. *Information Hiding*. The details of how an object of given type is represented or where it is stored are hidden inside the level responsible for that type. Hence, no information can be moved into or out of an object except by applying an authorized operation to it.

2.1. The Single-Machine Levels 1-8

Levels 1-8 are called "single-machine levels" because their operations are well understood from single-machine operating systems and require little modification for multi-machine operating systems. The capability mechanism (Level 6) is an exception to this statement.

Levels 1-4 provide a basic machine with procedure calls and an interrupt mechanism. Interrupt signals trigger automatic procedure calls on handler routines.

Level 5 implements primitive processes, which are simply the abstractions of processors. It handles multiplexing of available processors among ready processes. It implements semaphores and the wait and signal operations. It receives signals from external devices and gates them to private semaphores of device driver processes. (This level has a simple hardware implementation [DenD81].) Primitive processes are analogous to "system processes" in PSOS and "lightweight processes" in LOCUS. They are not to be confused with user processes at Level 13, which are the virtual machines containing user programs in execution.

Level 6 implements capabilities, which are object-pointers protected by the hardware. This level allows capabilities to be read, but not altered, and ensures they can only be presented as parameters to procedures of the proper level. Each object created by Levels 9-14 is addressed by a capability issued by the create procedure of the level for that type. Using a validate operation, procedures of this level verify that capability parameters of expected types and access are present on entry. The implementation of capabilities proposed here differs from others because there is no central mechanism for mapping pointers to their associated objects. Details appear in Section 3.

Level 7 contains the device drivers used to manipulate transfers of blocks of information between main and secondary memory on a given machine. These drivers provide a standard interface whereby higher level processes may insert block-move requests in a work queue for the device and be awakened when the transfer is complete [DenD81]. A device driver can be realized as a primitive process running in the device driver code.

Level 8 is a standard segmented virtual memory. A segment pointer is protected, either by tagging or by being stored in read-only tables. When we write

seg := OP(parameters),

we mean that the operation returns a pointer to a segment containing the result. Segments can be shared among address spaces on the same machine but not different machines.

2.2. The Multi-Machine Levels: 9-15

Levels 9-15 are called "multi-machine levels" because they manage objects that can be shared among the machines. Operations familiar from single-machine operating systems must be carefully evaluated for extension to multi-machine systems.

Hiding the maps between names and objects is a central principle in many third generation operating systems. It is responsible for the machine independence of the user environment. To extend the principle for multi-machine systems, the design must hide the locations of all sharable objects (channels, directories, files, devices, user processes, and extended types). This requires the solution of three problems: reliable exchange of information between processes on different machines, global naming of objects, and efficient access to objects. The first problem is solved by the *communications level*, the second by the *directories level*, and the third by *distributing the interpretation of capabilities*. These solutions will be summarized below and studied in later sections.

Level 9 provides a single mechanism, the channel, for reliably exchanging information between processes, independent of whether they are on the same or different machines. A channel appears to its users as a queue of memory segments; a read operation waits until at least one segment has been written into the channel. Channels have properties similar to pipes in UNIX [RitT74] and ports in iMAX [KahC81]. Level 9 is the lowest level at which the communications level has access to the functions of the host machine needed to meet its reliability requirements [DenB83, Come82]. Details are given in Section 4.

Level 10 provides a global directory tree structure and a mechanism for ensuring that portions encached at each machine are consistent. Each entry of a directory contains a name, access list, and a capability. This level can find a directory given a directory capability, but is not responsible for locating any other object. Details are given in Section 5.

Level 11 is a file system extended so that it can open files that may be stored on other machines. Level 12 provides access to other devices such as printers, plotters, terminal keyboards, and terminal displays; it is extended to use channels to connect to devices on other machines. A standard interface for opening, reading, writing, and closing all files, devices, and channels is described in Section 6.

Level 13 implements user processes, which are virtual machines containing programs in execution. A user process includes a primitive process, a virtual memory, a current directory pointer, and parameters passed on invocation. User processes should not be confused with primitive processes (Level 5). User processes are described in Section 7.

Level 14 is an extended types manager. It creates protected type-marks and instances of objects of each type. Because it allows the same capability validation mechanism as in Level 6, a procedure call is no more expensive for extended type objects than for other system operations. Details are given in Section 8. Unlike Hydra [WulL81] or PSOS [NeuB80], our model places type extension close to the user level. This is because of the efficiency principle: since we do not have to deal with the most general case deep inside the operating system, we can use standard implementations of common system objects. The level structure is a hierarchy of functional specifications. The purpose is to impose a high degree of modularity and enable incremental verification, installation, and testing of the software. A program at a given level may directly call any visible operation of a lower level; no information flows through any intermediate level. The level structure can be completely enforced by a compiler, which inserts procedure calls or expands functions in-line [HabF76]. It has been used in, among others, an efficient operating system, XINU, for a small distributed system based on LSI 11/02 machines [Come82].

The level structure discussed here should not be confused with the layer structure of network protocols [Tann81]. In network protocols, information is passed down through all the layers on the sending machine and back up through all the layers on the receiving machine. Each layer adds overhead to a data transmission, whether or not that overhead is required. Models for long-haul network protocol structure may not be efficient in a local network [PopW81].

No one level is capable of efficiently hiding the locations of all objects in a distributed system. For example, the file system must know whether a given file is local or not to be able to perform the most efficient read and write operations. The device level must know the machine to which a given device is connected. The user process level must know whether a given process is to be spawned on the current machine or another. A single central mechanism cannot do all this efficiently. Accordingly, we are led to the principle that each level is responsible for hiding the location of the objects it manages. In turn, this leads to the principles of distributed capabilities that will be defined and illustrated in the next sections.

3. DISTRIBUTED CAPABILITIES

There are four generic requirements on capabilities [DenV66, Fabr74, WilN77, WulL81]:

1. A capability acts like a general, virtual address for an object.
2. The holder of a capability is presumed to have permission to access an object as enabled by access rights in the capability.
3. Capabilities are a low-level tool for implementing data abstraction. They are not visible in user environments, but are images of syntactic structures expressible in the programming language.
4. Hardware must prevent capabilities from being altered and must assure that only the authorized operations are applied to objects denoted by capabilities.

All previous implementations of capabilities take advantage of the shared memory in a single-machine operating system. They are based on a central mechanism that maps a capability to the descriptor block of the object it denotes. The mappings can be implemented by extensions of virtual memory addressing schemes [Denn76].

The concept of a central mapping scheme is incompatible with a distributed system whose component machines may fail. We need instead a method that allows distributing both the objects and the ability to interpret capabilities pointing to them. Distributed interpretation is achieved by allowing each level to read the contents of validated parameter capabilities. Distributed objects are achieved by establishing search rules whereby a machine can find a nonlocal object. These aspects are discussed in the next two subsections.

3.1 Distributed Capability Structure and Interpretation

Our method of distributed interpretation is inspired by the Morris model of sealing and unsealing objects [Morr73] and the Dennis model of protecting capabilities with a tagged architecture [Denn80]. The Morris model requires no central control; the Dennis model requires no changes in compiler technology.

A basic capability consists of three parts: (type, access, identifier). It is stored in a memory word whose tag field is set to "cap". The simplest form of tag is a single bit: 1 denotes "cap", 0 denotes all other types. The type field indicates the type of object; the ten type-marks discussed later are listed in Table 3.1. We will use the notation "T_cap" (e.g. file_cap) to denote a capability of type "T" where "T" is one of the abbreviations in the table.

Level	Type mark	Abbreviation
9	channel	ch
	open channel	op_ch
10	directory	dir
11	file	file
	open file	op_file
12	device	dev
	open device	op_dev
13	user process	up
14	type	type
	extended type	ext

TABLE 3.1: Capability Type Marks and their Abbreviations

The access field defines which operations of the object's level may be applied to the object. The identifier field contains a unique address for the object. Levels 8-13 in the operating system are type managers for objects known to the system: their representations are accessible only to the procedures that compose their type managers. Additional user-defined types are managed by Level 14. The method of known-types increases efficiency by circumventing the generality of the extended type manager when these objects are manipulated.

A capability can be issued only by a create-object procedure, using the CR_CAP instruction to be described below. The type-mark comes from the program status word (PSW), which is set from the procedure descriptor by the call instruction. Only the create-object procedures have non-null type-marks in their procedure descriptors. The general form of the create-object call is

T_cap := CR_T().

The create procedure allocates space for a new object of type T, puts a descriptor block for it in a heap of descriptor blocks owned by the T-object manager, and returns a T-capability whose identifier can be mapped to the descriptor block.

Figure 3.1 illustrates these principles with an object manager authorized to create objects of two types, T1 and T2. The file system is an example: it can create capabilities for files (file_cap) and for open files (op_file_cap). The figure shows that the object x1 is in the local virtual memory and the object x2 is in a segment in the local secondary store. The object x3 has no descriptor, which means it is not local and must be searched for on another machine if someone presents a capability for it.

Because the hardware prohibits altering a capability, and because the object descriptors are private to the object manager, the holder cannot alter a capability or use it to locate an object. A user program may extract the fields of a capability but they are of little use unless a capability for the segment containing the descriptor blocks is available. Normally, this segment capability is private to the object manager.

The holder can present a capability to a procedure in the object manager, which will perform an operation on the object denoted. Corresponding to each parameter expected to be a capability, the called procedure contains a VALIDATE instruction (described below), which checks that the actual parameter is a capability of the required type and its access field authorizes the

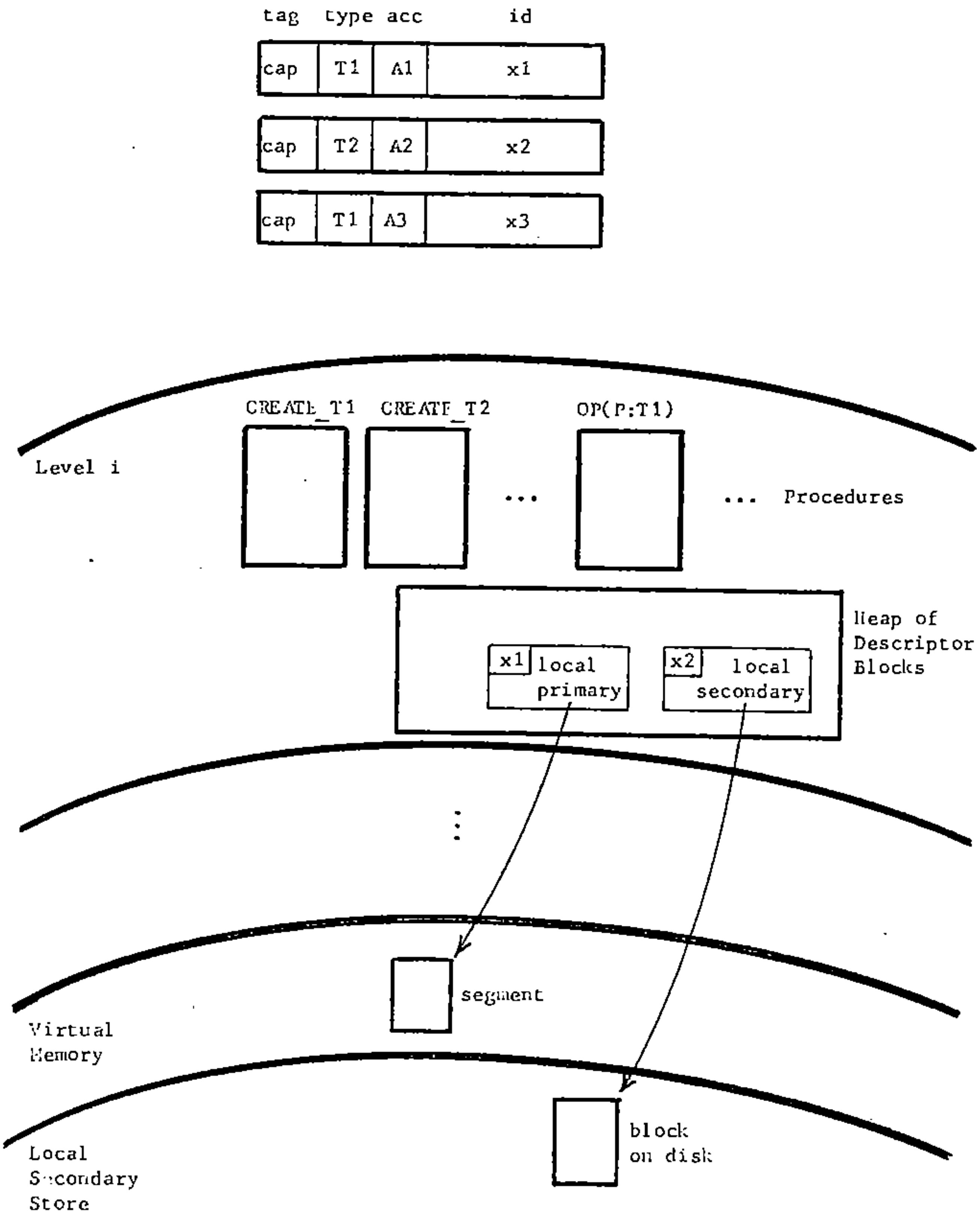

FIGURE 3.1: Principles of distributed capabilities.

call. The procedure in the the object handler can then use the various fields of the capability (e.g. the identifier field) directly as it might use the elements of an aggregate data type. For example, a read operation on a file is programmed as

```
seg := READ(op_file_cap);
```

using the verify operation, the READ procedure checks that the parameter is an open file capability with read access and, if so, copies the contents of the file into a segment in the caller's address space.

Once the type and access fields of the capability have been validated upon procedure entry, they need not be checked again because the procedure is presumed certified by prior verification.

3.2 Distributed Objects

Each object in the system referenced by a capability has a descriptor block associated with it. The create procedure of a level allocates storage for a new instance of a descriptor block in a heap owned by that level on the host machine. It then generates a unique identifier and updates an identifier-to-descriptor mapping table so that other procedures in the level can locate the descriptor quickly when given the capability. Our model does not specify the organization of the mapping table or descriptor block heap. These structures may be optimized differently in each level of the system.

On each valid entry to a procedure in a level, the identifier field of each capability parameters must be mapped to a local descriptor block address. If the mapping procedure cannot find a descriptor block locally, it initiates a poll to find another machine that does. Eventually, the object is located and can be used remotely or moved to the local machine.

The most common instance of mapping failure will occur during the open operations for channels, files, or devices -- either the object does not exist anywhere in the system or it is locked by a process on another machine. Programs must, following standard programming practice, contain appropriate checks for failures of operations that may apply to remote objects.

3.3 Implementation of Capability Operations

Figure 3.2 shows a full encoding of capabilities. The identifier field is refined to (l, m, x) for local bit "l", machine number "m", local index "x". The combination of (type, l, m, x) acts as a global unique identifier among all types; hence, indices x can be generated as needed within each level of each machine.

Form of call	Effect
T_cap := CR_CAP(id,local_bit)	If the type-mark in the current PSW is non-null, create a new capability with type field set to that mark, maximum access code, local bit set to the given value, and the identifier set to the given id. This operation is not available above user processes, Level 13.
VALIDATE(p, n, (T1,a1),...,(Tn,an))	Verify the capability addressed by "p". This capability must be tagged as "cap". For at least one i=1,...,n the following must be true: The capability contains "Ti" in its type field and permits access at least as great as "ai". If the local bit in the capability is set, then the "machine" field must contain the machine identifier of the local system. (Fails if these conditions are not met.)
cap := ATTENUATE(cap, mask)	Returns a copy of the given capability with the access field replaced by the bitwise AND of "mask" and the access field from "cap".

TABLE 3.2: Specification of Capability operations..

Table 3.2 shows the basic operations implemented by the capability level. The create-capability instruction can be executed only by create-object procedures, whose PSW's contain non-null type-marks. The VALIDATE instruction is used to check that a given parameter is a capability of given types and access. The compiler of a procedure generates one VALIDATE

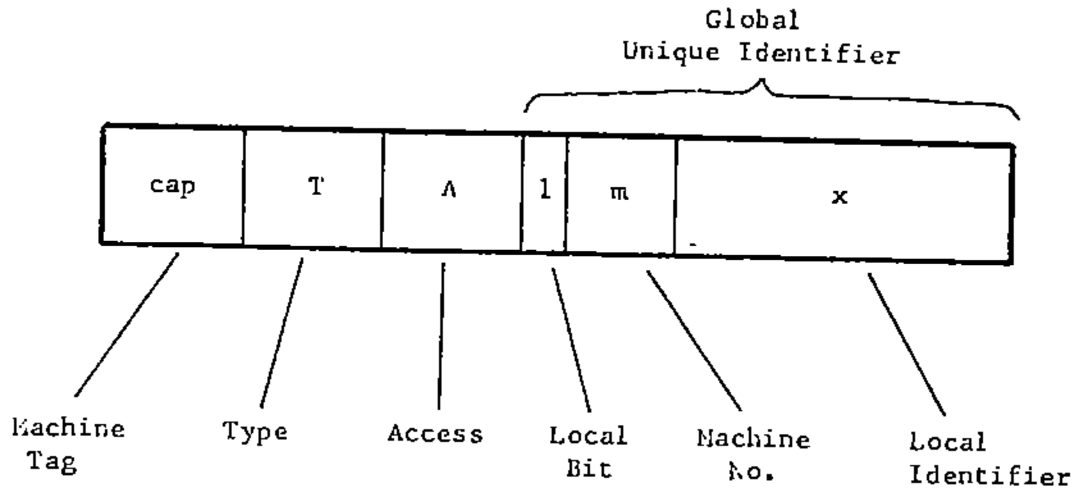

FIGURE 3.2: Format of basic capability.

instruction for each capability parameter. If the local bit is set, this instruction also checks that the value of the machine field matches the local machine number. Figure 3.3 shows an example of a procedure heading in a high level language and the VALIDATE instructions generated by the compiler. The ATTENUATE instruction permits reducing the privilege contained in a capability.

```
High Level Language                Generated Machine Instructions

procedure example(c1,c2,x)         example: validate c1,1,(t1,READ)
t1_cap c1 allows READ;                      validate c2,1,(t2,WRITE)
t2_cap c2 allows WRITE;                          .
integer x;                                       .
begin                                            .
   .                                             .
   .                                             .
end                                         return
```

FIGURE 3.3 Compiler Generated "validate" Instructions

The local bit in the capability, when on, denotes that this capability can only be used correctly by procedures executing on the machine identified in the machine field. When a global capability is converted to a local capability (e.g. by an open procedure), the object is moved (if need be) to the requesting machine and a descriptor block for the object is created on that machine. The local capability then references this descriptor block. Some capabilities always have the local bit set (e.g. the segment capability).

The efficiency principle is satisfied by this model because each level in the system reserves responsibility for the maintenance of its descriptor blocks. The type managers can optimize the organization of the descriptors and mapping from capability indices to heap locations using standard techniques. No overhead is incurred within the type managers after the validate instruction has been performed because rights are not checked at each access to the object; program verification compensates for any loss of correctness this may imply.

4. THE COMMUNICATIONS LEVEL

The communications level provides a single mechanism for exchanging information between two processes, independent of whether they are on the same or different machines.

The external interface presented by the communications layer is suggested by Figure 4.1, which shows Process 1 sending a sequence of segments to Process 2. The sequence is moved across a *channel*, which is an object created and managed by the communications layer. When the two processes are on the same machine, the queue of segments is in shared memory, whereupon the READ and WRITE operations reduce to the familiar "send" and "receive" for message queues [Brin73].

Figure 4.2 illustrates what happens when the two processes are on different machines. The communications layer must implement the network protocols required to move information reliably between machines. These protocols can be greatly simplified compared to 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 [PopW81]

Specifications for six channel operations are outlined in Table 4.1. There are commands to create and delete channels. A channel capability can be passed to another machine (over an existing open channel) for later use by a companion process on that machine; a channel capability can also be listed in the directory hierarchy, whereupon the channel becomes accessible throughout the system. There are commands to open and close channels: the sender and receiver must each open the channel; at most one sender and one receiver are allowed. And there are read and write commands for moving a segment of information across the channel.

If the sender and receiver are on different machines, a connection protocol assures the consistency of the two open channel control blocks and a network protocol manages information transfer. If the sender and receiver are on the same machine, the segments can be exchanged by passing pointers in shared memory without the overhead of network protocols. The open channel control block tells the READ and WRITE operations which case applies.

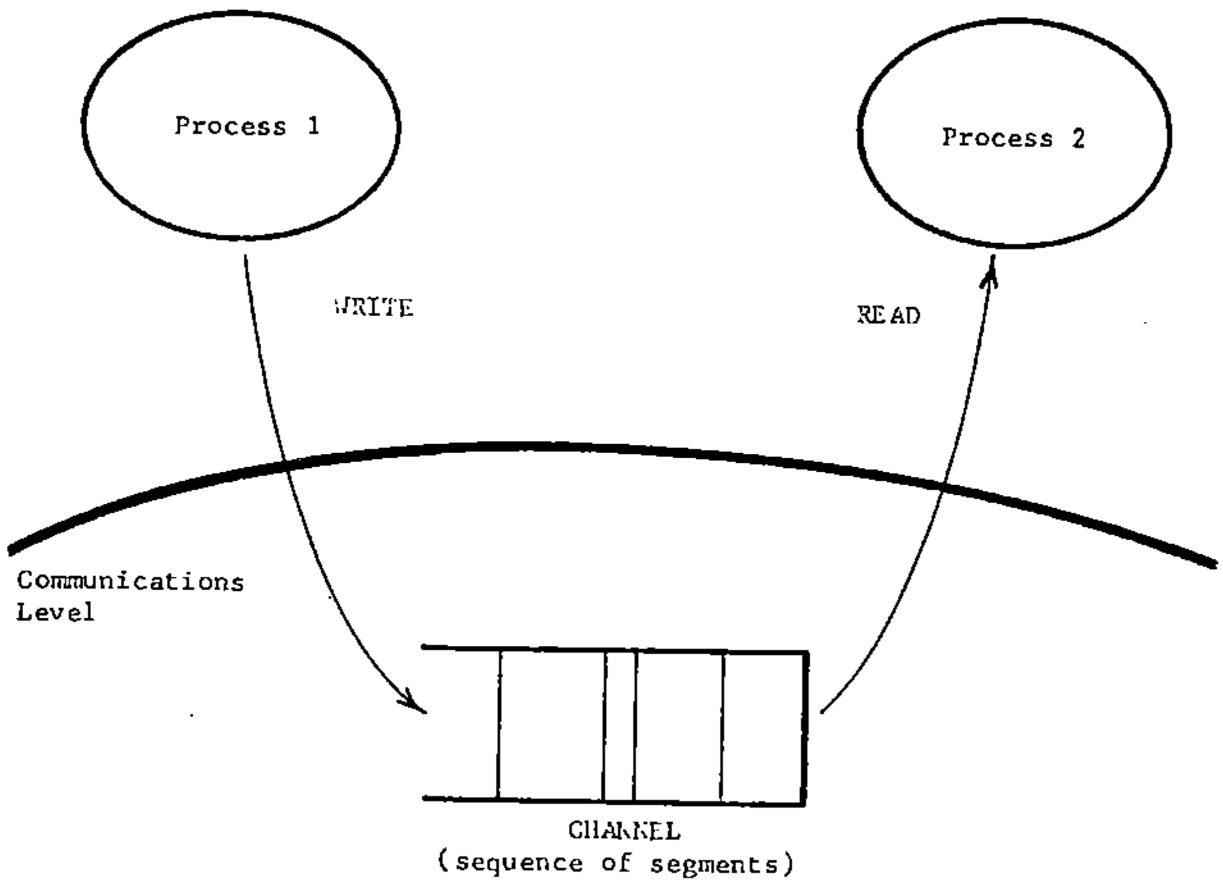

FIGURE 4.1: User's view of a channel.

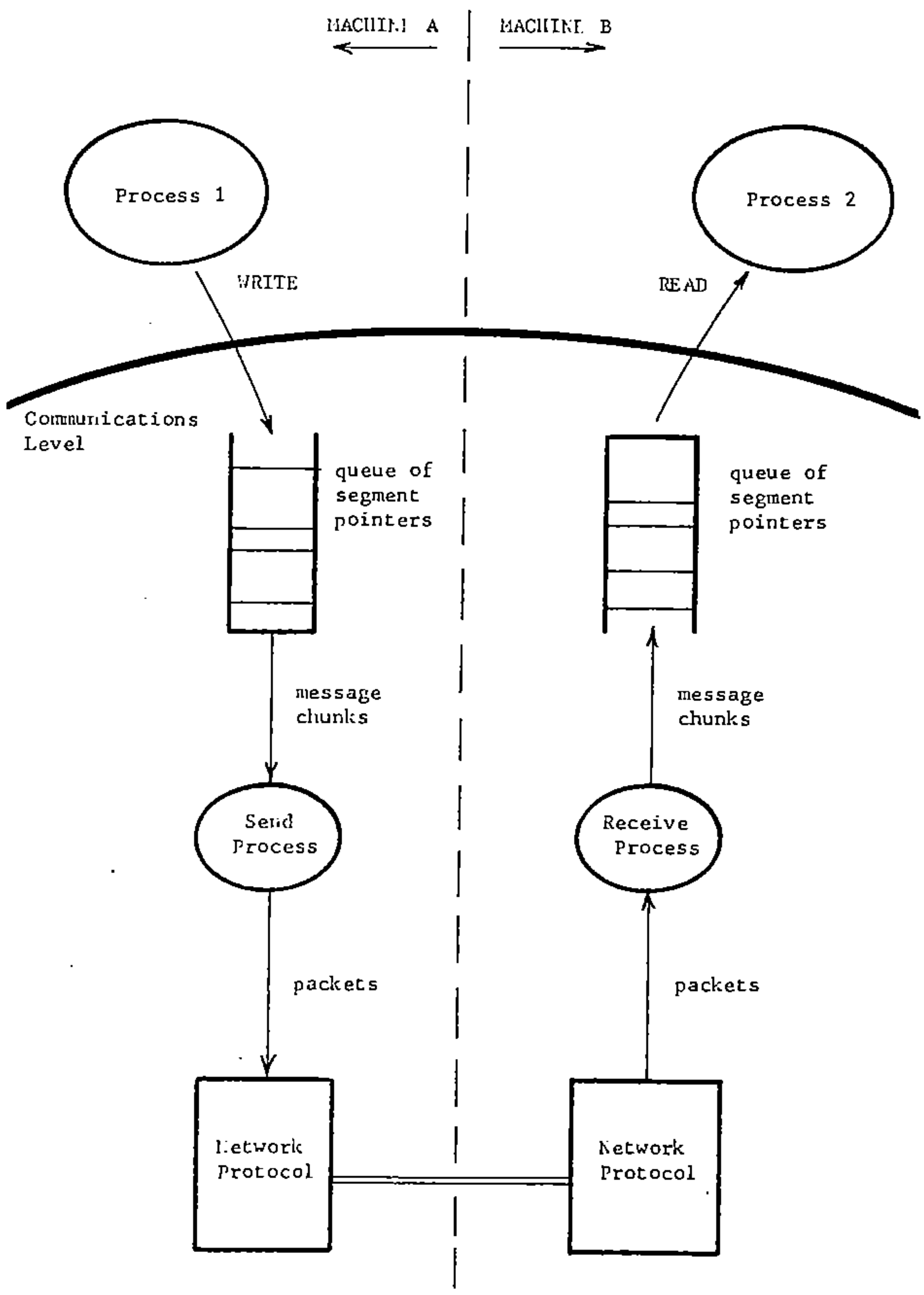

FIGURE 4.2: Internal view of communications level.

Form of call	Effect
ch_cap := CR_CH()	Creates a new channel and returns a channel capability for it; if the caller then stores this capability in a directory entry, the channel becomes available throughout the system.
DELETE_CH(ch_cap)	Undoes a create channel operation.
op_ch_cap := OPEN(ch_cap, rw)	Opens the channel named by the channel capability; returns an open channel capability with write permission enabled if rw="write" and read permission enabled if rw="read". (Fails if the channel is already open for writing when rw="write" or reading when rw="read".) If both sender and receiver are on the same machine, the open channel control block will indicate that segments can be transferred directly from sender to receiver.
CLOSE(op_ch_cap)	Undoes the open channel operation. (Uses the write/read permission bits in the capability determine which open operation to undo.)
WRITE(seg, op_ch_cap)	Causes the segment named by the given segment to be transmitted over the given open channel. (Fails if the open channel capability does not contain write permission.)
seg := READ(op_ch_cap)	Waits until there is a segment in the channel to receive, then returns a pointer to it. (Fails if the open channel capability does not contain read permission.)

TABLE 4.1: Specification of Communication Level Interface.

Because user processes are defined at a higher level, it is not possible to open a connection directly to another user process. It is also not normally possible to open a connection directly to a primitive process. they are short-lived and not listed in the directory hierarchy.

It is possible to store a capability in a segment and send that segment over a channel to another machine. If the capability is of a type that can only be interpreted locally (i.e., on the machine that issued it), the other machines must refuse to interpret it. For example, a segment or open-channel capability defined on one machine does not map to a meaningful object on another machine. This requirement is easy to implement in a machine's hardware because capabilities contain the identifier of the issuing machine and a local bit; the READ and WRITE operations do not check whether capabilities are in segments sent over channels.

If the design hierarchy were altered to allow global segments, channels would still be necessary: a communications layer would be required for reliable updating of directories on all machines and for synchronizing a sender and receiver. (The directory update problem will be described in the next section.)

5. THE DIRECTORY LEVEL

The directory level implements a systemwide directory structure that permits tree pathnames to be used as global names for any permanent object. Figure 5.1 shows that each entry of a directory contains Name, Access, and Capability fields for each object listed. The feasibility of a capability based directory system has been demonstrated in CAP [WilN79, NeeB77]. A directory containing only the self and parent entries is considered "empty".

Only capabilities for permanent objects may be placed in a directory. In the hierarchy of Table 2.1, this includes channels, directories, files, devices, user processes, type-marks, and extended types; it excludes segment pointers and capabilities for open channels, open files, and open devices, which have meaning only on the machines that issued them. Information about object attributes, such as ownership or time of last use, is kept in the object descriptor blocks maintained by the object manager levels.

The directory layer simply stores global capabilities but does not attempt to interpret them. The responsibility for mapping a capability to an object lies with the level that manages that type of object. The directory level cannot locate any object except a directory.

The directory layer also has the responsibility for ensuring that the directory hierarchy is consistent across all machines of the system. This can be accomplished by methods for

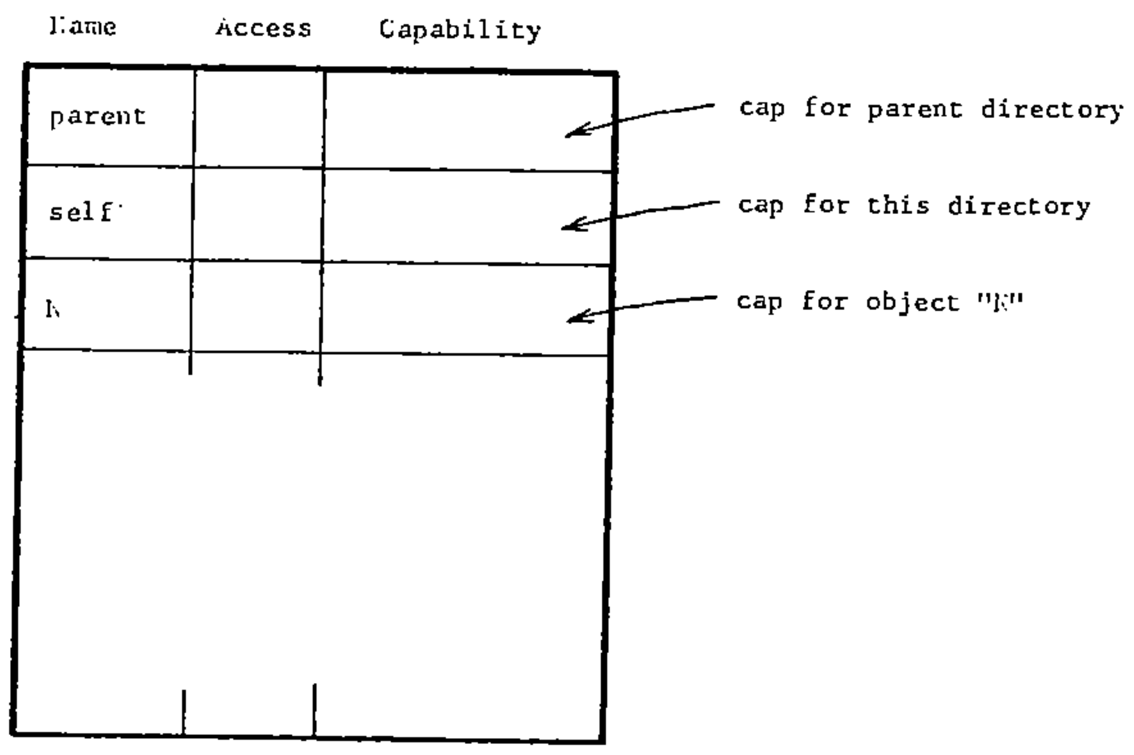

FIGURE 5.1: Format of a directory.

replication in a distributed database system [Sch80]. To control the number of update messages in a large system, the full directory database would be kept on only a small subset of machines (e.g., two or three) implementing a "stable store". Copies of the views of the directory database being accessed by a given user can be stored locally in a workstation after that user logs in. Operations that modify an entry in a nonlocal directory must send updates to the stable-store machines, which relay them to affected workstations.

A specification of the external operations of a directory level is given in Table 5.1. The specifications allow higher levels to create objects and store capabilities for them in directories. The ATTACH operation is used to enter an object capability into a directory under a given name; the DETACH operation undoes this. If the affected directory is nonlocal, both these operations must notify the stable store so that changes become effective throughout the system. Unattached objects will not be retained after termination of the user process that created them.

The ATTACH operation allows its caller to specify an access code that will apply to this entry and may reduce privileges enabled in the capability's access field. An access code can be complex, like Access Control Lists in Multics, or simple, like owner-group-public bits in UNIX. The access field of the capability returned by a SEARCH operation will be the AND of the access code pertaining to the owner of its caller and the access field already in the capability.

The ATTACH and DETACH operations are more complex when the object being attached is a directory. On creation, a directory capability is "local" and can be interpreted only on the creating machine. User processes can build directory subtrees rooted on a local directory; the ATTACH operations involved do not notify other machines. When a local directory is attached to a nonlocal directory, ATTACH must traverse the entire (local) subtree rooted at the (local) directory and notify the other machines; in so doing it must convert directory capabilities contained therein to nonlocal form.[3] The ATTACH operation must also define the parent of the newly

[3] This strategy incurs no additional cost relative to a strategy that notifies other machines as each entry is made in a directory; the same set of notifications must be issued sooner or later.

Form of Call	Effect
dir_cap := CR_DIR(access)	Allocate an empty directory with its permission bits set to the given access code. Return a capability for it. (This directory is not attached to the directory tree.)
DELETE_DIR(dir_cap)	Remove the given directory. (Fails if the directory is nonempty.)
ATTACH(obj_cap, dir_cap, access, name)	Make an entry of the given name in the given directory; store the given object capability and given access code in it. (Fails if the name already exists in the given directory or if the object is a directory whose parent is defined.) If the given directory is nonlocal, notify the update processes in other machines of the new entry for the given object. If the given object is a local directory: mark it as nonlocal and notify the update processes on other machines of the entire subtree rooted at this directory.
DETACH(dir_cap, name)	Remove the entry of the given name from the given directory. (Fails if the name does not exist in the given directory or if the named object is a nonempty directory.) If the given directory is nonlocal, notify the update processes on other machines.
obj_cap := SEARCH(dir_cap, name)	Find the entry of the given name in the given directory and return a copy of the capability stored therein. Set the access field in this capability to the minimum privilege enabled by the access fields of the directory entry and the capability. (Fails if the name does not exist in the given directory.)
seg := LIST(dir_cap)	Return 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 -- e.g., date of last change.)

TABLE 5.1: Specification of a Directory Manager Interface.

attached directory; ATTACH fails if a parent is already defined.

The DETACH operation only removes entries from directories; it does not delete the object to which the capability points. To delete an object, the DELETE operation of the level that manages that type of object must be used. To minimize inadvertent deletions, DETACH and DELETE operations fail if applied to nonempty directories.

The SEARCH operation returns the capability stored with a given name in a directory; a search usually precedes other operations on an object, e.g., opening and reading a file. The LIST operation provides the raw data used by a formatting program to prepare a summary of the objects listed in a directory.

The specifications in Table 5.1 are not intended to be a complete set of operations for a directory manager. (For example, we specified no command to change the access field in a directory entry.) The purpose is to illustrate the possibility of replicating the directory structure consistently among several machines.

6. FILES AND DEVICES

The simplest model of a file system assumes that a file must be opened to be read or written, a file may be open by at most one process at a time, read operations return a copy of the entire file in a segment, and write operations replace the file with a new version contained in a segment. (This is part of the "version model" of objects in many distributed database systems [ReeS80].)

Likewise, a simple model of device management assumes that a device must be opened to be read or written, at most one process can open a device at a time, a device driver for a readable device returns the results of read operations in segments, and a device driver for writeable devices assumes each portion of output is contained in a segment.

Because of the similarity of the channel, file, and device models, it is possible to define a single, generic interface for these three types of objects. A specification appears in Table 6.1. A generic interface is important because it permits the three types of objects to be connected interchangeably to a process. A programmer need not know whether the input (output) stream of a program comes from (goes to) a channel, file, or device. By achieving independence from the medium through which input and output streams flow, the interface helps make many programs look like software parts usable in many contexts. This property, called ***input-output independence***, is critical to the success of UNIX [RitT74].

Form of Call	Effect
op_T_cap := OPEN(T_cap, rw)	Open a connection to the object of type T for reading (if rw="read") or writing (if rw="write"). Return a local capability pointing to the open connection. The access code in the open connection capability is set to the value of rw.
CLOSE(op_T_cap)	Close the connection specified by the given open connection capability.
seg := READ(op_T_cap)	Store a copy of the state of the given, open object in a segment and return a pointer to it. (Fails if the given open connection capability does not enable reading.)
WRITE(op_T_cap, seg)	Set the state of the given open object to the value contained in the given segment. (Fails if the given open connection capability does not enable writing.)

TABLE 6.1: Specification of a Generic Channel, File, and Device Interface.

In a multi-machine system, the file and device levels must deal with a problem not present in single-machine systems: nonlocal objects. Suppose a process opens a connection to a file located on a different machine. What happens? There are two alternatives:

1. Open a pair of channels to a process on the file's home machine; the read and write command are relayed via the forward channel to that surrogate process for remote execution; results are passed back over the reverse channel.

2. 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.

Both methods are feasible. An instance of the first is in the Berkeley Version 4.2 UNIX system [JoyC82]. An instance of the second is the Purdue STORK file system [ParT82].

Similarly, suppose a process opens a connection to a device on a different machine. What happens? Here, the second alternative for files is not open because devices cannot migrate. Only the first is feasible.

The open connection control block for a channel, file, or device indicates whether READ and WRITE operations can be performed locally or must interact with a surrogate process on another machine through an open-channel capability embedded in that control block. Because the open-connection capability is local, the READ and WRITE operations do not have to deal with the problem of finding a nonlocal object; they simply access the object through the control block.

Figure 6.1 illustrates the types of capabilities generated and used during a typical session, editing a file. Capabilities are shown in abbreviated format (cap, type, access, identifier). The file system's heap contains descriptor blocks for local file x, nonlocal file x', local open file y, and nonlocal open file y'. The descriptor for y' contains an open-channel capability to a process on the machine on which the file resides. The steps in the editing session are:

1. Obtain a capability c1 = (cap, file, RW, x) for the file by searching a directory.
2. Open the file for reading, obtaining the capability c2 = (cap, op_file, R, y) := OPEN(c1, read).
3. Open the file for writing, obtaining the capability c3 = (cap, op_file, W, y) := OPEN(c1, write).
4. Read the file, obtaining a segment pointer s = (seg, RW, z) := READ(c2) to a copy of the whole file.
5. Edit the contents of the segment.
6. Using an operation WRITE(c3, s), replace the file with the contents of the edited segment.

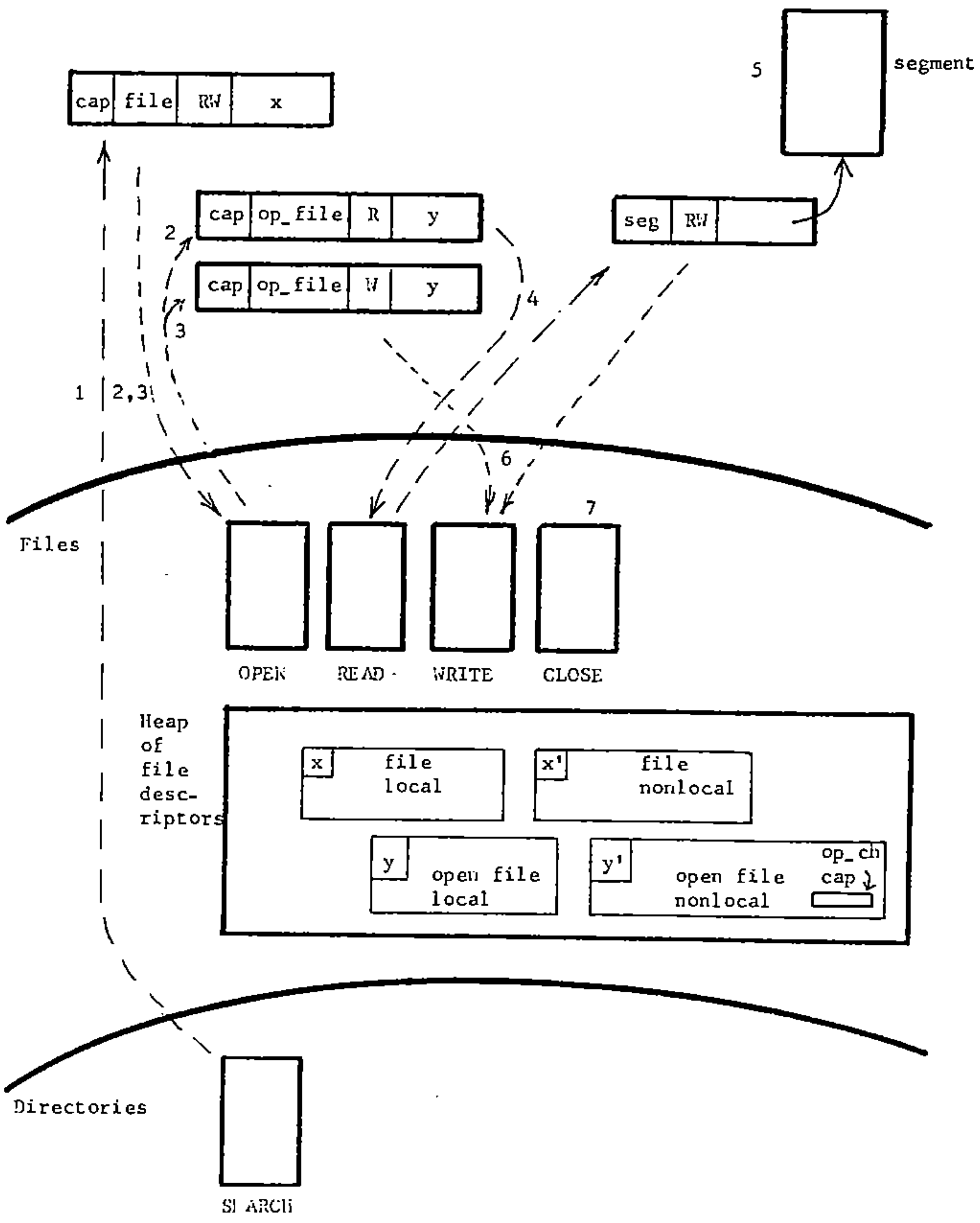

FIGURE 6.1: Illustration of an editing session.

7. Close the file. Delete the capabilities and the segment.

The basic file system operations can be improved in two ways. One is to allow multiple readers and writers. Another is to retain different revisions of a file using a version control system [Tich82].

7. EXTENDED TYPES

The extended types level supports user-defined, abstract data types. Level 6 provides capabilities that efficiently protect a small set of basic types known to the operating system. Level 14 extends these capabilities.

Extended capabilities have the same requirements as listed in Section 3. They act like protected, virtual addresses for shared objects. The objects are created and manipulated by software packages stored in libraries. While we are willing to assume the operations contained in those packages are verified, we are not willing assume user programs will call them with the proper access rights or the proper parameters. Type and access errors cannot be prevented outside extended-type packages because most programming languages lack facilities to express such restrictions -- and because language systems do not usually enforce these restrictions rigorously, even if they can be specified. Therefore, extended-type operations must check types and access rights of capability parameters.

Our approach takes a middle ground between a pure compile-time check as proposed in [JonL76] and a pure runtime check, as implemented in Hydra [WulL81] or CAP [WilN79]. Pure compile-time checking is unrealistic because most languages and language systems in use today do not support access controls on typed objects. All known implementations of pure runtime checking are too expensive. Our model is language independent and can be implemented with adequate efficiency. It assumes that a package has been verified before use, and that a package managing objects of a certain type is the only program that may generate capabilities for that type.

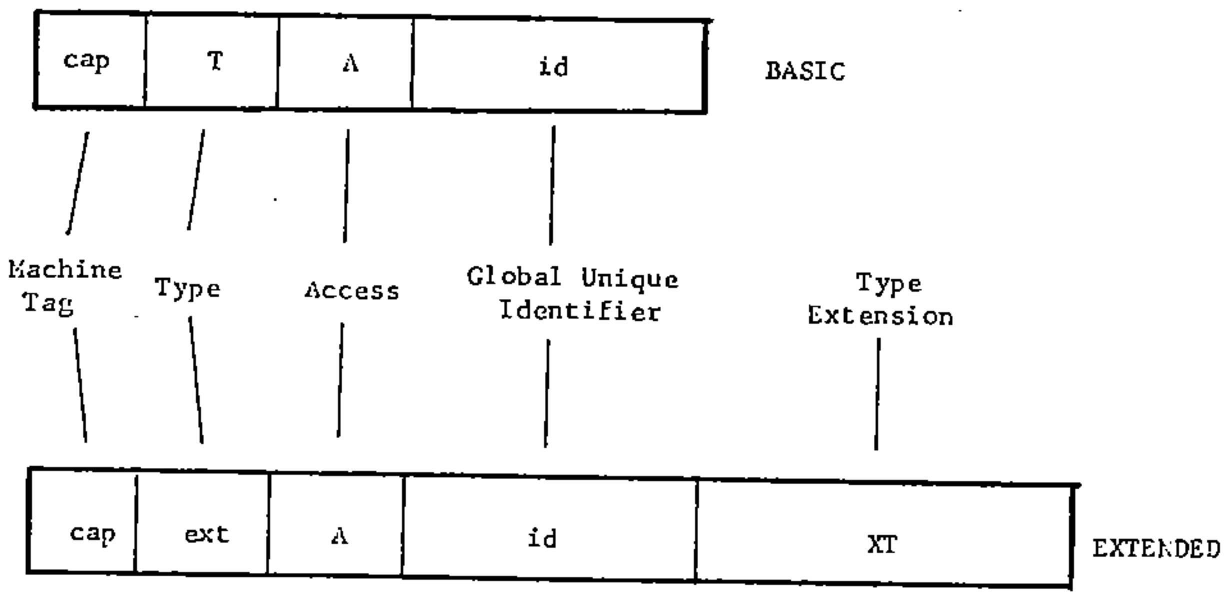

FIGURE 7.1: Formats of basic and extended capabilities.

Figure 7.1 compares basic and extended capabilities. The only difference is that the extended capability has a type extension field that stores a type-mark (XT). The original type field is set to "ext" to indicate the presence of an extension field.

Form of call	Effect
type_cap := CR_TCAP()	Create a new capability with the type field set to "type", an empty access field, and a unique id derived from the machine number and the clock.
ext_cap := CR_XCAP(type_cap, local_bit)	Create a new extended capability with a unique identifier derived from the given local bit, the machine number, and the clock. The type extension field is identical to the identifier contained in the given type-capability. The access field permits maximum access.
X_VALIDATE(c, type_cap, a)	Verify the extended capability addressed by "c". This capability must be tagged as "cap", contain "ext" in the type field, contain type_cap.id in the type extension field, and permit access at least as great as "a". Fails if these conditions are not met.
ext_cap := X_ATTENUATE(ext_cap, mask)	Return a copy of the given extended capability with the access field replaced by the bit-wise AND of "mask" and the access field from "ext_cap".

TABLE 7.1: Extended Capability Operations

The interface presented by the extended types level is given in Table 7.1. The function CR_TCAP returns a type-capability. Its type field is set to "type" and its identifier field contains a unique type-mark. Its access bits are irrelevant.

The function CR_XCAP generates an extended capability. It sets the type to "ext", stores the identifier of the given type-capability into the extension field, and constructs a new identifier out of the given local bit, the machine number, and a unique number derived from the clock. CR_XCAP should encrypt the clock value as an additional measure of protection against attempts to forge capabilities by tampering with the clock. (Encryption makes it extremely difficult to determine what clock value produced a given identifier.)

The functions X_VALIDATE and X_ATTENUATE work like VALIDATE and ATTENUATE, respectively, except that they operate on extended capabilities. X_VALIDATE is used to check capability parameters in the same way as VALIDATE (see Figure 3.3). For fast execution, it should be implemented as a machine instruction.

A programmer who decides to make a new package publicly available must obtain a unique type-capability and register it in a standard directory. For example,

```
ATTACH(CR_TCAP(), "/type", access, "T")
```

creates a new type-capability and stores it with name "T" in the directory "/type".

The create operation in a package for an extended type must first allocate a new object and its descriptor block. Then it issues an extended-type capability with the proper type-mark by executing

```
ext_cap := CR_XCAP(SEARCH("/type", "T"), local_bit).
```

Finally, it reads the identifier from this new capability and enters it into a hashtable to define a mapping from the capability to the corresponding descriptor block. If the local bit is off, the programmer must provide additional software to help other procedures of the package look up capabilities on other machines.

This model assumes that a package is a trusted subsystem which has been verified to handle its objects according to specifications and does not collaborate with other packages to circumvent the checking of capabilities. The model does not permit strong solutions to the mutual suspicion problem, the modification problem, or the confinement problem [WulL81]. However, the assumption that verified packages can be trusted is reasonable in practice. Within this model, capabilities need be checked only at procedure entry, thereby permitting an efficient implementation of extended types.

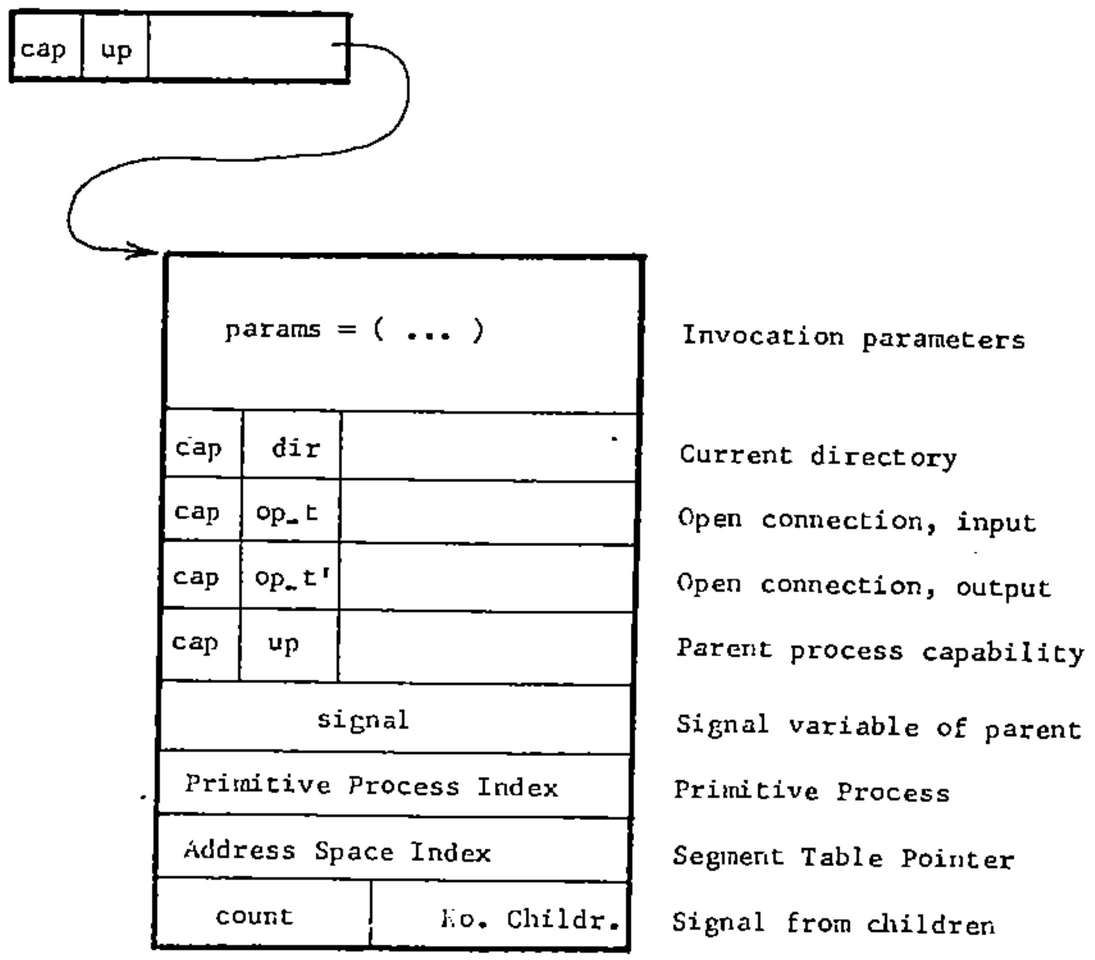

FIGURE 8.1: User Process Descriptor Block.

SOURCE is either a readable device (e.g., a keyboard) or a file. The SINK is either a writeable device (e.g., a display) or a file. The shell obtains capabilities for the components of the pipeline by a sequence of commands:

```
c1  :=  SEARCH(dir_cap, "SOURCE");
c2  :=  SEARCH(dir_cap, "F");
c3  :=  CR_CH();
c4  :=  SEARCH(dir_cap, "G");
c5  :=  CR_CH();
c6  :=  SEARCH(dir_cap, "H");
c7  :=  SEARCH(dir_cap, "SINK");
```

where c3 and c5 are capabilities for channels between program pairs (F,G) and (G,H), respectively. The shell then creates and resumes user processes that execute the three components of the pipeline, and awaits their completion:

```
RESUME(FORK(c2, -, cur_dir, OPEN(c1, read), OPEN(c3, write), signal);

RESUME(FORK(c4, -, cur_dir, OPEN(c3, read), OPEN(c5, write), signal);

RESUME(FORK(c6, -, cur_dir, OPEN(c5, read), OPEN(c7, write), signal);

JOIN(signal, 3);
```

Finally the shell can delete these processes and acknowledge completion of the entire command to the user (by a "prompt" character).

3. CONCLUSION

We have shown that the levels model can be used to describe the functions of a multi-machine operating system. Systems confirming to this model will be coherent because the physical locations of objects are hidden by the levels that manage them. They will be fault tolerant because each user machine contains a full copy of the operating system and a subset of the objects.

Two important assumptions underlie the model. The verification principle holds that reasonable steps have been taken a priori to establish that all operating system components, and all extended type packages, meet their specifications. The efficiency principle holds that all functions should be simple and compact. Verification supports efficiency by reducing the amount of runtime checking needed. Efficiency supports verification by reducing the complexity of system components.

The most significant new aspects of the model are its approaches to object management, capabilities, and directories. Each level of the system is the manager of a set of objects; it has full responsibility for mapping object identifiers to object instances, for relocating objects in the storage hierarchy of its machine, and for finding objects stored on another machine. Distributing the responsibility for object management in this way greatly improves the potential for efficiency because each level can use standard algorithms, data structures, and optimizations and does not have to use a central mapping service. Distinguishing local from nonlocal objects permits the efficient methods of shared-memory operating systems to be used when possible; the network is searched only when needed. To support remote searching and object use, the network communications level is placed below all the levels implementing sharable objects.

Capabilities are important because they obviate most runtime checks. They must be reformulated for multi-machine systems. The authorization for a process to hold a capability is checked once, when the capability is issued. The procedures of a level validate capability parameters once, on entry, by compiler-generated instructions; no further checking of capabilities is needed inside the level. Although hardware protects capabilities from alteration, it allows them to be read. This leads to a great simplification of mechanism whose reduced security is compensated by the presumed verification of programs authorized to use them.

A short form of capabilities can be used for system objects known a priori (Levels 9-13). The full generality of long capabilities is required only in the extended types level, which can be situated very close to the user interface.

The directory hierarchy, which is visible on all user machines, provides a uniform name space for all sharable system objects. The directory function can be generalized cleanly from its traditional role by storing capabilities rather than file identifiers in directory entries. No user machine has a full copy of the directory structure; it only encaches the view with which it is currently working. The full structure is maintained by a small set of "stable store" machines.

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

10. ACKNOWLEDGEMENTS

We are grateful to many persons for their advice and counsel while we formulated and refined this model. These include E. Dijkstra and N. Habermann for early inspirations about hierarchical structure; K. Levitt, M. Meliar-Smith, and P. Neumann for numerous discussions about PSOS; D. Comer for corrborating, in the XINU operating system, our intentions about placing network communication in the middle levels; and D. Denning for advice on access control, capability systems, and verification. We are also grateful to the National Science Foundation, which supported part of this work through grant MCS-8109513 at Purdue University.

11. REFERENCES

BirL82. Birrell, Andrew D., Roy Levin, Roger M. Needham, and Michael D. Schroder, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM* 25(4) (April 1982), pp. 260-274.

Brin78. Brinch Hansen, Per, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM* 21(11) (November 1978), pp. 934-941.

Brin73. Brinch Hansen, P., *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ (1973).

Come82. Comer, Douglas E., "XINU: A Real-time Operating System for Micros," Unpublished manual, Purdue University (April 1982).

Denn76. Denning, Peter J., "Fault-Tolerant Operating Systems," *Computing Surveys* 8(4) (December 1976), pp. 359-389.

DenD81. Denning, Peter J., T. Don Dennis, and Jeffrey A. Brumfield, "Low Contention Semaphores and Ready Lists," *Communications of the ACM* 24(10) (October 1981), pp. 687-699.

DenB83. Denning, Peter J. and Robert L. Brown, "Should Distributed Systems be Hidden?," CSD-TR-426, Purdue University, West Lafayette, IN (1983).

DenV66. Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM* 9(3) (March 1966), pp. 143-155.

Denn80. Dennis, T. Don, "A Capability Architecture," Ph.D. Thesis, Purdue University, West Lafayette, IN (1980).

Dijk68. Dijkstra, Edsger W., "The Structure of the THE-Multiprogramming System," *Communications of the ACM* 11(5) (May 1968), pp. 341-346.

Fabr74. Fabry, R. S., "Capability-Based Addressing," *Communications of the ACM* 17(7) (July 1974), pp. 403-412.

HabF76. 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.

Hoar78. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM* 21(8) (August 1978), pp. 666-677.

JenL76. Jones, Anita K. and Barbara H. Liskov, "A Language Extension for Controlling Access to Shared Data," *IEEE Transactions on Software Engineering* SE-2(4) (December 1976), pp. 277-285.

JonC79. Jones, Anita K., Robert J. Chansler Jr., Ivor Durham, Karsten Schwans, and Steven R. Vegdahl, "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.

JoyC82. Joy, William, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, and David Mosher, "4.2BSD System Manual," CSRG Manual, University of California, Department of EECS, Berkeley, CA (September 1982).

KahC81. Kahn, Kevin C., William M. Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, Fred J. Pollack, and Michael R. Gifkins, "iMAX: A Multiprocessor Operating System for an Object-Based Computer," *Proceedings of the Eighth Symposium on Operating Systems Principles*, (December 1981), pp. 127-136.

Morr73. Morris, James H. Jr., "Protection in Programming Languages," *Communications of the ACM* **16**(1) (January 1973), pp. 15-21.

NeeB77. Needham, R. M. and A. D. Birrell, "The CAP Filing System," *Proceedings of the Sixth Symposium on Operating System Principles*, (November 1977), pp. 1-10.

NeuB80. 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).

OusS80. Ousterhout, John K., Donald A. Scelza, and Pradeep S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *Communications of the ACM* **23**(2) (February 1980), pp. 92-105.

ParT83. Paris, Jehan-Francois and Walter F. Tichy, "STORK: An Experimental File System for Computer Networks Based on Migration," *IEEE INFOCOM*, (1983), (to appear)

PopW81. Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, (December 1981), pp. 169-177.

ReeS80. Reed, D.P. and L. Svobodova, "SWALLOW: A Distributed Data Storage System for a Local Network," pp. 355-374 in Local Networks for Computer Communication, ed. P. Janson,North-Holland, Amsterdam (August 1980).

RitT74. Ritchie, D. M. and K. L. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* **17**(7) (July 1974), pp. 365-375.

Seli80. Selinger, P. G., "Replicated Data," pp. 223-231 in Distributed Data Bases, ed. F. Poole,Cambridge University Press, Cambridge, England (1980).

SmiI82. Smith, David C., Charles Irby, Ralph Kimball, and Eric Harslem, "The Star User Interface: An Overview," *Proceedings of the AFIPS National Computer Conference*, (1982), pp. 515-528.

Tane81. Tanenbaum, Andrew S., *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ (1981).

Tich82. Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System," pp. 58-67 in Proceedings of the 6th International Conference on Software Engineering, IPS, ACM, IEEE, NBS (September 1982).

WilN79. Wilkes, M. V. and R. M. Needham, *The Cambridge CAP Computer and its Operating Sys-*

tem, Elsevier/North-Holland Publishing Co. (1979).

WulL81. Wulf, William A., Roy Levin, and Samuel P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill (1981).