Reflections on an Operating System Design[1,2]

Butler W. Lampson and Howard E. Sturgis[3]

Xerox Palo Alto Research Center

The main features of a general purpose multiaccess operating system developed for the CDC 6400 at Berkeley are presented, and its good and bad points are discussed as they appear in retrospect. Distinctive features of the design were the use of capabilities for protection, and the organization of the system into a sequence of layers, each building on the facilities provided by earlier ones and protecting itself from the malfunctions of later ones. There were serious problems in maintaining the protection between layers when levels were added to the memory hierarchy; these problems are discussed and a new solution is described.

Kev words and Phrases: operating system, protection, capabilities, layering domains, memory hierarchy, faults

CR Categories: 4.35

1. Introduction

This paper is a backward look at an operating system for the Control Data 6400 called the Cal system, which the authors and their colleagues designed and built at Berkeley between l968 and 1971. The system had a number of characteristics that were unique at the time, and are still quite unusual. Furthermore, its implementation reached a point where it was able to support a number of users doing substantial programming tasks over a number of months. As a result, a considerable amount of practical experience was gained with its novel features. Our distillation of this experience, and our subsequent ideas about how to avoid the many problems that were encountered in the Cal system, form the main body of the paper.

We begin by describing the goals of the system and the hardware environment in which it was built, together with a brief summary of its history and performance. Then we explain the basic ideas and present the main features. With this background, we point out some aspects of the design which worked out well, and then delve into some areas which gave us difficulty and where we now see room for improvement.

[1] A version of this paper was presented at the Fifth ACM Symposium on Operating Systems Principles, The University of Texas at Austin, November 19-21, 1975. A revised version appeared in *Comm. ACM* **19**, 5 (May 1976), pp 251-265. This version was created from that one by scanning and OCR; it may have errors.

[2] Copyright 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

[3] Authors' address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

1.1 Goals

We wanted to construct a general purpose operating system which would support both batch and timesharing operation. The system was to run on a commercially available machine, the Control Data 6400. It had to be reasonably competitive in performance with Scope, the manufacturer’s existing operating system, although we were willing to tolerate some loss of batch performance in return for the ability to support interactive users.

We defined three classes of applications that we wanted to support. One was simple interactive computation; editing, running small BASIC programs and the like. We did a simple-minded analysis which indicated that it was reasonable to handle 200 simultaneous users doing this kind of work on the hardware we had at our disposal. A second was the typical Fortran batch jobs which made up most of the load on the existing Scope system. We wanted to be able to simulate Scope completely, so that both the translators and utilities and the user programs could run without change. Finally, we wanted to allow large and complicated programs to be developed and run at a cost reasonably proportional to the demands they put on the hardware.

There were also some goals for the properties of the system seen by the sophisticated programmer. We wanted a protection system uniformly based on capabilities. We intended to construct the system as a sequence (actually a tree) of layers, each protected from the ones which followed it, and we wanted users to be able to add layers in the same way, and to intercept and handle exceptional conditions without incurring any overhead in the normal case. Among other *things,* this meant that users had to be able to create new types of objects.

1.2 Hardware

The system was designed for and implemented on a CDC 6400 with Extended Core Store (BCS). Our machine had 32K 60-bit words of Central Memory (CM), and 300K 60-bit words of ECS. Access to ECS is by block transfer to and from CM, with a start up time of about 3 microseconds and a transfer rate of about 10 words per microsecond. A transfer can start at any address in ECS and CM, and can be of any length. Note that this device is not at all like a drum, since the latency is negligible and there is no fixed block size.

The hardware memory protection is provided by two pairs of registers: one pair controls access to CM and the other access to ECS. One member of each pair is an address relocation register and the other an address bounds register. There is a single system call instruction, Central Exchange Jump (CEJ). The CEJ exchanges the contents of all hardware registers, including the memory protection registers, with some region of CM.

All direct access to input-output devices is provided by ten Peripheral Processing Units (PPU’s), small computers which can transfer data directly between their own memories and CM. The PPU’s in turn obtain input-output requests from agreed-upon locations in CM. The system arranges that user programs never have access to those locations. Only system code is run in the PPU’s.

1.3 History

Design of the system started in June 1968 with five participants. Coding began in December 1968, and we demonstrated two terminals editing, compiling, and running Fortran programs in July 1969, using the system kernel and an improvised file system and command processor. By October 1969 the system was being used for its own development, and design of the permanent file system and command processor began. In April 1971 these were ready. In November 1971 the project was terminated for lack of funds.

The funds ran out because, after three years of development, the system was neither efficient enough nor usable enough to be put into service by the computer center. There were three reasons for this.

First, there were a great many new and untested ideas in the system. Most of them worked out quite well, but there were still several which caused trouble, either by slowing down the development of the system or by hurting its performance. Experience with other large systems containing many new ideas, such as OS/360 or Multics, indicates that it is usually necessary to implement many parts of the system two or three times before the performance becomes acceptable. There was no time to do this with Cal.

Second, the management of the project was quite inexperienced, and as a result there were many times when substantial effort was directed into something which looked interesting, rather than into something which was really essential for the success of the system. For the same reason, there were times when implementation revealed major flaws in the design, but the decision was made, often almost by default, to go ahead anyway, rather than to redo the design. These incidents usually cost us dearly in the end.

Third, we failed to realize that a kernel is not the same thing as an operating system, and hence drastically underestimated the work required to make the system usable by ordinary programmers. The developers of Hydra [16] appear to have followed us down this garden path.

About a dozen people worked on the system during its life, and a total of about 20 man-years were invested; this includes all the support software except what was provided by CDC as part of the Scope system. Except for one part-time adviser, none of these people had ever participated in the development of an operating system before. There was no suitable high-level language available for the machine when the project was begun, and consequently all the programming was done in machine language.

At the end of its development the system could support about 15 users; it was limited by the shortage of ECS space, not by processor cycles. In the last three months of operation it was run for at least 8 hours each working day with a fairly continuous load of several users. During this time there were 18 recorded system crashes, of which 14 were due to levels of the system above the kernel. 3 were believed to be of hardware origin, and the cause of one was unknown. The 14 higher-level crashes left the kernel in good working order, but affected other parts of the system which are necessary to the well-being of users.

2. Philosophy

Our design was guided by a number of principles, acting either as a framework, or to direct choices between competing designs. In retrospect, some of these principles appear to be unsound, especially if applied too rigidly. Later sections of the paper comment on some of the problems that arose from such rigid application.

The first three principles are crucial to the structure of the system. They involve several interrelated concepts: domain, object, capability and operation. The three principles should be read as a whole, since each uses terms defined in the others.

Protection is based on domains. All code outside the system kernel runs within some protection domain (it is incorrect, but often convenient, to say that the domain itself is running). The only resources inside the domain are a set of machine registers, a portion of CM, and a local capability list *(c-list).* Resources outside the domain can be accessed only through *capabilities* stored in the local c-list. A program running within a domain *D* has only one way to interact with anything outside *D:* by invoking an operation on some objects found in *D's* local c-list. Figure 1 illustrates two domains.

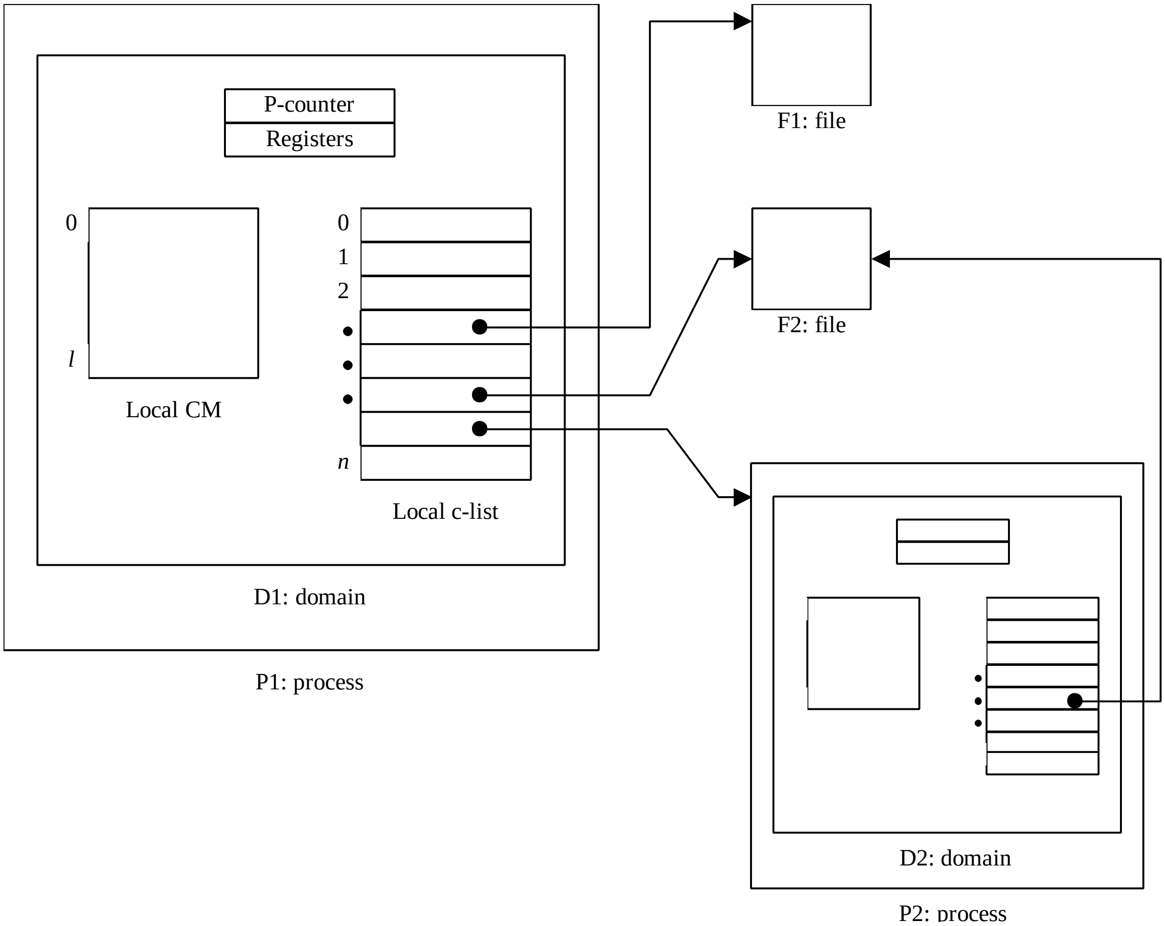


Fig. 1. Domains, capabilities, and objects

The purpose of a domain is to *provide a protection context.* This implies (among other things) that it must be possible to completely isolate a domain from undesired external influences, or in other words, to give it exclusive control over every aspect of its environment on which its correct functioning depends. Not every domain, of course, will actually have such exclusive control: in many cases one domain will be cooperating with, or subordinate to, another one. Nonetheless, the possibility of complete isolation is a crucial property of the protection system.

The system provides a *virtual world composed of objects.* Different *types* of objects are used to embody different kinds of facilities, i.e. c-lists to contain capabilities, files to contain data, processes to perform sequential computations, allocation blocks to control the use of basic resources such as ECS space and CPU time, and so forth. An object can only be manipulated by invoking an *operation,* which is itself an object. For each type of object, there are operations to create and destroy objects of that type, and to read and modify the state of such objects.

The system changes state only as a result of the activity of a *process.* For example, a process may execute a machine instruction that changes the state of the machine registers of the domain in which it is running, or it may perform an operation on objects in the domain’s local c-list. An autonomous input-output device is represented in this scheme by one or more pseudo-processes, which contain the state of the device. These pseudo-processes then communicate with other processes through event channels and files just like ordinary processes.

In the Cal system, as in most capability-based systems, objects are intrinsically shareable: any domain can have access to any object if a capability for the object appears in its local c-list. It is of course possible for this capability to appear in only one local c-list, but the structure of the system does not enforce or even encourage such exclusive access. The only things to which a domain automatically has exclusive access are its CM and its registers. By contrast, in a virtual machine system such as VM/370 [11], a virtual disk belongs to a particular machine, and is normally inaccessible to all other machines.

Objects are named by capabilities. A capability is an unforgeable name for (or pointer to) an object. The system kernel guarantees the integrity of the capability by storing it only in a c-list, to which non-kernel programs never have direct access. The contents of a c-list can only be altered by operations that the kernel provides, and these preserve the integrity of the capabilities stored in the c-list.

A program outside the kernel can name an object *only* by giving an index in the local c-list of the domain in which the program is running. For example, in Figure 1, domain *D1* can name the file *F1* by the integer 2. The index specifies a capability, which in turn points to the object. In this paper we usually make no distinction between an object, a capability for the object, and a c-list index which specifies the capability; it should be obvious from the context which one is meant. When we speak of an operation *returning* a capability, we mean that the operation takes two parameters: a c-list, and an integer that specifies a position in the c-list where the capability is to be stored.

Objects are defined abstractly. Each object can be accessed only through a small set of operations. Any design proposal is expected to consist of the following three parts:

- an abstractly specified set of states for an object;
- a set of primitive operations on the object and their effects on the states;
- a representation for the states.

Thus, an object is very much like a protected version of a Simula class [3].

Tile system is built in protected layers. The first layers are simple: The correctness of a layer must not depend on the correct programming of any later layer; in other words, the layers are protected. Further, it must be possible to change the implementation of an earlier layer without reprogramming any later layer. The system can always be extended by adding another layer. Each new layer defines a new virtual world.

Extensions can be transparent. The most frequently used objects and operations can be represented directly by the objects and operations of earlier layers. Only if the earlier layers cannot perform the operation should the later one become involved. This is a rather strong requirement that had a great deal of impact on the system design.

Any use of a resource should be chargeable to some specific user. There should be no anonymous use of machine resources. Since a user is to be charged for his use of resources, this use should be the same if the same program is rerun under different load conditions.

3. The Cal System Kernel

The actual Cal system is constructed in four layers, not counting domains that implement the Scope simulator and other specialized services. The description in this paper collapses the last three layers into one for simplicity, leaving two layers that we will call the *kernel* system (described in this section) and the *user* system (described in the next section).

The kernel defines the first protected layer. it provides what we thought were a minimal set of facilities for the efficient construction of further protected layers. Everything else needed by real user programs is provided by the user system. In particular, the user system extends the memory hierarchy to include the disk, and it provides symbolic names for objects.

Some knowledge of other systems that implement a memory hierarchy suggested that the system code for moving representations of objects to and from the disk would be quite complicated. It must not only deal with the technical aspects of efficient disk input-output, but also solve the strategic problem of choosing which representations to move. In view of this, we decided that the kernel would deal only with objects represented in ECS. All kernel data is stored in ECS or CM. The disk is simply an input/output device to the kernel system. One attractive consequence of this decision is that the kernel never has to wait for an input/output operation to complete. Some of the other, less attractive consequences are explored in Section 8.

3.1 Outline

The kernel system implements the kernel world, which consists of the following eight types of objects, and about 100 operations which can be performed on them:

kernel files
event channels
allocation blocks
c-lists
labels
operations
processes
types

Domains are not full-fledged kernel objects, but lead a second-class existence within processes.

In this section we give a brief description of these objects and the operations on them. All objects have create and destroy operations, which are not mentioned in the individual descriptions. The objects and operations are summarized in Table I. For more detailed information, the reader may consult [14]. He should note that in this paper we are using somewhat different names than are used there.

3.2 Kernel Files

Files provide the primary facility for storing data. A file is a sequence of 60-bit words, divided into equal size pages, each of which can be present or absent. Figure 2 illustrates this structure. It was designed to make it convenient for a kernel file to represent a user file that might be mostly on the disk. The way this is done is discussed in detail in Section 4.2.

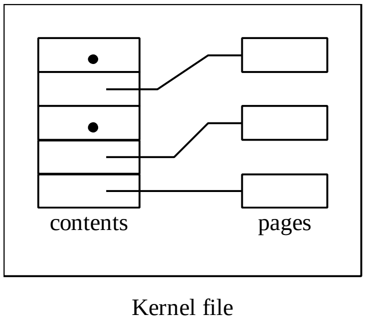


Fig. 2. A kernel file with five pages, two of which are absent

Operations are available to create and destroy individual pages, and to transfer sequences of words between consecutive file addresses and consecutive CM addresses within a domain. Finally, there is a swap operation for exchanging two pages between two files. The swap action is provided so that kernel files can be transparently extended to user files: see Section 4.2.

3.3 Event Channels

Event channels are used for interprocess signaling, and to transfer single word messages called *events.* Two operations are available: send an event, and get an event. A channel contains storage for a fixed number of such events, and the send operation returns a failure indication if the channel is full.

There are four variations of the ‘get an event” operation:

- get an event from a single event channel; if no event is waiting,

    return and so indicate,

    wait until one is available.

- -get an event from one of several event channels; if no event is waiting on any of them,

    return and so indicate,

    wait until one is available on one of the channels.

The only way for a process to suspend execution voluntarily is to wait for an event from an event channel.

3.4 Allocation Blocks

Allocation blocks provide the authority to use kernel system resources, such as ECS space and CPU time. Creation of an object requires an allocation block with sufficient ECS space to store the object’s representation. When the object is created, the space is removed from the allocation block.

An allocation block contains a list of all the objects created on its authority and hence *dependent* on it. There is an operation which, given an allocation block, returns a capability for the nth object dependent on it. This allows the bolder of a capability for an allocation block to systematically delete the dependent objects and recover the space they occupy. The dependency relation defines a tree structure on allocation blocks with a unique root. This root is created at system initialization time with ownership of all the system resources.

Object	Components	Operations
File	n: integer data: array [n] of empty *or* page: array of word	read/write (address in CM, address in file, number of words: integer) create/destroy page (address in file: integer) swap page (F, G: **file**, address in F, address in G: integer)
Event channel	n: integer events: array [n] of event: word waiting: queue of **process**	send event (event: word) get event (if empty: {wait, return}) get event from several channels (**if** empty: {wait, return}, list of event channels: **c-list**)
Allocation block	ECS space: integer CPU time: integer dependents: queue of objects	get capability (object number: integer) transfer funds (source, dest: **allocation block**, space, time: integer) (all create operations also take an allocation block parameter)
c-list	n: integer contents: array [n] of empty *or* capability = type: integer rights: set of rights value: word	move capability (source, dest: **c-list**, source index, dest index: integer, rights mask: set of rights) read contents (index: integer)
Type	type number: integer	create capability (value of new capability: word)
Label	value: integer	none
Process	call stack: array of domain: **label** PC: integer registers: array [16] of word account: **allocation block** timers: array of integer set of domain = name: **label** father: **label** local c-list: **c-list** map: array of map entry = file: **file** address in file: integer address in CM: integer number of words: integer set of error codes	create domain (name, father: **label**, size of local c-list, size of map: integer) send interrupt (target domain: **label**, value: word) return (type: {normal, abnormal}, value: word) return with error (error number: integer) jump return (number of levels: integer)
Operation	nlevels: integer levels: array [nlevels] of action = domain: **label** *or* kernel action: integer nparams: integer array [nparams] of parameter spec = variable data: empty *or* variable cap = type number: integer rights: set of rights *or* fixed data: word *or* fixed cap: **capability**	copy operation add level (action: **label**, number of parameters: integer, array of parameter specifications) tighten specification (parameter number: integer, required rights: set of rights) fix data part (parameter number: integer, fixed data: word) fix capability part (parameter number: integer) fixed capability: **capability**)
Notes	The notation is borrowed from Pascal. Boldface words stand for objects of the indicated type. All the operations take an object of the type being described as a parameter, in addition to the parameters that are shown explicitly. All objects have create and destroy operations in addition to those listed.	

Table I. Cal System Kernel Objects and Operations.

3.5 C-lists and Capabilities

A c-list is a finite sequence of capabilities. A capability is a system maintained, unforgeable, authorization [5]. Many capabilities contain pointers to the representations of system maintained objects, such as files and event channels. A capability also contains a list of the things which can be done on its authority; these are called *rights*, following Hydra [15].

A capability has three components:
 a type *t;*
 a set of rights *r;*
 a value *v*.
The type is an integer that identifies the type of object named by the capability. The kernel system has the eight built-in types listed above, and it also allows new types to be created. The set of rights *r* in a capability is a subset of the set *R* of all possible rights. When a capability is supplied as a parameter to an operation, the operation can require certain rights to be present in *r*: see Section 3.8. Typical rights for a file capability might be *read* and *write*.

The value is simply an integer. Its interpretation depends on the type *t*. The implementation of type *t* can interpret this integer in any way it pleases. Typically the kernel interprets the value of a capability for a kernel object as a pair (unique name, index). In this pair the unique name is an integer which uniquely identifies the object in the set of all objects ever existing in the system. The index points to an entry in a *master object table* or MOT, which in turn points to the object. An MOT entry also contains the unique name of the object, and this unique name is compared against the unique name in the capability whenever the path from capability to object is followed. Figure 3 illustrates the scheme.

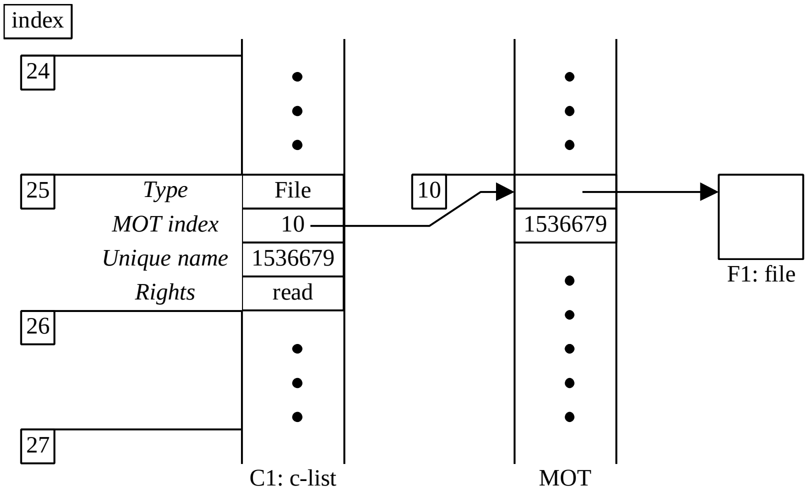


Fig. 3. Capability 25 in c-list *C1* points to file *F1* through MOT entry 10.

This arrangement has two advantages. First, it is trivial to move objects around in ECS, since there is only one place in the system where the ECS address of an object can appear, namely the MOT entry. Second, objects can be deleted without worrying about whether any capabilities for them remain extant, since any later attempt to use such a capability will fail the unique name check.

There is an operation for moving a capability from one c-list to another. During this move it is possible to remove some rights from the capability's set of rights. It is also possible to read the contents *(t, r, v*) of a capability as a set of bits.

3.6 Labels

A label is a global name for equivalent domains in different processes. The implementation is simply an integer. A capability for a label contains the integer in its value part. The reason for using labels to name domains is discussed in Section 3.8.

3.7 Processes and Domains

Processes are quite complicated, and contain a number of components most of which are not accessible as independent objects. The components are a tree of domains, a call stack, a set of event channel chain pointers, an allocation block to pay for processor cycles, a set of machine registers, and some timers.

A domain consists of a local c-list, a map, a name that is a label object, a father domain, and a list of the errors the domain is willing to handle. The map specifies what portions of CM are to contain what portions of files. When a domain is run, the map is consulted and the mapped portions of files are copied into the appropriate portion of CM. From time to time, CM is copied back to the appropriate files. This scheme makes it unnecessary to have a special object to hold the contents of a domain's CM, since ordinary files serve this purpose. We also thought that it would provide a convenient mechanism for sharing data between domains, as it does in systems with hardware implemented mapping such as Tenex (see [l]).

Unfortunately, the map mechanism turned out to cause many problems: some of them are later discussed in Section 6.1.

The domain tree is used to maintain a control relationship among domains. Each domain has exactly one father, which is his direct controller. There is one root of the tree. This tree is used to decide which domain should be called when an error occurs (see below), according to the following rule: choose the nearest ancestor of the domain signaling the error which has indicated a willingness to handle the error.

Whenever a domain calls on an ancestor, we define a *full path,* which consists of the called domain, the caller, and all domains in between in the tree. The CM and local c-list of the called domain are extended by concatenating the CM's and local c-lists of all the domains on the full path. The reason for this is to give a controlling domain direct access to the memory of a controlled domain. The full path turned out to be a bad idea.

Finally, there is an interrupt facility, which permits a program in one process to get the attention of a particular domain in another. Its parameter is a label that names the domain whose attention is desired. The effect is that the named domain, or an ancestor, will be called as soon as the named domain or a descendant is running. Thus the domain tree is used to ensure that a domain cannot be interrupted by an inferior domain.

There are operations to send an interrupt, to signal an error, and to create and destroy a domain. Domains are normally rather static objects, which are created and destroyed much less often than they are called. In this respect they are similar to Fortran subroutines. By contrast, the parallel type of object in Hydra [l5], the *local name space* or LNS, is normally created to handle a single call, like an instance of an Algol procedure. Because a Hydra LNS is short-lived and frequently recreated, the prescription for creating it is an important part of the system; it is called a *procedure,* and is exactly analogous to an Algol procedure (as contrasted to an *instance* of a procedure). In Cal the prescription for creating a domain, called a *domain descriptor,* is not even a kernel construct, but is provided by the user system (see Section 4.6).

The reader should be warned that this neat parallel between Cal and Hydra is confused by the fact that Hydra procedures also perform the function of Cal operations (described in the next section); i.e. they authorize transferring control to a domain as well as specifying how to create it.

3.8 Operations

Operations contain the authority to invoke a computation in some protection context other than the current domain: either in the kernel, or in some other domain. An operation is made up of *levels,* each of which contains two parts:

> the action to be performed, i.e. the new protection context in which the computation is to proceed, and the entry point in that context;
>
> a list of *parameter specifications* for the parameters which should be passed to that action.

The action can either be a kernel action or an entry point to a domain. In the latter case, the domain is named by a label object, and such an object must be presented to construct an operation which calls on a domain, as well as to create the domain. This global naming scheme allows operations to be shared between processes. Consider two domains in different processes, each implementing the same set of operations on the same type of object. We want the same operations to be usable for calling both domains, and we therefore need a single name that refers to the proper domain in each process. A label performs precisely this function. An alternative solution to this problem is the one adopted by Hydra, which creates the domain only when the operation is invoked.

In order to invoke an operation, the program in a domain calls the system with a CEJ, specifying the operation and its capability parameters by indices into the domain’s local c-list, and its data parameters by integers. Figure 4 shows a simple example. If the action of the operation is a domain, the kernel system searches the set of domains in the process for one whose name is equal to the label in the operation. Parameters are picked up from the c-list and CM of the calling domain according to the parameter specifications of the operation, and placed in the c-list and CM of the called domain. The call stack in the process is used to store the return location for the call.

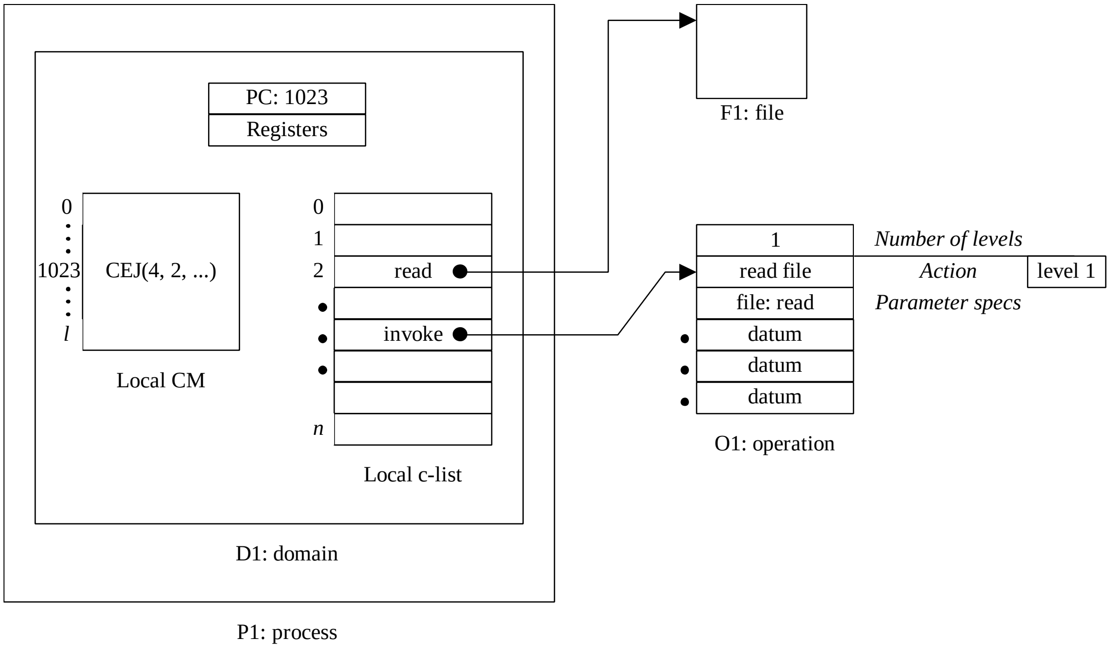


Fig. 4. Domain *D1* is about to read from file *F1*

Each parameter specification determines how a single parameter is to be obtained:
as data in the caller’s CM;
as a capability in the caller’s local c-list;
as fixed data stored in the operation;
as a fixed capability stored in the operation.

A specification for a capability can also specify the type and rights which the actual parameter must have. An operation with fixed parameters appears to have fewer parameters to the caller than the called domain. This feature was designed to allow a general operation to be specialized in a protected way. It can also be used, however, to seal a capability [12], so that it is accessible only to the domain which gets invoked by the operation, and not to other domains which merely have a capability for the operation.

This can be done by simply embedding the capability in the operation as a fixed parameter. It will then be passed to the domain that is called when the operation is invoked, but there will be no other way to extract it from the operation. In effect, the label which is the action of the operation is the seal. The Plessey 250 system [4] uses a similar mechanism, but with the benefit of hardware support. Unfortunately we did not realize while we were building the Cal system that our operations had this much power.

There are several kinds of return from a domain call, each invoked by an operation:
normal;

abnormal;
return and signal an error from the caller;
return to a specified domain more than one level back on the stack.

When an operation is invoked, the parameters are collected and action taken according to the first level. When the action returns, it can do so either normally, in which case control reverts to the caller, or abnormally. In the latter case, if it is the *i*-th level of the operation that is returning abnormally, the (*i*+1)-th level is invoked if it exists. Otherwise control reverts to the called with a special indication that the return is abnormal. This mechanism is provided to facilitate transparent extension; a level can be added to an operation to handle a fault without adding any overhead if the fault does not occur.

There are operations to copy an operation, to tighten the specifications of a parameter, to supply a fixed value for a parameter, and to add a level to an existing operation. New levels or new operations can only contain actions to call domains, never kernel actions.

3.9 Types

Type objects authorize the creation of capabilities for non-kernel objects. One operation is available on a type object; create a capability of that type. It requires two parameters:
a type T
a single word of data
The result is a capability of type *T,* with all rights present and with the data word as its value. If such a capability is handed to a random program, that program can never change the type or the data part. Hence if the capability is later presented, as authority for an intended action, to a domain which implements the type *T*, the domain can read the type and the value in order to determine whether to accept the capability. There is no way such a capability can be forged, since it cannot be created without access to the type *T,* and it cannot be modified once created.

Thus, a type may be thought of as a seal [12] that can be used to seal a single word of data. This sealing mechanism is not as flexible as the one that uses operations (Section 3.8), but it is much cheaper. It also had the advantage that we understood it while we were building the system.

3.10 Input-Output

Each input-output device communicates with users of the kernel system using one or more files and event channels. The device accepts and returns control information, and possibly small amounts of data, on the event channels. Large amounts of data are deposited in or taken from the files. The device itself may be thought of as a process, and in fact this design ensures that a device can always be simulated by an ordinary process, which is nice for debugging and for compatibility in the face of changes in hardware.

The primary design criterion for the interface presented by a device is that the properties of the hardware should be preserved as fully as possible. For example, a magnetic tape appears as a sequence of records of variable length, rather than with some more elaborate structure of labels and files which would prevent an arbitrary tape from being read or written. The primary job of the kernel system code that handles a device is to reconcile the timing characteristics of the device and the response times of the user programs that will deal with it. Thus in the case of the tape, a number of data transfers can be queued, and the responses are similarly queued, so that the tape can be run at full speed even though the user program runs relatively infrequently.

4. The Cal User System

The user system provides a richer collection of objects, and the ability to store object representations on the disk. Types of objects in the user world include:

all of the kernel types
disk files
access keys
directories
name tags
domain descriptors

All user system objects which are not kernel objects are also called *disk objects,* because the user system allows them to be stored on the disk.

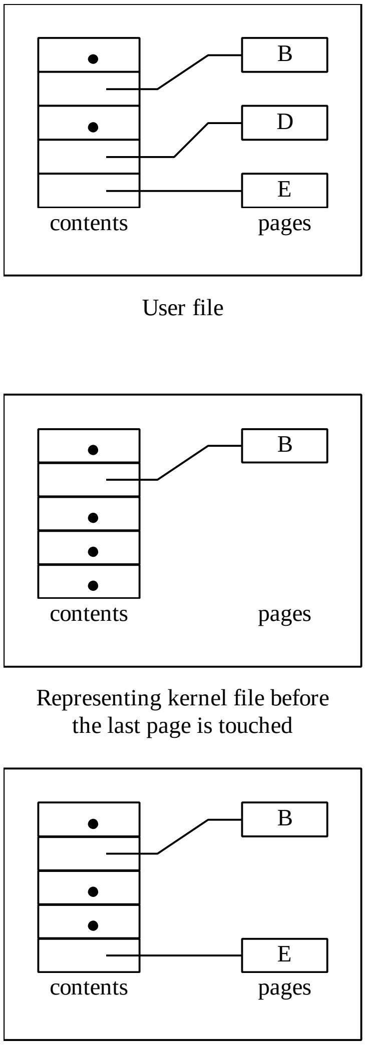
When the user system is initialized, it obtains a type capability for each new type of object it supports. These capabilities are then used to construct capabilities for such objects as they are needed, as described in Section 3.9.

4.1 Kernel Objects

Any type of kernel object is available to a user with two important restrictions: no kernel object can be represented on the disk, and no capability for a kernel object can appear in a directory. The reason for these restrictions is that the kernel views the disk simply as an input-output device. Hence it is willing to transmit bits to and from the disk. It is not, however willing to do the same for objects, since it can have no control over the integrity of the disk representation of an object.

4.2 Disk Files

A disk file F_d has the same structure as a kernel file. However, a portion of a disk file can reside on the disk and a portion in ECS**.** The portion in ECS is represented by a kernel file F_k, as illustrated in Figure 5. An *attach* operation is available to force a portion of a disk file to reside in ECS.



User file

Representing kernel file before the last page is touched

Representing kernel file after the last page is touched

Fig. 5. A user file represented by a kernel file

A capability for the representing kernel file F_k is obtained by applying an *open* operation (provided by the user system) to F_d. The user system determines whether F_k already exists, and if not it constructs an empty kernel file to serve as F_k. A capability for F_k is then returned.

The disk file read and write operations are multilevel operations which take an F_k as a parameter; the first level of these operations contains a kernel action. If the portion of F_d referenced by the read or write is represented in F_k, the operation succeeds immediately without invoking the user system. If not, an abnormal return occurs which invokes the second level of the operation, a call on the user system. The user system now performs the appropriate disk input-output.

A portion of a disk file can also appear in one or more domain maps (Section 3.7). When this happens, the user system forces into ECS any file pages containing data which is referenced by a map.

Since a page is the smallest unit of a kernel file which can be created or destroyed, and since the fact that data in a user file is missing from the representing kernel file can only be detected by the fault which occurs when a nonexistent kernel file page is referenced, adding or removing a block from a kernel file must be an atomic operation. The swap operation mention in Section 3.2 is provided for this purpose.

4.3 Access Keys

An access key is an object containing a single integer that cannot be modified. The integer is stored in the value part of each capability for the access key. An access key is used like an ordinary key, which can fit certain locks. Associated with an entry in a directory is a list of access key values that act as locks. In order to access a given entry in a directory, a domain must present an access key which fits one of the locks on the entry, i.e. which has a value equal to one of the lock values [8].

4.4 Directories

A directory consists of a list of entries, each containing a symbolic name, an object specification and a list of access locks.

The symbolic name is a string of characters. The object specification can be one of three things:

- an owned entry, containing a pointer to u disk object;
- a hard link, containing a pointer to a disk object not owned by this entry;
- a soft link, containing
  a pointer to another directory,
  a symbolic name to look up in that directory,
  an access key to use as authority.

An access lock is a pair:

- a number to be matched against the value of an access key;
- a set of rights.

Each user object that occupies space on the disk, except for a single root directory, has exactly one ownership entry in some directory; this entry is constructed when the object is created. Thus, the directories induce a tree structure on the user objects, which is used for accounting purposes, much like the allocation blocks of the kernel system (see Section 3.4).

A directory is normally accessed with an operation that requires three arguments: the directory, a symbolic name, and an access key. It succeeds if an entry is found in the directory with the specified symbolic name, and the access lock list for that entry contains an access lock equal to the value of the access key. In this case, a capability is returned for the object specified in the entry, and with the rights associated with the access lock. The user system constructs this capability by using one of its type capabilities. If the object specification in the entry is a soft link, the action is repeated on the directory given in the link. Otherwise, the operation returns abnormally.

4.5 Name Tags

Since kernel capabilities for kernel objects cannot appear in directories, we invented name tags to stand for them. Like an access key, a name tag is an object containing a single unchangeable integer, and the integer is stored in the value part of each capability for the name tag. The user system maintains a table of correspondences between these integers and other capabilities, in particular, capabilities for kernel objects. This table is destroyed each time the system is taken down, or crashes. The system was taken down at least once a day, so the correspondence was quite ephemeral.

An operation is available to set the correspondence. Moreover, the user system initializes the table to contain various special objects, such as the files and event channels used to communicate with input-output devices.

4.6 Domain Descriptors

A domain descriptor contains a prescription for building a user domain: the label value which names the domain, a specification of the map to be used, and capabilities to be placed in the domain's local c-list when it is initialized. The only operation on a domain descriptor is to invoke it, and the effect is to create the described domain. In particular, the capabilities specified for the initial local c-list cannot be extracted from the descriptor. Thus, a calling domain can request the construction of a domain that contains capabilities for objects that the called domain cannot itself access. Another way of saying this is that a domain descriptor is a sealed object.

Within the kernel system a domain descriptor could be implemented by an operation which contains the label and the initial capabilities as fixed parameters (see Section 3.8*),* and has an action which invokes a system domain to construct the new domain and pass it the capabilities. It would still be difficult to store such an operation on the disk, but we never looked closely at this problem because we did not realize that our operations could be used in this way. The section on layering below treats this issue in detail. In fact, the user system represents a domain descriptor as a special kind of directory.

5. Various Successes

The preceding sections have described the Cal system more or less as it actually existed. The remainder of the paper is devoted to a discussion of things that probably should have been done differently. Before plunging into this, we will pause to survey things that worked out well.

In spite of the problems described in Section 8 below, the layering structure worked out quite well. Among other things, it led to a highly reliable kernel system; during several months of operation there were at most four crashes that could possibly have been due to kernel failures.

Capabilities were also very successful. They gave us a consistent and uniform way of naming objects and controlling access to them, and presented no problems except for the difficulty of representing them on the disk, which is discussed in detail in Section 8.

The way in which the kernel handled input-output was good. We were able to obtain performance at least equal to, and often better than, that provided by CDC's Scope system. The code in the PPU s was not too complicated, and debugging of the kernel code was quite easy. Although we had little opportunity to take advantage of the ability to simulate devices with user processes, that also worked well when it was used.

We were able to implement a complete simulator for the Scope system with about three man-months of effort, and to run a large variety of programs written for that system without any changes. This simulator was written entirely as a user program, running first on top of the kernel system, and later on top of the user system. It required no additions to these systems, except for a mechanism that fielded the rather peculiar supervisor call used by Scope.

The extensibility of the kernel was put to a rather severe test by the construction of the user system, since we did very little design of the user system before specifying and implementing the kernel. In terms of the functions provided, the kernel met this test quite well; only a few rather minor changes were made to accommodate the special needs of the user system. There were some serious problems with maps, discussed in the next Section, and with moving capabilities out to the disk, discussed in Section 8. Furthermore, the performance was not satisfactory, because the overhead incurred in a few common kernel operations was too high.

On the whole, however, we were able to extend the kernel with three additional layers rather successfully. These three layers were parts of the user system; the details of the division were suppressed in the description above. The Scope simulator was a fourth layer, but it did not make use of any of the system's extensibility features, since the programs run under it of course did not attempt to use directly any of the features of our system.

6. Various Problems

This section describes four areas in which we had fairly serious problems, but which do not seem to raise any major conceptual issues.

6.1 Maps

Our attempt to simulate mapping hardware by copying blocks of information between CM and ECS (Section 3.7) worked out badly. First, we were unable to do the simulation precisely enough, and second, the maps interacted with the extension of kernel files into disk files in unanticipated and unfortunate ways.

The simulation of mapping hardware breaks down when the same data word appears in more than one place in the physical CM; the reason should be obvious. Unfortunately, there is no way to prevent this from happening, by programming convention, with any system design which allows more than one domain’s CM to be in the physical CM at a time. A system with more than one CPU will have the same problem.

As a result, the map is actually used only to avoid multiple copies of shared code. For this purpose it is entirely satisfactory; in fact, it will work well for any shared information which is never modified. A much simpler mapping scheme than the one we implemented would be quite adequate for this application.

The second problem arises from the fact that the kernel requires any portion of a file that appears in a map to actually exist. It is impossible to wait until a program actually refers to the file and then generate a fault, as most systems which have paging hardware do, because the hardware provides no way to detect a memory reference to a portion of CM that happens to correspond to a missing portion of a file.

Because of this requirement, it is necessary for the user system to ensure that every file page containing any data that appears in a map is actually in ECS. This has two unfortunate consequences. First, all changes to maps that involve user system (disk) files must be interpreted by the user system, so that it can keep track of which pages must be kept in ECS. The kernel system enforces its requirement by preventing any page in a map from being deleted, but this does not help, since the user system has no way of knowing when a page is finally removed from all maps without keeping track of all changes. The double bookkeeping that results is very annoying. The second, equally annoying, consequence is that the only way to swap out data that appears in a domain’s map is to destroy the domain.

Both of these problems could be avoided if the kernel complained about missing file pages only when it actually tried to bring the missing data into CM. Then the user system could use its best judgment about what to keep in ECS, and any error in judgment would result in a fault when the affected domain started to run.

This fault could be dealt with in the usual way, by bringing the missing data back from the disk. To make this work, each domain would have to have associated with it another domain (presumably belonging to the user system) which the kernel could call to report the fault.

A third bad consequence of maps as we defined them is that they make it difficult to access a domain’s CM with simple reads and writes of files, since it is quite difficult in general to figure out which file blocks correspond to a given section of the CM. This difficulty motivated us to define the full path (Section 3.7) arid make the convention that all the CM of all the domains in the full path is brought into physical CM simultaneously, so that it can be easily addressed with the machine’s load and store instructions. The effect of this scheme is that the maximum size of a domain’s CM is limited to the available physical CM not occupied by resident system, less the sum of the CM’s of all its ancestors. Since our physical CM was only 32K, this was very serious, and resulted in many circumlocutions in the implementation of the user system for the sole purpose of minimizing the size of domains in the full path.

6.2 Cost of Kernel Operations

The minimum cost of invoking an operation is about 250 µs; this time goes into

- saving and restoring state information;
- following the capability for the operation through the MOT to the actual representation of the operation in ECS, and interpreting the operation;
- passing one or two parameters.

Operations to send and receive events (mostly for mutual exclusion), and to read and write small blocks in files, are very frequent in the user system, and in fact take about half the total execution time for a program which simply reads a disk file at full speed. Changing to another domain is even more expensive, but is done less frequently.

This problem is to some extent inherent in a system that implements domain changing entirely in software. In fact, the Cal system can make such changes considerably faster than other software-only systems, such as the original Multics system or Hydra. The only real solution is hardware support, as in the current Multics [l3] or the Plessey 250 [4]. We believe that treating the most common kernel operations as special eases could have reduced the cost of those operations by about a factor of 2.

6.3 Ceilings

We used allocation blocks to control the use of our most obvious scarce resources: ECS space and CPU time. Unfortunately, our design had several other resources that were available in limited quantities, and each of these constituted a ceiling against which the system as a whole or some user program might bump. Each therefore required some amount of special handling, and each thus became a source of small design problems and obscure bugs.

The system allocates a fixed amount of space to the Master Object Table (MOT), in which almost every kernel object has to have an entry (see Section 3.5). It is therefore possible to run out of MOT entries without running out of ECS space. The reason for the fixed size is that capabilities contain indexes into the MOT, which would become invalid if MOT entries were moved. This means that we cannot shrink the size of the MOT, so we cannot simply charge the space occupied by an object's MOT entry against its allocation block, as we do with the other ECS space it uses.

This problem could be avoided by allowing MOT entries to move, and searching the MOT with the unique name as a key when the unique name comparison fails. If the search turns up the entry in another location, the capability can be fixed up so that the cost of the search is incurred at most once for each copy of the capability. Alternatively, all the capabilities in the system could be fixed up whenever the MOT is compacted; we had a religious bias against using such a non-incremental garbage collection technique.

Two other ceilings are the total number of types in the system, limited to about 20,000 by an accident of the implementation, and the number of message slots in an event channel.

6.4 Accountability

The goal of accounting for all costs and charging them to a specific user was a mistake. We did not really attain this, but whenever there was a choice, we favored a decision that would make some cost more accountable. The parallel goal, that a user’s costs should be predictable, led us to conclude that any system activity charged to a user must be under the direct control of his program. In other words, the system should not cause implied activity which may or may not occur. Latent in these two goals is a bias against sharing of physical resources.

For example, we wanted to charge all CPU time to users. If this is to be feasible, a user request should directly give rise to a period of system computation, all of which is necessary to satisfy the request. This concept prejudiced us against automatic memory management systems, since in such systems there is a lot of activity that is difficult to identify with particular requests.

The desire to identify the user originating a system computation also influenced our process design. Since a substantial portion of the system is implemented outside the kernel, we wanted to associate the execution of such code with particular users. Hence a process consists of several domains. Certain of these domains contain system programs, whose execution is automatically associated with the process in which they are embedded. An alternative approach, which uses a single process to manage each global database, would avoid many deadlock problems and greatly reduce the cost of mutual exclusion.

6.5 Memory Hierarchy

These same accountability questions arose with respect to ECS space. In a system in which representations move between ECS and the disk under control of the system, it is difficult to identify a particular user as responsible for the presence of an object in ECS. Hence we designed the system so that the user could control the movement of representations, and so be equitably charged when they were in ECS.

In particular, we transferred data from disk to ECS on demand, but wrote it back or discarded it immediately thereafter, unless there was an explicit request to leave it around. As a result, disk files that were frequently referenced by independent programs, such as directories, accounting files, and commonly used subsystems, were transferred to and from the disk over and over again. Our reluctance to manage the contents of ECS automatically was motivated by the accountability considerations just discussed, but we failed to appreciate how great the cost in performance would be.

7. Domains and Messages

The Cal system contains two separate, though related, mechanisms which provide isolation between computation: processes, which permit a number of independent sequential computations to exist, and domains, which permit a number of independent protection contexts to exist. So that the isolation can be breached in a controlled way, each mechanism includes facilities for communication: event channels for processes, and operations for domains. There are many parallels between the two mechanisms, and especially between the communication facilities.

Considerations of efficiency led us in Cal to make these mechanisms separate, and to embed domains within processes. It now seems to us, however, that this may have been a mistake, and that it might be better to identify the notions of domain and process, and to use messages uniformly for all communication mediated by the system. Space unfortunately prevents us from expanding on these ideas here, but we hope to do so in a subsequent paper.

8. Layering

One of the major goals of the Cal system design was to construct a layered system The code for the system is divided into two large sections. One, the kernel, forms a reasonably complete system in itself. The other, the user system, runs as user code relative to the kernel, and in turn implements the system seen by actual users. In this way, the kernel can operate correctly even if the user system is incorrectly programmed.

The actual design we chose was motivated by a consideration of the memory hierarchy. Since ECS is too small to contain representations for all existing user objects, some objects will be represented in ECS and some on the disk. By analogy to paging systems, objects being referenced by a running process should be represented in ECS. Thus, representations will move between the disk and ECS.

The total system must do two things:

- represent objects and provide operations on them,
- move object representations between the disk and ECS.

There are two conceivable ways to divide these responsibilities between the two layers:

- The kernel system provides a large virtual memory, swapping pages between the disk and ECS. The user system then represents objects in this large virtual memory.
- The kernel system provides a representation for objects in ECS only. The user system moves object representations between ECS and the disk, and uses the kernel system to represent objects in ECS.

We chose the second alternative.

In order to form a system based on this alternative, the two layers must have certain properties. These may be summarized as follows:

(1) The kernel is a system in its own right.

(a) It defends itself against an incorrectly programmed user system.

(b) It provides object representations and operations in ECS and CM.

(c) It provides access to the world outside ECS and CM, e.g. the disk, as ordinary data input-output. The kernel makes no attempt to move object representations to and from the disk.

(2) The user system is implemented as a layer on top of the kernel.

(a) The user system code runs in domains of kernel processes.

(b) While in ECS, user objects are represented by kernel objects. For efficiency, frequent actions on user objects are implemented as kernel actions on the representing kernel objects.

(c) To move an object from ECS to the disk, the user system uses kernel operations to read the state of the kernel object, and constructs a description of the state of the represented user object. This description is simply a sequence of bits. Using appropriate kernel operations, the user system writes this description on the disk. Moving objects from the disk to ECS is done by reversing this process.

8.1 A Difficulty

As we have seen earlier (Section 4.2) we were successful in providing these properties for disk files. Unfortunately, we were unsuccessful for other objects. We did succeed in constructing a kernel satisfying (1), and a user system satisfying (2a) and (2b). However, we could not satisfy (2c). In particular, we were unable to represent c-lists on the disk. The only objects for which we could give a disk representation were those which had no direct kernel representation (e.g. directories) and files.

As an illustration of the difficulty we encountered consider c-lists. Assume a user system has been constructed which allows user c-lists to move to and from the disk. Picture the user system about to move the representation of a user c-list from the disk to ECS as in Figure 6. The representation in ECS is to be by means of a kernel c-list. The user c-list contains capabilities for various user objects. Some of these objects have existing kernel representations, and some do not. Further, there may exist other user c-lists that contain capabilities for the user c-list we are moving, and which already have a kernel representation.

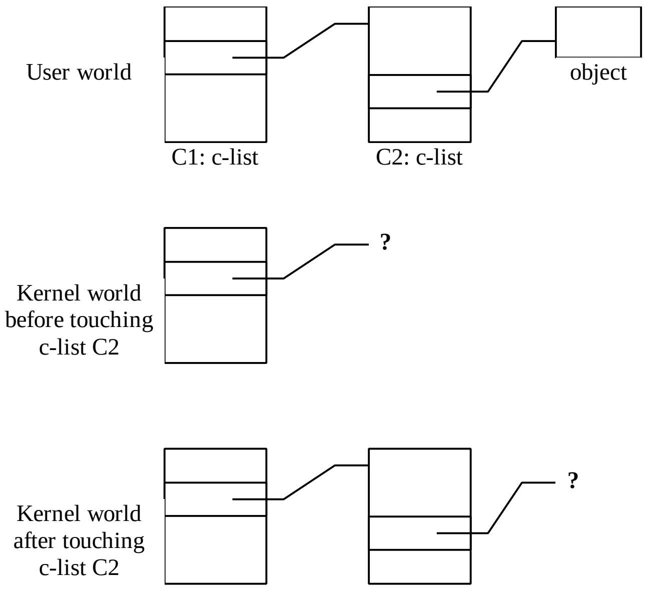

User world
object
C1: c-list
C2: c-list
?
Kernel world before touching c-list C2
?
Kernel world after touching c-list C2

Fig. 6. Moving a c-list from disk to ECS

Thus, the user system is about to perform a sequence of kernel actions with the following possible results:

- A new kernel c-list will exist.
- This new kernel c-list will contain capabilities for some pre-existing kernel objects.
- This new kernel c-list will contain capabilities for some kernel objects that do not exist.
- A capability in a pre-existing kernel c-list will now point to this new kernel c-list.

One of the kernel actions will certainly be to create the new kernel c-list. With the kernel we constructed, there is no way for a capability in a pre-existing kernel c-list to point to this new kernel object, nor for a capability in the new c-list to point to a nonexistent kernel object. Moreover, we could not conceive of a way to modify the kernel design to permit this result. The only thing we could think of was to permit the user system to convert an arbitrary collection of bits into a kernel capability. To include such an operation in the kernel would violate (1a).

8.2 A Compromise System

We eventually adopted a compromise design in which many objects could not move to the disk. User objects are essentially divided into two classes, those that can and those that cannot move to the disk. Reasoning which we no longer accept convinced us that this division was tolerable.

This reasoning was based partly on the idea that a natural division of the objects already existed. Some objects must have existence between user sessions, such as files to hold saved data and programs, and directories for file names and access rights. Other objects involved in active processing, such as processes and event channels, need not exist between sessions.

Further, there seem to be two reasons for placing an object representation on the disk. The first is to save space in fast storage. Since there are relatively few logged in users, we assumed that objects associated only with such users would occupy a small amount of space. The second is that the disk is more resistant than ECS or CM to loss of information due to a hardware or software crash. This resistance is only important for objects intended to exist between user sessions, i.e. files and directories.

Thus, we perceived no difficulty with having two classes of objects. Files and directories could migrate to the disk, and all other objects would remain in ECS.

8.3 Further Difficulties

As development of the system proceeded, we ran into a number of problems which arose only because we had these two classes of objects. Some of the problems we solved with special purpose mechanisms, and at least one was not solved at all. Examples of special purpose solutions are name tags (Section 4.5) and domain descriptors (Section 4.6). A problem for which we had no solution was that, contrary to our expectations, objects with no disk representation occupied a significant amount of ECS space.

Name tags were introduced to solve the problem that some kernel objects were not as ephemeral as we bad thought. In particular, the kernel objects used to communicate with the various input-output devices (Section 3.10) were reconstructed essentially the same each time the system was restarted. Moreover, access to these devices was controlled by controlling access to the interface kernel objects. Hence we provided name tags as objects which could stand for kernel objects and could appear in directories.

Domain descriptors were introduced to provide protected prescriptions or templates for constructing domains. Since operations were not representable on the disk, and hence were ephemeral, it never occurred to us to use operations for this purpose.

8.4 Other Systems

In Cal, the kernel provides an *internal* (ECS) representation, while the user system provides an *external* (disk) representation. This same division can appear in other contexts. For example, consider a system designed to run on several computers that are interconnected by a network. Suppose this system is to implement objects that move from computer to computer. Further, suppose it is desirable to have a local system on each individual computer that can defend itself against mishaps on the other computers. The relationship between this local system and the global system is similar to the relationship in Cal between the kernel and the user systems. That is, the objectives stated earlier for layering can be reinterpreted in this context. In fact, (1a) must be even stronger since the local system must defend itself against incorrectly transmitted object descriptions.

8.5 A Single Class of Objects

Recently, we have realized that it is possible to construct a user system that provides a single class of objects, and satisfies the objectives stated earlier. This scheme requires modifications to the kernel which come close to violating objective (1a), without actually doing so.

The first modification is to allow each kernel object a new possible state, inactive. An inactive object contains no other state information, not even a type. The create operations are augmented by an operation to convert an inactive object into an active object of a specified type. The destroy operations are augmented by an operation to deactivate an object. (it is now possible to change the type of a kernel object, by first deactivating it, and then activating it with the new type.)

The second modification is to introduce an operation for swapping the state of two kernel objects, one of which is inactive. This will permit the user system to construct a kernel representation of an object in stages, while making the construction appear atomic to user programs. The Cal system already has a similar operation for file pages (Section 3.2).

The third modification is to introduce an operation for creating a capability. To this end, the space of kernel objects is divided into *regions;* a capability for a region authorizes the creation of objects in that region. The capability creating operation takes a region, a type, an integer that specifies the desired value, and bits that specify the rights.

Dividing the object space into two regions is sufficient to protect the kernel. One region contains the kernel created objects such as those used for input/output interfaces. The other region contains objects created on behalf of the user system. No capability for the first region exists. For generality we might also introduce an operation to create a subregion of a given region, so the construction can be applied recursively.

It should be observed that this new ability to create arbitrary capabilities does not substantially increase the powers available to a user system. In Cal, the user system has available to it an allocation block from which all kernel objects created on behalf of the user system are dependent. The kernel provides an operation that returns a capability for any object authorized by such an allocation block. We have simply extended this fabrication power to capabilities for all objects, active or inactive, in a given region.

Implementation of these changes should not be difficult. An object is identified by its unique name, and the MOT index in the capability is used only for efficiency. An inactive object has no representation and no MOT entry. A request to activate an inactive object results in the selection of an arbitrary MOT index for the representation. The method described in Section 6.3 for moving MOT entries is used to ensure that already existing capabilities for the object will remain valid. If no MOT entry with the specified unique name can be found, then the named object must be inactive. In this case, the kernel action makes an abnormal return, just as in the case of a missing file page (Section 4.2). Finally, the regions can be implemented as ranges in the value of the unique name.

With this new kernel it is possible to design a user system which can move all objects to and from the disk. An inactive kernel object is used to represent a user object whose actual representation is on the disk. Consider again the problem of moving a user c-list X from disk to ECS. The disk representation X_d includes the kernel unique name for X, as well as a description of each capability in X. Since X is on the disk, the kernel object X_k used to represent it is inactive.

The user system proceeds by creating a temporary c-list C. It then examines the descriptions of the capabilities in X_d. For each such description c_d it constructs an appropriate capability c_k which it puts into C. If c_d is for a kernel object in the second region (user system created kernel objects), the user system fabricates c_k using the capability creating operation. If c_d is in the first region (kernel system created objects), then the user system obtains c_k from a master c-list of kernel constructed objects, being careful to remove any rights which c_d does not call for. Finally, the user system swaps the states of C and X.

9. Conclusion

We have outlined the design of a layered, capability-based operating system for the Control Data 6400. Some aspects of the design were quite successful: the use of capabilities, the idea of protected layering, the conversion of input-output devices into processes with a minimum of interpretation. Some aspects were definitely bad: the attempt to provide the illusion of a mapped address space on unsuitable hardware, and the way in which the disk was incorporated into the memory hierarchy. The system was too large and too slow, but it was quite reliable and did a great deal, considering the amount of work that was put into it.

We have discussed several areas in which improvements in the design now seem to us to be possible. The most important of these is the problem of extending a layered system to include a new level of memory, which we now believe can be done in a quite general way.

Acknowledgments.

Parts of the system design were done by Bruce Lindsay, Paul McJones, David Redell, Charles Simonyi, and Vance Vaughan. A number of other people made important contributions to the implementation. Jim Gray and Jim Morris contributed valuable advice and some user system facilities. The project was supported by the Computer Center of the University of California at Berkeley. We are indebted to numerous readers of early drafts of this paper, and to the referees, for whatever clarity it now has.

References

1. Bobrow, D.G., et at. Tenex.. a paged time-sharing system for the PDP-10. *Comm. ACM* **15**, 3 (March 1972), 135-143.
2. Brinch Hansen, P. The nucleus of a multiprogramming system. *Comm. ACM* **13**, 4 (April 1970), 238-241, 250.
3. Dahl, 0-J., and Hoare, C.A.R. Hierarchical program structures. In *Structured Programming***,** Academic Press, New York, 1972.
4. England, D.M. Capability concept, mechanisms and structure in system 250. Symp. on Protection in Operating Systems. IRIA, Roquencourt 78150 Le Chesnay, France, Aug.1974, 68-82.

5. Fabry, R.S. Capability-based addressing. *Comm. ACM* **17**, 7 (July 1974), 403-412.

6. Gray, J. et al. The control structure of an operating system. IBM Research Rep. RC 3949, Watson Research Center, Yorktown Heights, N.Y., July 1972.

7. Lampson, B.W., et al. A user machine in a time-sharing system. *Proc. IEEE* **54**, 12 (Dec.1966), 1766-1774.

8. Lampson, B.W. Dynamic protection structures. *AFIPS Conf. Proc*. **35** (1969 FJCC), AFIPS Press, Montvale, N.J. 1969, 27-28.

9. Lampson, B.W. On reliable and extendable operating systems. *State of the Art Report*, Vol.1**,** Infotech Ltd., Maidenhead, Berkshire, England, 1971.

10. Lampson, B.W., et al. On the transfer of control between contexts. In *Lecture Notes on Computer Science* **19,** Springer-Verlag, Berlin, 1974.

11. Meyer, R.A., and Seawright, L.H. A virtual machine time sharing system. *IBM Systems J.* **9**, 3 (1970), 199-218.

12. Morris, J.H. Protection in programming languages. *Comm. ACM* **16,** 1 (Jan. 1973), 15-21.

13. Schroeder, M. D., and Saltzer, J.H. A hardware architecture for implementing protection rings. *Comm. ACM* **15,** 3 (March 1972), 157-170.

14. Sturgis, H.E. A Post-mortem for a time-sharing system. Ph.D. Thesis, U. of California, Berkeley, and Rep. CSL 74-1, Xerox Research Center, Palo Alto, Calif., Jan.1974.

15. Wulf, w., et al. Hydra The kernel of a multiprocessor operating system. *Comm. ACM* **17,** 6 (June 1974), 337-345.

16. Wulf, W., et al. Overview of the Hydra operating system development. *Operating Systems Rev***. 9**, 5 (Nov. 1975), 122-131.