MAC TR-140

NAMING AND PROTECTION IN EXTENDIBLE OPERATING SYSTEMS

David D. Redell

November 1974

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

MAC TR-140

NAMING AND PROTECTION IN EXTENDIBLE OPERATING SYSTEMS

David D. Redell

This report reproduces a thesis submitted to the University of California, Berkeley, on September 23, 1974 in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science

Publication of this report was sponsored by the Computer Systems Research Division of Project MAC, an M.I.T. Interdepartmental Laboratory and was supported in part by the Air Force Information Systems Technology Applications Office (ISTAO) and by the Advanced Research Project Agency (ARPA) of the Department of Defense under ARPA order No. 2641 which was monitored by ISTAO under contract No. F19628-74-C-0198; and in part by Honeywell Information Systems Inc.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

NAMING AND PROTECTION IN EXTENDIBLE OPERATING SYSTEMS

David Day Redell

Abstract

The properties of capability-based extendible operating systems are described, and various aspects of such systems are discussed, with emphasis on the conflict between free distribution of access privileges and later revocation of those privileges. The discussion culminates in a set of goals for a new capability scheme.

A new design is then proposed, which provides both type extension and revocation through the definition of generalized sealing of capabilities. The implementation of this design is discussed in sufficient detail to demonstrate that it would be workable and acceptably economical.

The utility of the proposed capability mechanism is demonstrated by describing two facilities implementable in terms of it. These are: (a) revocable parameters for calls between mutually suspicious subsystems, and (b) directories providing a civilized medium for the storage and distribution of revocable capabilities.

Acknowledgments

First, I would like to thank my thesis advisor, Professor R.S. Fabry, for providing that skillful blend of encouragement and constructive criticism which constitutes good advice. I am also indebted to the other members of my committee, Professor James H. Morris and Professor Martin Graham, for reading and commenting on earlier versions of this thesis.

It is a pleasure to thank the others who read and commented on earlier drafts, including Dr. James Gray, Dr. Butler Lampson, Gene McDaniel, Dr. Bernard Peuto, Dr. Howard Sturgis, and especially Paul McJones. Earlier conversations with Bruce Lindsay also underlie much of the work described here.

Ruth Suzuki deserves the credit for the extremely fast and accurate typing of the final draft of this thesis.

Most of all, I thank my wife Connie, not only for her patience and understanding, but for typing the rough draft as well.

Contents

Page

Abstract i

Acknowledgments ii

Chapter 1: Introduction 1

1.1 Overview 1

1.2 Protection 2

1.3 Framework for Discussion 4

1.4 The Computer Utility 6

1.5 Extendibility 8

1.6 Thesis Plan 9

Chapter 2: A Typical Capability System 11

2.1 A Typical Capability System 11

2.2 Implementation of Capabilities in TCS 21

2.3 Revocation of Access Privileges 38

2.4 Indirection Through Link Segments 54

2.5 Type Extension 61

2.6 Hierarchies of Objects and Types 71

2.7 Type Extension Using Sealed Capabilities 77

2.8 Goals for a New Capability System 82

Chapter 3: A New Capability System 83

3.1 A New Capability System 83

3.2 Design Considerations for Revocation 83

3.3 Interactions with Type Extension 96

3.4 Generalized Sealing 97

3.5 Examples of Generalized Sealing 106

3.6 Implementation of Generalized Sealing in NCS . . 113

Page

3.7 Some Implementation Details 130

3.8 Possible Elaborations on the Design 134

Chapter 4: Two Facilities Using the New Capability System . 137

4.1 Possible Facilities Using Generalized Sealing . 137

4.2 Revocable Parameters 138

4.3 Directories 143

Chapter 5: Summary and Conclusions 154

5.1 Summary 154

5.2 An Area for Further Research 154

5.3 The Future of Protection 157

References 158

Chapter 1

Introduction

1.1 Overview

Computers have been with us now for just over a quarter of a century. Although their ultimate potential impact on society is still hard to predict, it seems safe to say that they will rank with such transforming inventions as the printing press and television in their effect not only on the way we live, but also on the way we think. Already their role has shifted from that of simply high speed calculating tools to a more fundamental function as the natural repository for an increasing amount of society's body of information. The near future should see the development of computer utilities bringing reliable and economical computer access to the general public, in the form of services of unprecedented scope and power [Fr 74].

These new roles of computers raise many serious social questions which are far from being answered [Ro 74, DF 65, HEW 73]. Moreover, even if these questions are satisfactorily answered, the resulting policies will require an appropriate technological framework within which they can be expressed and enforced [Po 74, Pe 74]. Thus, such social and legal issues as privacy, secrecy, confidentiality, and accountability generate a technological problem which could be called the "total system security problem."

The main subject of this thesis is <u>protection</u>. Protection is that aspect of the total system security problem which deals with the control of access by programs running within a computer system to information stored within the system [La 71, Jo 73]. It is thus concerned with prevention of undesired accesses, whether accidental or malicious. Protection is intimately involved with the naming mechanisms used by programs to specify which items of information they wish to access. We will discuss system designs which provide both naming and protection in a single integrated mechanism [DVH 66, Fa 74]. We also emphasize the notion of freely distributable access privileges, in the sense that any possessor of a privilege may pass it on as he sees fit [La 69]. On the other hand, we recognize the importance of allowing later revocation of such privileges. The main result of the thesis is the description of a naming and protection mechanism allowing both free distribution of privileges and subsequent revocation in an orderly way.

Another desirable characteristic of naming and protection mechanisms is extendibility [La 69b, Wu 74]. This property allows the construction of the system in layers or "levels of abstraction" [Di 68b], thus increasing reliability and allowing user-written extensions to augment the system with new services in a uniform way. The extendibility of the proposed mechanisms will be discussed in some detail.

1.2 Protection

The protection problem is only one aspect of the total system security problem. Thus, in discussing the protection problem, it is important to delimit the scope of the discussion by distinguishing several other closely related problems, including:

a) Hardware reliability

b) Physical security

c) User authentication

d) Personnel certification

All of the above problems exhibit two rather unfortunate properties:

1) They do not admit of complete solutions, but only of soluions quantitatively comparable in terms of cost-effective prevention of trouble (e.g. high penetration cost, long mean-time-between-failures, etc.)

2) The failure of a solution to any one of them can undermine the entire protection system.

On the other hand, if we hypothesize a situation in which problems (a) through (d) have been completely solved, we can consider the protection problem as occurring in a self-contained artificial universe, free of such real-world distractions as locks which can be picked and circuits which can burn out. Within this idealized framework, the protection problem does admit of complete solutions in many important situations [La 74]. This is not to say, of course, that all solutions constructed within such a framework are automatically complete. For example, one can protect data by requiring accessing programs to provide a password or key authorizing the access [La 69]. Internal passwords, like external passwords, are vulnerable to guessing, and are thus not a complete solution. On the other hand, one can implement internal keys which are unforgeable, opening locks which are unpickable, thus providing a complete solution to the problem. The significance of this lies not primarily in the reduction of the probability of failure (from

What do you say about [illegible] [illegible] negligible to zero) but in the conceptual shift in how one views the mechanism (with absolute confidence, rather than quantitative optimism).

It can be argued that the above viewpoint is unrealistic, since problems (a) through (d) do not admit of complete solutions as hypothesized. The point, however, is that this factorization of the total security problem allows one to take a very rigorous approach to the situation in which malicious intent manifests itself in the behavior of high speed internal computations. This is precisely the situation in which our intuitions are least likely to prove reliable in assessing the quantitative adequacy of incomplete solutions.

1.3 Framework for Discussion

For our purposes, we can regard the function of the operating system as being the transformation of the basic hardware resources of the computer into a universe of abstract resources or _objects_, and a set of _operations_ for manipulating those objects. This point of view is often referred to as the _object-oriented_ approach, and the collection of operations as the _abstract machine_. Each object has an attribute called its _type_, which determines the set of operations which can meaningfully be applied to the object. Various types of objects are provided, most notably _processes_. Processes are the active entities in the system, capturing the intuitive notion of a "locus of control" or "execution point." Processes can attempt to access other objects in the system by performing various operations on them, and it is these accesses which are checked and allowed or disallowed by the protection mechanisms of the system. At any given time, a process has some set of privileges, specifying which operations it may perform on which objects. This set of privileges is called the domain in which the process is executing. The privileges available to a process can change as a result of either:

a) addition or removal of privileges in its domain of execution, or

b) switching to a different domain of execution.

Thus, domains themselves have an independent existence and are objects in their own right. (The reasons for taking this point of view will become clear in Chapter 2.) A domain can be characterized as a passive object, serving to control the execution of an active process. It will often be convenient, however, to refer to the actions of a process executing in a domain as being performed by the domain itself, and we will use this active characterization when there is no danger of ambiguity.

The domain model is general enough to describe most protection schemes found in existing systems [La 71]. We are interested in a particular class of such schemes in which a domain consists of a set of capabilities [DVH 66, La 69, Fa 74]. A capability serves both as the name of an object and as a set of privileges to access that object. Thus, in a capability system, a domain is able to name only those objects to which it has access via its capabilities. Those capabilities are stored in the memory of the domain, which we will assume consists of a number of segments [De 65, BCD 72], each of which comprises a variable length array of addressable items. A domain may copy its capabilities and distribute them as it sees fit, although it may not, of course, make arbitrary modifications to them. Thus, capabilities are like data "sealed in a box," a characterization which we will pursue in some detail later.

1.4 The Computer Utility

The mechanisms discussed in this thesis would be useful in any computer system. The context which maximizes their importance, however, is that of the computer utility. The notion of a computer utility has received considerable attention in the literature [CV 65, Sa 66, Sc 72, Fr 74] and seems likely to play an increasingly important role in the future. In such a utility, a large user community shares an appropriately large information storage and processing facility in much the same manner that the users of electrical and telephone utilities share the corresponding power generation and communication facilities. Such _physical_ sharing (i.e., sharing of physical resources) provided the original motive for developing multi-user computer systems. That motive was the desire to lower the cost of hardware resources through economies of scale and statistical smoothing of load fluctuations. This is gradually being rendered less important by the continual decline in hardware costs. A much more fundamental motive remains, however, which is in itself more than adequate justification for building a computer utility. This is the desire for flexible _logical_ sharing (sharing of information) between users, so that they may build upon each other's work [Sa 66, De 68].

Since the user community of a computer utility consists of the public at large, the logical sharing within that community takes on more the character of transactions in a marketplace than of informal friendly cooperation [Fr 74]. In particular:

a) Sharing is often financially motivated.

b) The parties involved may not trust each other.

Point (a) implies that sharing often represents sale or rental of the shared objects. The rental case is a strong test of the protection and accounting mechanism of the computer utility. This is particularly true in the case of subletting, in which access to a rented object passes through several hands before reaching the end user. Point (b), which is in part a result of (a), reflects the fact that the standard attitude of the parties involved in a transaction in any market place is usually some degree of mutual suspicion. Since programs in the system serve as the agents of users on the outside, the programs themselves also exhibit mutual suspicion. More detailed discussion and examples of mutual suspicion can be found in Lampson [La 69] and Schroeder [Sc 72].

One aspect of the mutual suspicion problem which can be awkward to handle is the fact that the degree of suspicion between two users may change with time. For example, an employee may join or leave a company, or a renter may be late in paying his bill. Thus, it is important that the privileges of a given user or program to access a given object be able to change with time. Moreover, it is very desirable that these adjustments of privileges be as painless as possible. We will address this issue at some length, particularly in the case of increasing suspicion where previously granted privileges are to be revoked.

1.5 Extendibility

The construction of a large operating system is a formidable task. As the richness of the user environment provided is increased, so also is the size and complexity of the system which provides it. In fact, unless controlled by a suitable design methodology, the complexity of a large operating system may preclude its ever being completely debugged. One of the most promising such methodologies is that of layering, in which the system is constructed as a base-level* and a series of extensions. Each layer extends the environment in which it runs, thus presenting a richer environment for higher layers. The key assumption in such a system is that no layer has embedded in it any knowledge of the functioning of higher layers. This, combined with the obvious precaution of protecting lower layers from interference by higher layers, yields a structure in which changes to and malfunctions of higher layers cannot affect the correct functioning of lower layers in any way.

The construction of a layered system can be viewed in two ways. From a top-down point of view, the task is one of appropriately dividing the desired set of functions into a sequence of layers. From a bottom-up point of view, the task is to transform some pre-existing system into a more complete environment by adding useful new features. The latter point of view is most appropriate in the

*Sometimes called the "kernel" [Wu 74] or "nucleus" [Ha 70].

case of user-written extensions, although to a large extent, the exact distinction between system programs and user programs becomes unimportant in a layered design.

Given the object-oriented point of view discussed above, the appropriate way to view extensions is as defining new types of objects and providing the appropriate operations on them. This immediately raises the question of how such objects are named and how access to them is controlled. It is most desirable for the base-level naming and protection mechanisms to provide these functions for all higher level objects in the system. We will describe various *type extension* features which allow this.

1.6 Thesis Plan

Since the mechanisms described in this thesis represent further developments of ideas found in several existing or proposed computer systems, it is appropriate to summarize those ideas. Therefore, Chapter 2 begins by describing a hypothetical system exemplifying the relevant features of those systems, and goes on to discuss the use of those features in various situations, placing special emphasis on revocation of privileges and on type extension. The chapter concludes with a list of goals derived from these discussions.

The central portion of the thesis is Chapter 3, which proposes a new system design satisfying the goals derived in Chapter 2, and discusses the implementation of that design in some detail. Some possibilities for further elaboration of the design are also discussed briefly.

Chapter 4 examines the use of the mechanisms of Chapter 3 in providing two facilities helpful in common situations: revocable parameters for mutually suspicious subsystem calls, and directories, for storage and distribution of capabilities.

Finally, Chapter 5 summarizes the results of the thesis and briefly evaluates their significance.

Chapter 2

A Typical Capability System

2.1 A Typical Capability System

The central goal of this thesis is the detailed specification of a proposed behavior for capabilities, and the description of an efficient implementation of capabilities exhibiting such behavior. The main aspects of capability behavior to be examined are the distribution and revocation of privileges, and type extension. To bring the issues into focus, we sketch a hypothetical system called "TCS" (for "Typical Capability System") to serve as a context for discussion and as a starting point from which various improvements can be explored. This typical system as described below is not identical to any existing or proposed system but contains features found in many previous systems, including CAL-TSS [La 69, St 73], Magnum [Fa 68], Plessy 250 [En 72, Co 72], HYDRA [Jo 73, Wu 74], Project SUE [Gr 71], BCC 500 [La 69], and Multics [BCD 72, CV 65, Sa 74].

In the definition of TCS, two conflicting considerations influence the level of detail at which the various features should be described. On the one hand, it is important that the definition be specific enough to make subsequent discussions clear and unambiguous. On the other hand, the inclusion of extraneous detail would not only cloud the issue, but might also falsely appear to restrict the class of systems to which our subsequent improvements are applicable.

For these reasons, the definition that follows tends to pin down only those details which are relevant to the later discussion. In other cases, several alternatives may be sketched, or the fine points may be glossed over entirely when not sufficiently interesting.

In defining TCS, a logical place to begin is with the capabilities themselves. As stated previously, a capability serves both as the name of an object and as a package of privileges allowing the object to be accessed in various ways. It is also desirable to distinguish between objects of different types; in TCS this distinction is carried in the capability, rather than in the object itself, for reasons which will become clear during the discussion of type extension. Thus, a capability for an object contains:

a) the _unique identifier_ or "ID" of the object*,

b) the _type_ of the object,

c) a set of _privileges_ to access the object.

Each domain in TCS has its own segmented _address space_. (As pointed out by Fabry [Fa 74], freely copyable capabilities eliminate the need for communicating domains to share a common address space.) The capabilities possessed by a given domain are stored within the segments of its address space. At the same time, those capabilities serve as the skeleton which defines and structures that address space. (It is worth emphasizing that an address space defined by freely copyable capabilities tends to be a much more fluid structure than a more conventional address space defined by system data structures.) Associated with each domain is a single

*The object ID has sometimes been referred to as the "unique name" or "global name" of the object. We wish to avoid this terminology to emphasize the fact that it is the capability itself which should be thought of as the global name of the object.

implicit segment, which serves as the "root" of its address space.*
A capability for the implicit segment is part of the definition of the domain. All other segments (or objects of other types) are addressed via capabilities in this implicit segment. There is no fundamental reason, however, to restrict capabilities to appear only in this implicit segment; in fact, it will be assumed here that capabilities and "normal" data can be freely intermixed in any segment. (Ways of implementing this without compromising the integrity of the capabilities will be discussed later.)

Outside the context of any particular address space, we can define the absolute address of an item (capability or datum) to be a pair $<C,d>$, where C is a capability (for a segment) and d is a displacement (word, byte, or bit number). Let (C,d) denote the contents of address $<C,d>$. Then if C_I is a capability for some domain's implicit segment, a simple address w issued by that domain corresponds to the absolute address $<C_I,w>$ (i.e., word w of the implicit segment). Similarly, the standard notion of the two part address $s|w$ of word w in segment s is equivalent to $<(C_I,s),w>$. When capabilities can be stored anywhere in the address space, addresses involving them can become more complicated, such as $s|w_1|w_2 \equiv <((C_I,s),w_1)w_2>$ (where both $<C_I,s>$ and $<(C_I,s),w_1>$ must contain segment capabilities). This suggests the provision of direct hardware implementation of these multi-level addresses and/or programmable capability registers to hold

*This is similar to the Multics descriptor segment [BCD 72] or the CAL-TSS working C-list [St 73]. In the MAGNUM [Fa 68] and Plessy 250 [En 72] machines, it is effectively implemented in hardware in the form of several capability registers. Lampson [La 74] refers to the implicit segment as the "access point" of the domain.

intermediate capabilities during the evaluation of such addresses. Lacking these features, a domain could directly utilize only capabilities in its implicit segment; all other capabilities would have to be copied into the implicit segment before use. Various forms of multi-level addressing have been provided in existing systems [Ha 72, St 73, Ne 72, Wu 74].

why does S_0 include S_4, but S_2 [illegible] include S_3 and S_4 as far as D_2 is [illegible]

Figure 2.1-1 depicts two domains D_1 and D_2, whose implicit segments are S_1 and S_2 respectively. The address space of D_1 includes segments S_1, S_3, S_4 and S_5. The address space of D_2 includes S_2, S_1, S_5, S_6, and S_7. Note that S_1 and S_5 are shared by both domains, and that the address space of D_2 may in fact include (indirectly) the entire address space of D_1, depending upon the privileges in D_2's capability for S_1.

As mentioned in Chapter 1, domains can be characterized as either active or passive objects. In its passive role as a collection of privileges, a domain in our typical capability system is identical to its implicit segment; from this point of view, the distinction between a domain and a segment is simply a question of emphasis. On the other hand, in its active role as an exerciser of privileges, a domain is sure to require additional information in its representation, relating to control structures, error handling, entry points and so on, which we will call its <u>domain-descriptor</u>. While the exact details of this extra information are not relevant to the current discussion, it will sometimes be useful to distinguish between the domain in this larger sense, and its implicit segment.

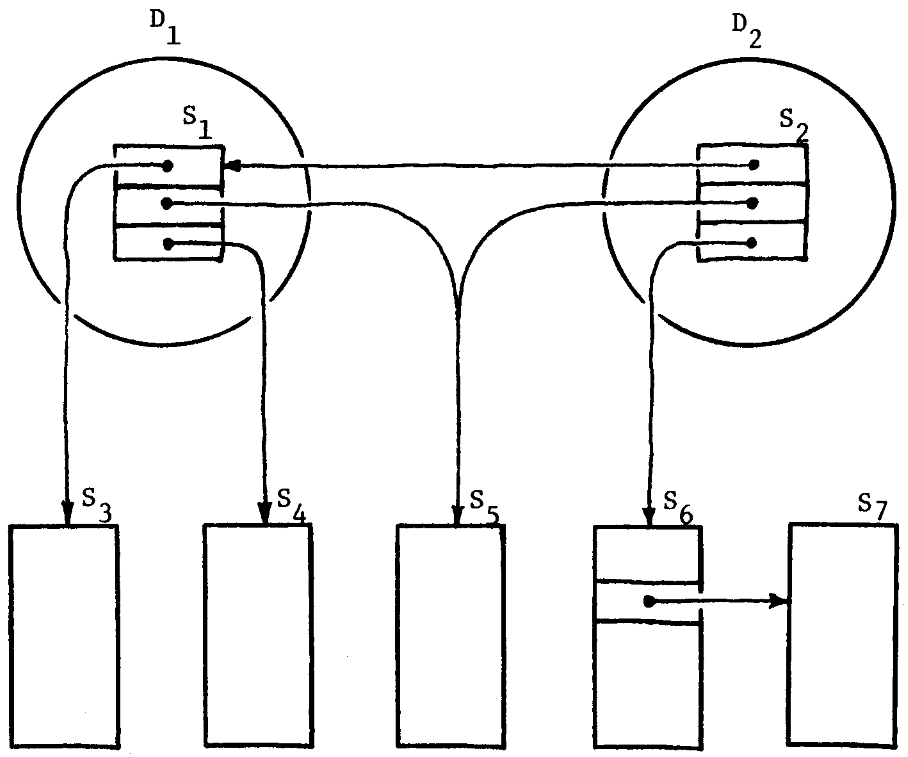


Figure 2.1-1: An example of two domains

The active characterization of domains is somewhat imprecise, since, strictly speaking, nothing is ever done by a domain but always by a process executing in or associated with the domain. This raises the issue of the exact relationship between domains and processes.* Since protection and scheduling are essentially independent functions, it is tempting to define domains and processes independently, and to allow processes (at least potentially) complete freedom to choose their domain of execution. This implies that

a) A given process may execute in various domains at different times.

b) A given domain may have zero, one, or several processes executing in it at any given time.

In such a scheme, two types of communication mechanisms are required. One is _interprocess communication_, which allows two parallel processes, in the same or different domains, to synchronize their execution and exchange messages. The other is _interdomain communication_, which occurs at the point in time when a process crosses from one domain into another. This is generally viewed as being much like a procedure call/return sequence, including the passing of parameters, and is thus referred to as a _domain-call_. This will be discussed in more detail later.

In actual systems, one or both of two simplifying restrictions is often imposed. The first restriction is to force a given process to always execute in the same domain. This eliminates the rather complex machinery needed for domain-calls, and forces all

*Called "environment binding" by Jones [Jo 73].

inter-domain communication to be cast as inter-process communication. While this is clearly a simplification of the base-level system, in practice it often forces higher level software to essentially simulate domain-calls using multiple processes, only one of which is active at any given time. This is not only inefficient, but can also be surprisingly clumsy, considering that parallel processes seem to be such a powerful construct. Indeed, the unused potential parallelism seems to cause much of the clumsiness. [illegible]

The other restriction which is often applied is to allow only one process at a time to execute in a given domain. This can be done dynamically, treating the domain as a "critical section," but is more often done statically, by associating each domain with a single process, and allowing only that process to execute in it. One reason for making this restriction is the previously mentioned correspondence between domains and address spaces. As pointed out by Lampson [La 69] this tends to result in address conflicts between multiple processes executing in the same domain. One way to avoid these conflicts is to equip each process with special base registers, or a pushdown stack for working storage, but what such mechanisms really provide is simply the ability for each of the processes executing in a given domain to see the domain somewhat differently, in a rather stylized way. A more straightforward and flexible approach is to actually provide a different "copy" of the domain for each process, and to use the standard sharing mechanisms to avoid redundant storage of the identical components of these domains, (e.g. pure procedures, unchanging capabilities, etc.).* In such a scheme, each process has a private set of domains, and moves among them using the domain-call mechanism. Such a scheme will be assumed in subsequent discussions of TCS, although this is not essential to the proper functioning of the improved capability mechanisms proposed later.

Given that a domain possesses some capability, it is allowed not only to use the capability to access the indicated object, but also to manipulate the capability itself in certain carefully constrained ways, including:

a) Copying: the capability may be freely copied at any time, here denoted by a simple assignment

$$C_b \leftarrow C_a$$

b) Reducing privileges: the privileges in the capability may be reduced, here denoted by

$$\text{reduce}(C_a, P)$$

where P is a mask indicating the subset of C_a's previous privileges which are to be retained.

In some systems [St 73] these two operations have been combined; here, they are presented separately to ease the later transition to an improved scheme.

One use of the mechanisms described so far would be the

*We will assume that a domain is created by an explicit create-domain operation, and remains in existence until destroyed [St 73]. A more complicated approach provides the automatic creation of a domain whenever a call is directed to a global domain-prototype object [Wu 74].

passing of capabilities between domains via shared segments. In one sense, this is a very powerful feature, since it allows any possessor of a privilege to pass it on without requiring any sort of approval by the original donor of that privilege (except in the special case in which the donor is empowered to disallow all such sharing; e.g. in the case of a "confined" subsystem [La 73]). In another sense, however, this feature is very weak, since it provides only a relatively costly, clumsy and unstructured method of inter-domain communication. This weakness would be particularly evident in the case of mistrust between domains (e.g. "mutually suspicious" subsystems). Both of these considerations suggest that the domain-call mechanism should provide for the passing of capabilities, as well as data, as parameters. The latter consideration suggests the utility of such a feature, while the former shows that the ability to keep a domain from giving away its privileges is already eliminated by freely copyable capabilities and is not further compromised by allowing the passing of capabilities as parameters.

Parameter passing may be less [illegible] & cheaper, than segment sharing. (Unfortunately, small segments never seem to be cheap.)

We assume that TCS allows the passing of capability parameters and implements this by copying the indicated capabilities from the calling domain (or <u>caller</u>) to the called domain (or <u>callee</u>) at the time of the call, and copying back any result capabilities at the time of the return. A domain-call thus takes the form

$$\text{call } (C_G, P_1, P_2, \ldots, P_N)$$

[illegible] authorization" ...

where the P_i are parameters (data or capabilities) and C_G is a gate capability for the callee, allowing activation at a particular entry point. Similarly, a domain-return takes the form

$$\text{return } (R_1, R_2, \ldots, R_M)$$

where the R_i are the results and the return gate is implicitly the site of the original call. We leave unspecified here such details as static vs. dynamic allocation of space for capability parameters in the receiver's address space, automatic type checking of capability parameters, and so on.

In addition to making unwanted accesses to objects, domains can misbehave by making unreasonable demands on the resources of the system [La 71]. Some mechanism must be provided to prevent them from interfering with each other in this manner. Since the details of accounting and resource allocation are beyond the scope of this thesis, we will simply assume that each domain is funded by an account, which limits its resource consumption.

One particularly tricky problem which occurs in capability systems is the "lost object problem," which arises when all capabilities for a given object are inadvertantly discarded, making explicit destruction of the object impossible, and the space occupied thus unrecoverable. Given our attitude about accounting, this is really an opportunity for self-inflicted harm, rather than malicious sabotage. Nevertheless, recovery from such situations must be possible, hence several possible solutions to the lost object problem will be discussed at appropriate points.

2.2 Implementation of Capabilities in TCS

In this section we discuss, in a fair amount of detail, certain aspects of the implementation of a system like TCS. Three considerations influence the choice of the particular mechanisms described in this section. For one thing, various systems similar to TCS have been constructed, and their implementations, although varying in many ways, have shown some common features whose advantages have become generally accepted. In addition, certain facilities not included in any existing capability system are widely regarded as desirable, hence their implementation implications are of interest. Finally, discussion of implementation of TCS is intended to set the stage for the corresponding discussion in Chapter 3 concerning the implementation of a more sophisticated capability scheme.

The most obvious necessity in implementing a capability system is some mechanism to protect the representations of the capabilities themselves from unauthorized alteration. The proper functioning of the entire system is based upon the integrity of capabilities, hence this mechanism should be simple, to maximize not only its reliability, but also its understandability, and thus inspire user confidence. Two mechanisms have been proposed, which we will call "partitioned memory" and "tagged memory."

All capability systems which have actually been constructed have used partitioned memory. As its name suggests, this scheme involves partitioning the segments in the system into two classes: capability segments which contain only capabilities, and data segments, which never contain capabilities. One obvious advantage of this mechanism is that the cost of distinguishing between capabilities and data is distributed over an entire segment, reducing the overhead per item, but the main advantage of partitioned memory is more subtle; it involves the avoidance of certain addressing complications which arise in the tagged memory approach, as we shall see shortly. The main disadvantage of partitioned memory is that the artificial division of a user's memory into two parts is inconvenient. It is often quite natural for information structures (e.g. entries in a table) to contain both data and capabilities. While such intermixing can be simulated using a pair of segments, this is a fairly clumsy procedure. For this and other reasons, discussed in detail by Fabry [Fa 74], we reject partitioned memory, as indicated by our specification of TCS as allowing free intermixture of capabilities and data in any segment.

The tagged memory approach allows such intermixture by attaching one or more extra "tag" bits to each information item in each segment. The term "item" is used here to denote the basic addressible unit of memory (word, byte, etc.). These tag bits are unmodifiable by any software except the most central routine of the base-level system. Each item's tag indicates its status as 'data' or 'capability.' An item must be tagged as a capability to be used as one. An item so tagged can be generated only by copying another such item, or by the base-level capability-creation routine. On the other hand, a tagged capability can be erased by overwriting it, either with data or with another capability. (The system could require that capabilities always be explicitly erased before their storage is reused. We reject this as too inconvenient for the user, although there are cases in which it would make things slightly easier for the system.)

The only production computers which use tagged memory are the Burroughs B5000 [Bu 61] and its descendants.* The protected items in these machines are "descriptors" rather than capabilities. The differences between the two do not concern us here, except for one: descriptors are considerably smaller than capabilities. A Burroughs descriptor is 48 bits long, while many extendible capability systems have allowed in excess of 100 bits for each capability. The impact of this will become clear in a moment.

While the advantages of tagged memory have been slowly gaining acceptance, another trend which has had even more impact is the reduction of the size of the addressable items in memory. While machines with items of 36, 48, or even 60 bits were common in the past, the byte (8 bit character) is rapidly becoming a universal standard, and strong arguments can be made for the ultimate reduction to bit addressable memories. In such schemes, a larger unit of information (e.g. a capability) is represented by a contiguous sequence of items and named by the address of its first item plus its length (implicit or explicit) in items.

There is a very real conflict between these two features. Two problems arise when the representation of a tagged capability is a sequence of addressable items in memory. One is the obvious increase in cost of associating a tag with each item as the items get smaller. The other is the possibility in such a scheme of

*Various experimental machines have used tagged memory, including the Rice computers and Iliffe's BLM. A general discussion of tagged memory is given by Feustal [Fe 73].

addressing the middle of a capability.

If we assume that each item has a one bit tag, we are faced with the question of which of the items in a capability should have their tags on (i.e., set to 'capability'). If all of their tags are on, there is no convenient way for the system to distinguish between a valid capability address, and one which points to the middle of a capability. The latter case could lead to the recognition of the last few items of one capability, together with the first few items of an immediately following capability, as constituting a valid capability, hence this ambiguity must be avoided. One way of doing this is to turn on only the tag of the first item in each capability, and require that the first (and only the first) item located by a capability address be so tagged. This makes the other items in a capability indistinguishable from data, however, and leaves them open to alteration unless every store operation scans the tags of the appropriate number of preceding items and turns them off to insure invalidation of any capability which contains the item(s) being modified.

It is clear, then, that an address pointing into the middle of a capability must be distinguished both from a valid capability address and from an address of untagged data. This suggests the need for two tag bits on each item, one indicating whether the item is part of a capability at all, and the other indicating whether it is the first item of a capability. Since the second tag is necessary only when the first one is on, it could be "stolen" from the bits of the item only when needed (although this obviously doesn't work on a bit-addressable memory, since the item would then have no bits left at all!).

The other problem, the high cost of tagging small items, exerts a strong pressure to increase the size of items. Arguments in favor of small items generally cite the fact that, for a given total bit capacity, address size grows only logarithmically with decreasing item size. Unfortunately, the cost of tagging grows linearly, reaching a maximum in the bit-addressable case of two tag bits per information bit, which is clearly out of the question.

One alternative tagging scheme which we reject allows small items but imposes the restriction that capabilities can only be stored at addresses which are an even multiple of the length of a capability. In such a scheme, memory is item-addressable for normal data, while capability addresses must locate one of the predetermined "capability frames." Such restrictions tend to complicate the software and sacrifice many of the advantages of item-addressability.

integral

A much more sophisticated scheme, which also involves the notion of a capability frame, attempts to exploit the fact that the assignment of tag bits to each item is a relatively inefficient encoding of the set of possible data/capability configurations in a given region of memory. Even if capabilities can begin at any address, the number of different arrangements in a given capability frame is not large. At most one capability can begin in a frame, and can be preceded by one or more data items and/or the trailing items of a capability which began in the previous frame. By associating with each frame the integer displacement of the capability, if any, beginning in the frame, it is possible to simulate two bit tagging of each item. This is a somewhat complicated approach, but may eventually prove to be the key to bit-addressable tagged memories, since it allows the cost of tagging, like that of addressing, to grow only logarithmically with decreasing item size. This scheme also has the rather intriguing property that reducing the size of capabilities does not always increase the efficiency of memory utilization. For a given pattern of usage, there is an optimum size for capabilities, such that deviation in either direction increases the total overhead for capability storage.* No existing system uses such a scheme, although it has been tentatively investigated by Gray [Gr 73].

We thus conclude that our implementation of TCS should use one of three tagged memory schemes:

a) Items should be single bits, and the scheme just described should be used to simulate two bit tagging.

b) Items should be a substantial fraction of the size of a capability, allowing a two bit tag per item at a reasonable cost.

c) Items should be large enough to hold an entire capability, allowing a simple one bit tag per item.

---

*Assume, for example, a bit addressable memory in which the average object is N bits long and is pointed to by k capabilities. Then the overhead for capability storage is the fraction of memory taken up by tags, plus the fraction holding the capabilities themselves. As a function of the size c of capabilities, this is

$$F(c) = \frac{\log c}{c + \log c} + \frac{kc}{N + kc} \quad .$$

For instance, if $N = 10^5$ bits and $k = 10$, the storage of 64 bit capabilities requires about 15% of memory, while reduction to 32 bits or expansion to 128 bits increases the overhead to about 17%, and 16 bit or 256 bit capabilities require about 22%.

To simplify subsequent discussions, we adopt alternative (c), although it would probably not be feasible for TCS as described, since capabilities are so large. In Chapter 3, however, we will describe a scheme in which capabilities fit into more reasonable sized tagged items.

The second major implementation aspect to be discussed is the mechanism for mapping the IDs found in capabilities into physical addresses of objects. The most obvious solution would be to simply use the physical address as the ID, but that would imply updating all the capabilities for an object whenever it was moved or deleted. This is impractical due to the proliferation allowed by free copy-ability, especially in a system allowing intermixing of capabilities and data in segments.

Most capability systems have solved this problem by localizing changeable information about objects in a system data structure and forcing all access to the object via capabilities to go indirectly through this structure, which has been referred to by such terms as "Master Object Table" [St 73], "System Capability Table" [En 72], and "Global Symbol Table" [Wu 74]. Here, we will refer to it as simply "the map."

There is a one-to-one correspondence between objects and entries in the map. An object and its map entry are created and destroyed together. Since the capabilities for an object are not updated when it is destroyed, it is not satisfactory to use the location of an object's entry in the map as its ID, since that would prevent re-use of map space freed by object destruction. In, fact, the ID of a destroyed object must clearly never be re-used, since capabilities for the old object could then be used to access the new one. This suggests that IDs should be quite long, so that the space of IDs can never be exhausted, even if objects are created and destroyed at the maximum possible rate for the entire life of the system. The alternative of occasionally stopping the system and compacting the space of IDs is plausible, but less attractive. Any generator of a sequence of unique long integers can be the source of IDs. A counter of the total number of objects created, or a real-time clock of sufficient length and resolution are the common examples. In either case, provision must be made for restarting the system after a failure without any possibility of repeating a previously used ID.

As a first approximation, we can consider the map translating such IDs into physical addresses as being implemented as a large hash table in primary memory, keyed on IDs. Figure 2.2-1 shows the representation of capabilities and map entries. (The field labeled "address" is assumed to contain any extra information necessary to distinguish between primary and secondary storage addresses. The details are not relevant here.) Each exercise of a capability involves:

1) checking the appropriateness of the action, given the type and privileges in the capability (and signalling an error otherwise),
2) hashing into the map to verify the existence of the map entry, and hence the corresponding object (and signalling an error otherwise),

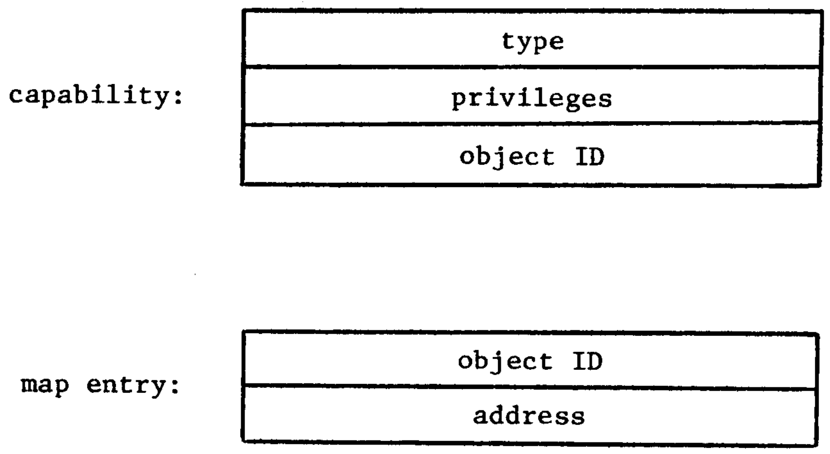


Figure 2.2-1: Format of capabilities
and map entries in TCS

3) checking the address in the map entry for the presence of the object in primary memory (and signalling an exception otherwise),

4) using the address to perform the access to the object.

These steps are simple enough to be implemented in hardware or firmware, and would be used heavily enough to justify such implementation.

As described so far, the mechanism does not deal adequately with the two extreme cases of objects which are accessed very frequently, and those which are accessed very infrequently. Objects in the former class, such as segments containing executing programs, are so heavily used that hashing into the map in primary memory is unlikely to be efficient enough. Thus, it is necessary to hold the most active map entries in special hardware.

In our implementation of TCS, this hardware takes the form of a special associative memory, each element of which can hold one map entry. The association is on IDs. On each access, the ID in the capability is first presented to the associative memory. If a matching entry is found, no reference to the map in primary memory is made. Otherwise, the standard map reference is done, and the result replaces the least active (e.g. least recently used) entry in the associative memory, as well as being used to perform the access. The effectiveness of similar hardware has been clearly demonstrated in existing systems [Sc 71].

Whenever an entry in the primary-memory copy of the map is updated or deleted, any corresponding entry must be invalidated in the associative memory. This can be done by selectively clearing the matching entry (if any) or by totally flushing the associative memory. The cost of reloading the entire associative memory on each such flush might be acceptable, but the extra complication required to do selective clearing is so low that it would undoubtedly be the method of choice. Note that total flushing of the associative memory is never logically necessary, due to the use of context-independent names as association keys. Similar mechanisms involving association on context-dependent names require total flushing each time the context (domain, process, etc.) is switched. Of course, the significance of this is entirely dependent upon the frequency of such context switching.

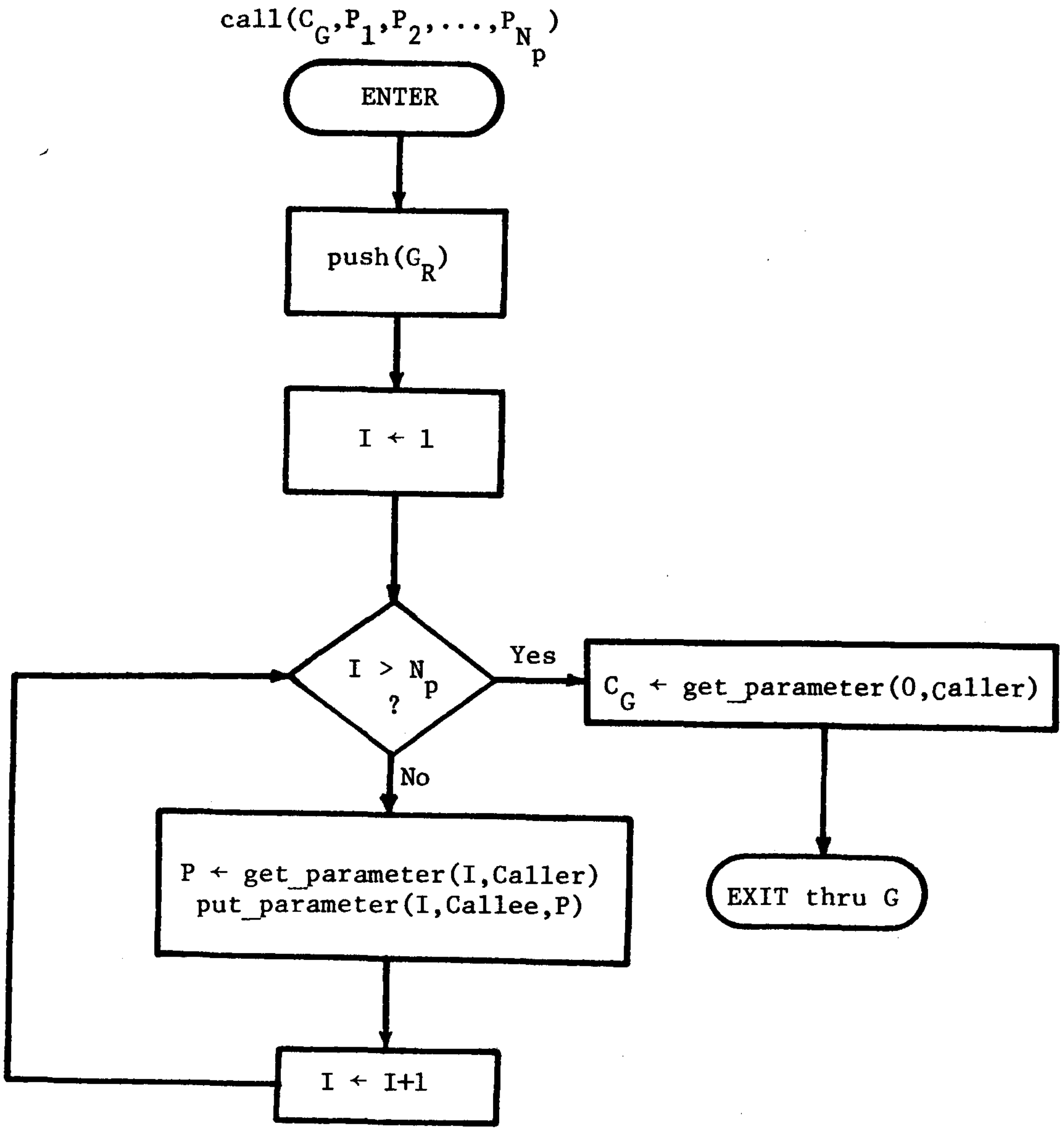
One apparent alternative to a special associative memory would be the provision of a general purpose associative memory or "cache" holding the most active items in primary memory, regardless of how they are being used. Such a cache would naturally tend to capture the most active entries in the map, and thus speed up the standard machinery for accessing via the map in primary memory. In spite of its appealing simplicity, we reject this scheme for several reasons. For one thing, a cache which is large enough to be useful for non-map items (e.g. instructions, data) is unlikely to be as fast as we can afford to make special hardware which captures only active map entries. Placing map entries in the same cache with other data also sacrifices any opportunity to access the two in parallel. In addition, the cache, by transparently speeding up primary memory, in no way bypasses the hashing necessary to locate a map entry. This means that entire "collision chains" from the map, rather than just active entries, would need to migrate into the cache, and would have to be scanned on each access, thus further degrading performance as compared with that of the special purpose associative memory. A more general way of stating all of these objections is to say that the cache simply makes the memory faster; the relative overhead for accessing map entries in memory is thus not reduced by the cache. Hence a cache, while valuable for other purposes, is not optimal for capturing active map entries.

Another alternative which has been adopted in some systems stems from the observation that active capabilities, as well as active map entries, should be held in fast hardware. To this end, programmable capability registers can be provided, into which an executing program can load capabilities before use [Fa 69, En 72]. Moreover, the map entry corresponding to an active capability is itself active, suggesting that space be provided in the register for the map entry as well. An access via such a "smart" register can then proceed directly to the object. Of course, it is still necessary to automatically reload any registers holding copies of a map entry which is updated, which adds a certain amount of complication to the mechanism. Also, the addition of programmable capability registers, whether smart or not, introduces the standard problems of register allocation, save/restore sequences, and so on, as well as the novel requirement that a calling domain explicitly erase registers containing capabilities not being passed as parameters. Other considerations in the use of capability registers are discussed by Needham [Ne 72].

We adopt for our implementation of TCS the associative memory approach rather than smart capability registers, although the preference is not a strong one. We assume that the overhead of fetching the capabilities themselves from primary memory is sufficiently reduced by transparent mechanisms such as a program-counter holding the current procedure capability, or hardware implementation of all or part of the executing domain's implicit segment.

The success of the associative memory approach is completely dependent upon the observed tendency for only a small number of objects to be heavily accessed during any given small interval of time (i.e., fraction of a second). On a coarser time scale (i.e., minutes), the same kind of behavior is observed in the sense that during a given coarse time interval most of the objects in the system will not be accessed at all. This suggests that the map entries for such objects be kept in secondary memory, and be brought into the hash table in primary memory only when needed [Fa 74]. Experience with a similar scheme (the "Active Segment Table" [BCD 72]) in Multics shows that this approach can be quite successful in saving a large amount of primary memory without incurring a significant speed penalty.

Another aspect of TCS' implementation to be discussed is parameter passing during domain calls. This is included mainly as background for a more elaborate scheme developed in Chapter 4, hence it omits details not relevant to that discussion. Figure 2.2-2 shows the workings of the domain call instruction. First, the return gate must be retained, allowing re-entry into the caller at the site of the call. This is saved in a pushdown stack of such



call(C_G,P_1,P_2,...,P_{N_p})
ENTER
push(G_R)
I ← 1
I > N_p ?
Yes
No
C_G ← get_parameter(0,Caller)
EXIT thru G
P ← get_parameter(I,Caller)
put_parameter(I,Callee,P)
I ← I+1

Figure 2.2-2: TCS domain-call operation

gates which is associated with the process.* Then the parameters are copied from the caller's address space into that of the callee. We assume the existence of two sub-operations internal to the base-level system:

$$P \leftarrow \text{get\_parameter } (I,D)$$

$$\text{put\_parameter } (I,D,P)$$

These operations serve to fetch and store the I th parameter P at the appropriate location in the address space of domain D. The actual layout of the parameters in the address space need not concern us here. We assume that N_P, the number of parameters, and G_R, the return gate, are automatically available to each base-level operation. (Most operations finish by exiting through G_R; the exceptions are domain-call and domain-return.) To simplify the discussion, we have omitted description of the copying of results from the callee back to the caller when the return is done, since this is virtually identical to the handling of the parameters during the call. Thus, Figure 2.2-3 shows only the retrieval of the return gate from the stack necessary to resume execution of the caller.

In concluding our discussion of TCS' implementation, we briefly consider two possible ways to attack the lost object problem, neither of which we regard as satisfactory. One approach is to maintain with each object a reference count of existing

*A variant of the call operation, referred to as a "jump-call" is obtained by omitting the saving of the return gate. This causes the callee to return not to the current caller, but to the previous caller. This is occasionally useful, as we shall see in Chapter 4.

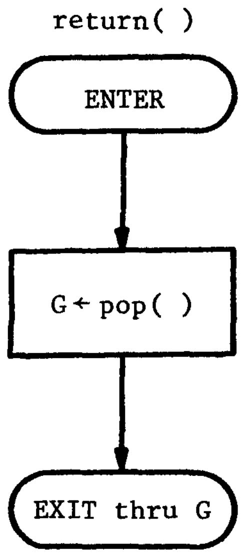


Figure 2.2-3: TCS domain-return operation
(without results)

capabilities, and to delete an object when it becomes lost, as well as when it is explicitly deleted.* There are at least three drawbacks to this approach:

a) The destruction of capabilities (e.g. through overwriting or segment deletion) must be detected and the reference counts maintained.

b) Lost self-referential structures are not deleted properly.

c) An object may be lost to the user who funds it, even though capabilities exist elsewhere.

We therefore reject the reference count approach. (For a contrary view, see Wulf, etal. [Wu 74]).

Another approach is to allow "un-losing" of lost objects by allowing a suitably authorized domain (e.g. one which owns the funding account) to request spontaneous generation of fully privileged capabilities for funded objects [CC 69]. This is rather inelegant and requires fairly complicated data structures which may or may not be otherwise necessary.

Other approaches to a base-level solution to the lost object problem can be envisioned (e.g. global garbage collection) but we choose instead to postpone the solution until a higher level of the system. Thus, the base-level system simply allows objects to become lost, and the users depend upon the directory system, as described in Chapter 4, to prevent this occurrence.

*We assume that explicit deletion is also available, since otherwise, the user who funds the object may be unable to reclaim the space occupied by it.

2.3 Revocation of Access Privileges

In the context of TCS, we now explore various approaches to the distribution of capabilities and the revocation of access privileges. As an example, we use the simple situation in which domain A wishes to grant to domain B a set of privileges to access object X.

The first approach which suggests itself is the simple copying from A to B of a capability for X containing the desired privileges, as shown in Figure 2.3-1. This is clearly the intended use of copyable capabilities, and is quite satisfactory provided that the amount of trust A has in B remains constant.* If, however, A subsequently decides that some different set of privileges is more appropriate for B, a second capability for X must be passed as a replacement. This may be quite inconvenient for B, who may have made various copies of the original capability, some of which may have been passed on to other domains. Moreover, unless the privileges in the new capability are a superset of those in the original, A must pessimistically assume that B will retain both capabilities, and thus possess the union of the privileges in the two. In other words, privileges once granted can never be revoked.

This simple example shows that the typical capability mechanism, while useful, does not adequately cope with the difficult situation of changing levels of trust, particularly when trust decreases and revocation of privileges is desired. Before proposing any

*We will generally omit the phrase "the person who owns a domain" and simply inpute feelings of "trust" and "suspicion" to the domains themselves.

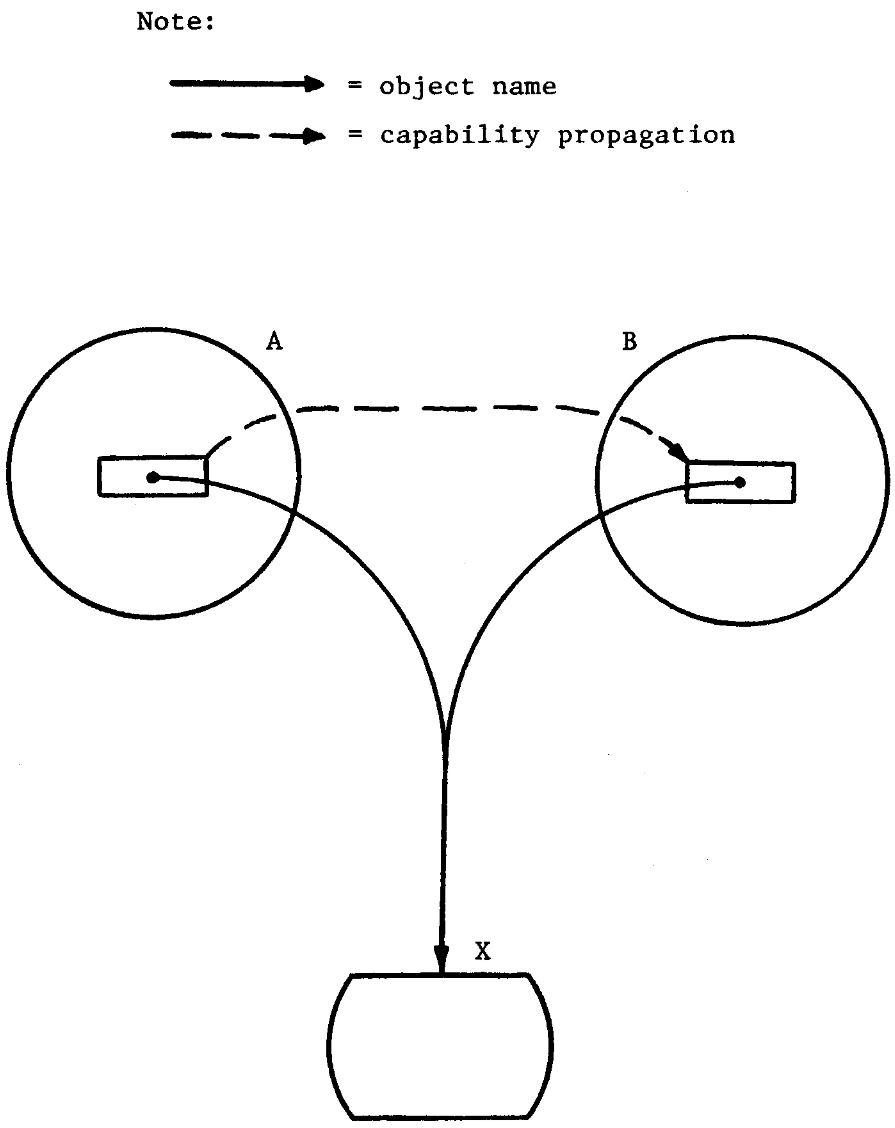


Figure 2.3-1: Passing a capability

fundamental changes to the behavior of capabilities, however, it seems appropriate to explore the various approaches which have been proposed for solving the revocation problem without making any major modifications to the underlying capability mechanism.

Caretakers: A standard "escape hatch" in most protection systems is the ability to interpose a "caretaker" domain between an object and the domains which access it. The caretaker can implement any access control protocol not provided by the system. This situation is shown in Figure 2.3-2, in which A has created a caretaker domain C, and given to B a capability to call C, rather than a capability to access X directly. Two problems are immediately evident. One is simply the inefficiency of calling C each time B accesses X. For example X may be a segment, in which case the extra domain-call is likely to cost much more than the segment access itself. The other problem is that B now receives a capability of type 'domain' rather than one indicating the type of X. Unless the system provides facilities for allowing domains to "masquerade" as objects, this will change the interface seen by B when accessing X. For example, to store into a segment, B must execute either a store-operation or a domain-call-operation, depending on whether or not a caretaker has been interposed.

More generally, one can object that the caretaker mechanism is not, in itself, a solution to the problem, but merely a framework within which a solution can be implemented. We have said nothing so far about the basis upon which the caretaker C decides to allow or refuse a given access request. In the simplest case,

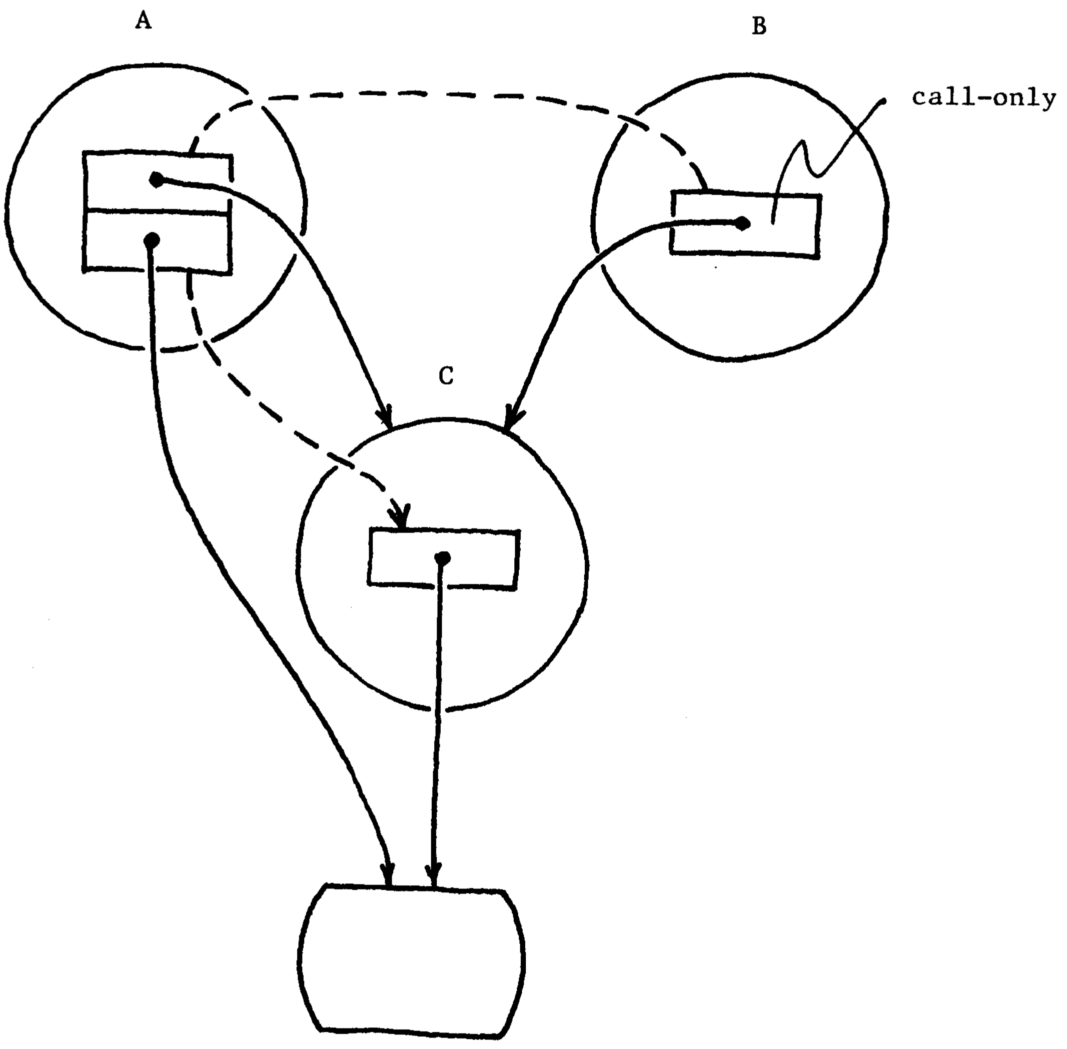


Figure 2.3-2: A caretaker domain

A specifies a single set of privileges and gives a corresponding capability to C, who exercises it each time B (or any other domain having a copy of B's capability) attempts an access. Whenever A's level of trust in B decreases, a weaker capability can be given to C. On the other hand, if A wishes to confer independently revocable privileges to access X on various domains by authorizing them all to call C, then C, given that it can distinguish reliably between its various callers, finds itself in the position of a process in Lampson's "message system" [La 71]; that is, C must essentially re-invent the system's protection machinery. This can be avoided by defining multiple caretakers for X, each allowing an independent set of privileges, as shown in Figure 2.3-3. Since the caretakers in this situation are not really making any decisions, but are merely using their privileges whenever requested, one would hope that the overhead of an actual domain call might be avoided. We will return to this point later.

Control: Most modern protection systems provide some mechanism to capture the notion of one domain being subordinate to, or under the control of, another domain. This is sometimes represented by a static domain hierarchy [St 73], but we will treat control as being a privilege which, when contained in a capability for a domain, authorizes the possessor of the capability to control that domain. (The distinction is not very important for the discussion which follows.) In our typical system, much of the power of control can be granted by giving one domain a suitably privileged capability for another domain's implicit segment, as was suggested in Figure 2.1-1, although complete control would require a

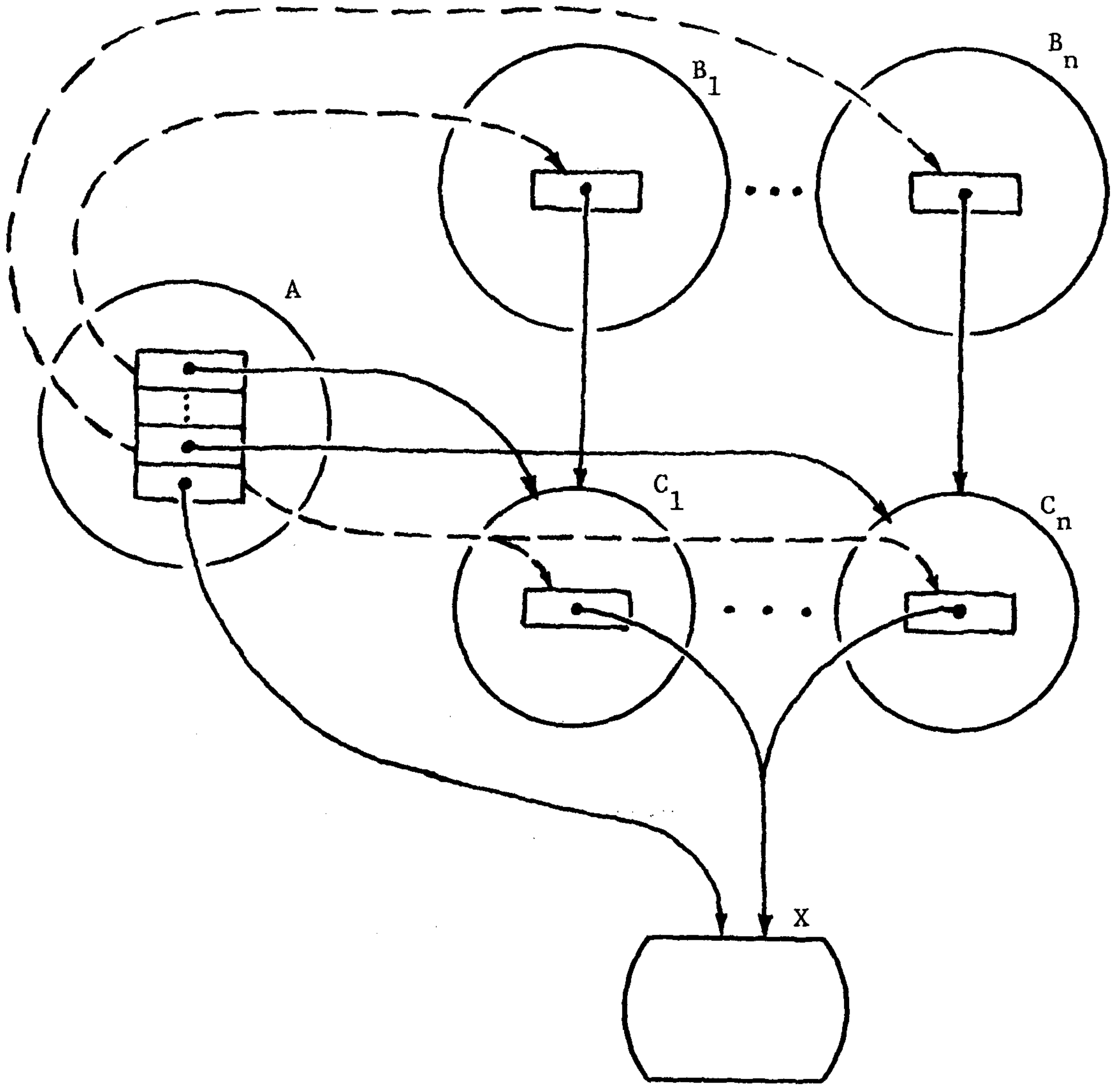


Figure 2.3-3: Multiple caretakers

capability of type 'domain' allowing access to the controlled domain's domain-descriptor.

This facility for one domain to control another is applicable to a subset of our problem of changing degrees of trust; domain A can attempt to enforce any reduction in its degree of trust of B by retain-ing control over B, although this requires that B have total and unconditional trust in A. The latter condition clearly limits the class of situations in which control of B by A is appropriate.

Even when the control facility is applicable, there are still problems with its use. It would appear that A, having given a capability for X to controlled domain B, could later search the entire address space of B, reducing the privileges in all copies of the capability to match its revised intentions. The success of this search, however, can be compromised if B is allowed to execute concurrently, making the capabilities in question "moving targets." Thus, concurrent execution by B (or any other domain able to manipulate B's address space) must be prevented, either implicitly by placement in the same process with A, or explicitly by being "stopped" by A, using its control privilege.

Even if A manages to successfully weaken the capabilities in B's address space, there remains the possibility that copies may have escaped to other domains which are not under A's control. To prevent this, A must carefully limit B's communication with other domains via shared segments, domain-call parameters, and so on. In short, B must be "confined," which, as noted by Lampson

[La 73] can be both very restrictive for B and very difficult for A. In the latter regard, however, it is worth noting that the problem of "covert channels" does not exist for capabilities, since transmission of the bits of a capability is not the same as transmission of the capability itself.

A simpler mechanism which has been proposed [La 71, Gr 72] to deal with the above problems uses a "copy-flag" contained in each capability. Originally, the flag is on to allow copying, but once it is turned off, it can never be turned back on, and all copying of the capability is disallowed. Thus, A can place a non-copyable capability for X in B's address space, and later revoke any desired privileges from that capability, confident that no other copies exist. This is even more of a restriction on B than confinement, however, since free copyability is one of the fundamental properties of capabilities. If one assumes that the passing of capabilities as domain-call parameters is done by copying, then non-copyable capabilities cannot even be passed as parameters, making them virtually useless. The scheme can be salvaged by introducing "indirect capabilities" which point to the non-copyable capability and are themselves copyable, but, as we will see later, such an indirection feature is powerful enough to completely eliminate the need for A to control B in the first place.

Ownership: The idea of one user or domain "owning" a shared object has appeared in many systems, for such purposes as accounting and resource allocation, as well as for protection. In the context of protection, the owner of an object is thought of as retaining ultimate control over the object, in the sense that any other domain's capability for the object should be subject to revocation by the owner. Ownership, like control, could be defined as a static relationship between each object and its owning domain, but again, we assume instead that 'ownership' is simply a privilege which confers 'owner' status on any possessor of a capability containing it.

As described thus far, ownership avoids the problems which limit the applicability of the control scheme. In particular, it is usable in the case of mutual suspicion, since it makes no assumptions about the relationships between domains. However, several issues have been left unresolved.

If the owner of an object wishes to revoke a given set of privileges from all outstanding capabilities for the object then the desired action is clear, if somewhat impractical. The base level system must suspend all other activity and search the address space of every domain in the system, performing the appropriate reduction on each capability for the object in question. It is worth noting that one case of such uniform revocation has a much more reasonable interpretation; if <u>all</u> privileges are to be revoked from all capabilities for the object, the owner can simply make a copy of the object and destroy the original. An even more efficient mechanism to produce the same effect can be provided in the context of the implementation in section 2.2 by simply allowing the owner of an object to change its ID, thereby invalidating all outstanding capabilities [CC 69]. (Of course, the operation must return to the owner a new capability containing the new ID.)

If the owner of an object wishes to revoke individual privileges, a global search is implied, as indicated above. If, however, the owner wishes to revoke these privileges from some but not all of the capabilities for the object, even more fundamental problems arise. The central question is how the owner should specify the set of capabilities on which the revocation is to take effect. In the context of TCS, the only obvious possibility is the specification of a set of domains in which the revocation should occur, either by listing the set, or by listing the complementary set of domains which should remain unaffected. The problem is that in a system providing freely copyable capabilities, the owner of an object is unlikely to have complete knowledge of the propagation of capabilities for that object throughout the system, and is therefore not in a position to provide either type of domain list. Figure 2.3-4 depicts the situation in which A has given capabilities for owned object X to B and C. Subsequently, B and C have passed copies of their capabilities to D and E, respectively. If A now decides to revoke some privileges from B's capability, the revocation should clearly effect D's capability, but not C's or E's. A domain list provided by A to control the revocation would specify either revocation from B, allowing D to escape, or exemption of C, incorrectly affecting E.

There are other relatively simple situations in which no correct domain list can be prepared, regardless of A's global knowledge of the distribution of capabilities among domains. Figure 2.3-5 depicts such a situation, in which domain D has

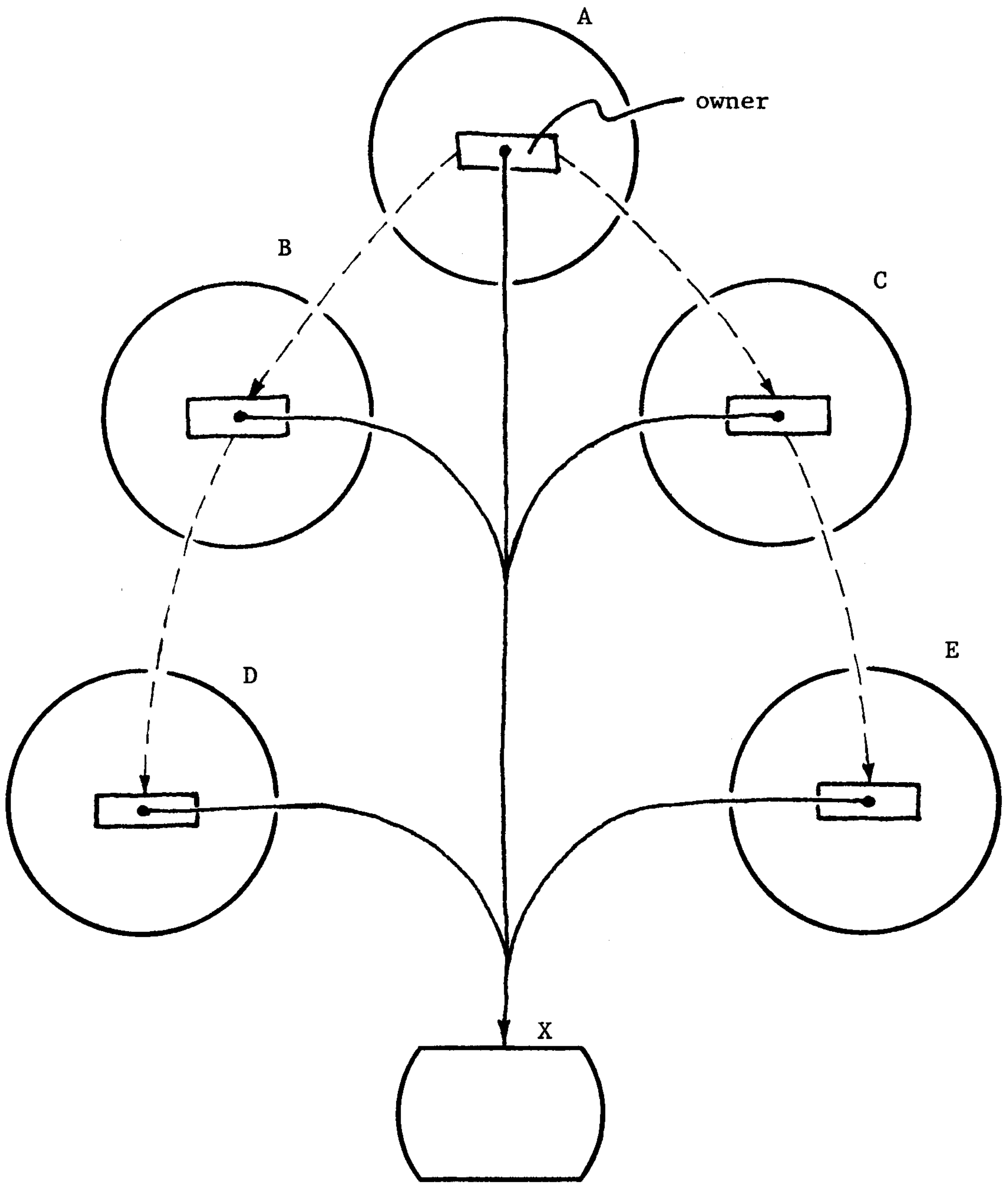


Figure 2.3-4: Ownership

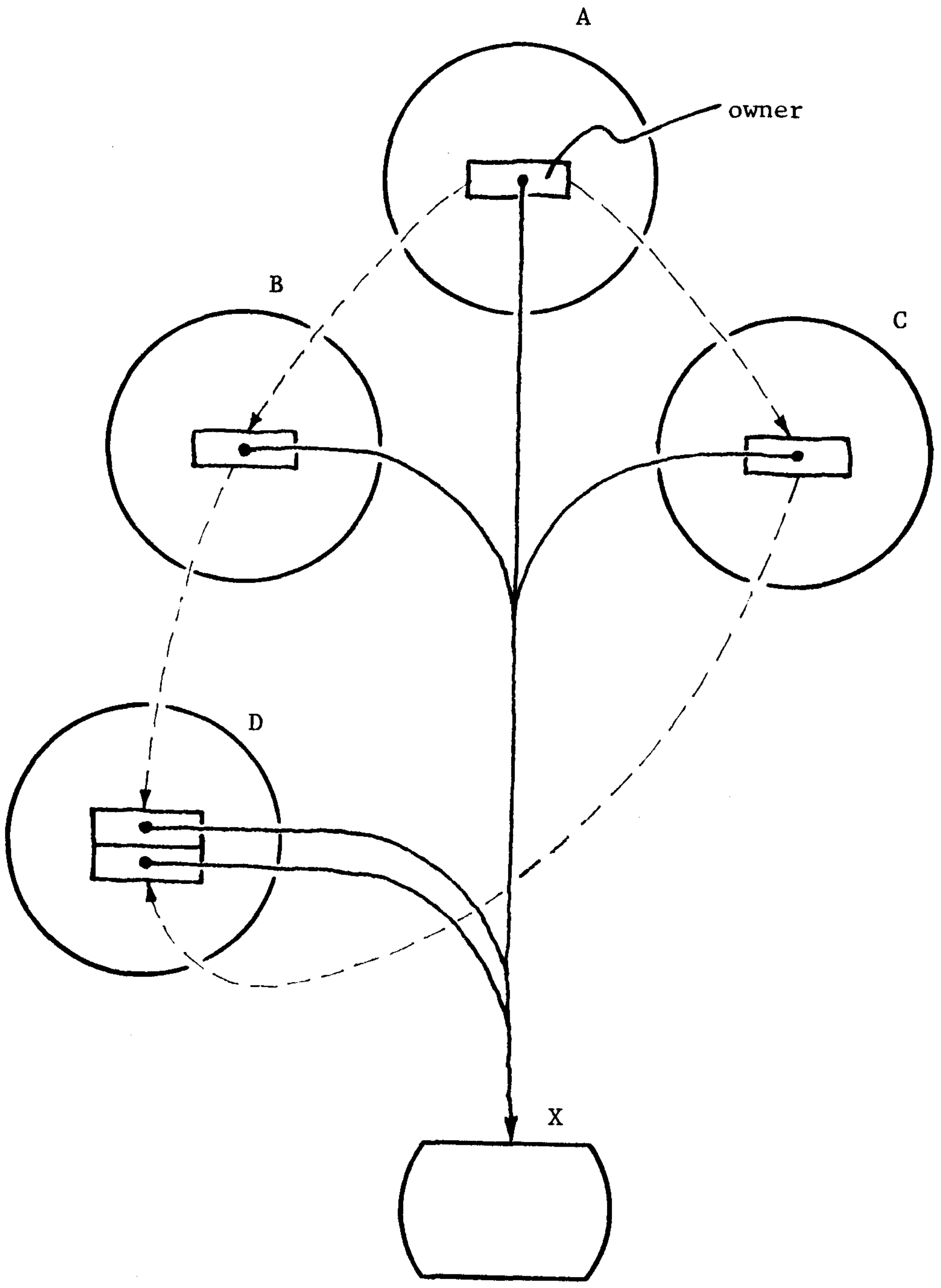


Figure 2.3-5: Multiple sources of capabilities

received capabilities for X from both B and C. Ideally, revocation of B's privileges should affect the capability which D received from B, but not the one received from C. Such distinctions clearly cannot be expressed in a domain list, and require of A a completely unreasonable amount of knowledge of the internal structure of other domains.

Yet another fundamental problem involves the authorization of revocation by domains other than the original owner. In Figure 2.3-4, for example, B stands in much the same relationship to D as A does to B, hence it would seem reasonable to allow B to revoke the privileges it granted to D. Since ownership is a normal privilege, A could authorize this by simply including 'ownership' among B's privileges, but this clearly gives B too much power (e.g. the ability to interfere with C and E). Similarly, in Figure 2.3-5, B should be authorized to revoke the privileges of the capability it has passed to D, but not the one D has received from C.

Thus, the privilege of ownership, while sufficient to authorize the total revocation of all capabilities for an object, is insufficient to deal with more general situations.

Indirection: Most of the problems with revocation in capability systems seem to be caused by the propagation of capabilities throughout the system. This suggests that domain A in our example should never give to B a capability for X whose privileges it may subsequently wish to revoke, but should retain the capability and give B a "pointer" to it. The success of this approach is very sensitive to the exact nature of the "pointer."

From domain A's point of view, the most obvious kind of pointer to the capability is simply its address in A's address space, but this address by itself is meaningless to B. To use the address, B needs to specify that it should be interpreted relative to A's address space, an action which clearly requires authorization in the form of a capability for A (or for A's implicit segment) allowing capabilities in A's address space to be exercised, but not fetched or stored. Giving such a capability to B clearly compromises A, however, since B may use it not only in conjunction with the pointer provided by A, but also with any other pointer B may invent. Moreover, this scheme also causes problems for B, since instead of a single capability for X, a capability for A and a pointer must be used. Thus, B effectively receives the absolute address $<C_{I_A}, A_X>$ where A_X is the multi-level address of X in A's address space. These problems can be reduced somewhat by the obvious expedient of always passing the simple absolute address <C,d> of A's capability for X, thus limiting A's vulnerability to a single segment, and guaranteeing that the pointer which B must handle will always be a simple displacement. Moreover, if this simple absolute address can itself somehow be squeezed into a single capability, both problems have been solved, since only the single "slot" in A's address space which contains the capability for X is usable by B, who need only keep track of the slot capability, rather than a capability and a pointer. Of course, care must still be taken to allow B to ignore the difference between a slot capability and a capability for the desired object.

Even ignoring the problem of squeezing so much information into a single capability, there are still restrictions on the use of indirection through capability slots. The problem is that such slots can never be reused. For example, suppose that A passes to B a capability for the slot containing one of A's own capabilities for X, as shown in Figure 2.3-6. If A later decides to revoke all of B's privileges to access X by erasing the capability from the slot, B still retains its slot capability. Therefore, A must be very careful never to place another capability in that slot.

One way of attacking the non-reusability problem is to squeeze still more information into the slot capability, namely the ID of X, and to check on each access that this ID matches the one in the slot. This eases the restriction somewhat: a slot may be used any number of times, but only once for any given object. Complete reusability of slots requires the inclusion of a "slot ID" in both the slot capability and the capability in the slot, to be compared on each access. This essentially amounts to re-invention of the unique ID mechanism of the base-level system, and is likely to be very cumbersome, for both user and implementor.

The non-reusability of slots in the indirection scheme is not really a fatal flaw. It simply forces the mechanism to be used in a rather stylized way. For example, domain A, rather than giving B a capability for some location in its own data structures containing a capability for X, must copy the capability for X to some spot which will never be used for anything except indirection via B's slot capability. Actually, A would

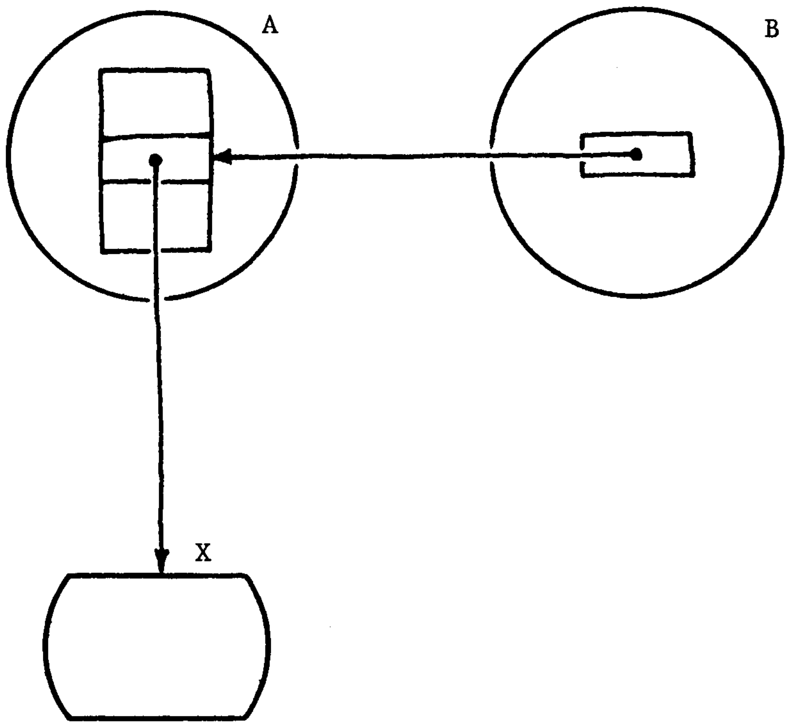


Figure 2.3-6: Indirection through a "slot"

undoubtedly have made an extra copy for B's use in any case, so that subsequent revocation of B's privileges would not interfere with A's own accessing of X. Thus, the only real burden on A is the careful allocation of slots so that they will never be reused. One approach would be to set aside one segment of A's address space and allocate slots in it sequentially. A much more attractive, if rather more expensive, scheme is the creation of a tiny new segment to hold each slot. This not only takes advantage of the base-level allocation machinery, but also implies that the displacement which we squeezed into the slot capability is always zero, and hence may be omitted.

Privilege revocation by indirection through such "link" segments is actually a fairly attractive scheme, which we pursue in some detail in the next section. It is conceptually related to both the caretaker and control schemes discussed above. If one thinks of the link segments as domains, in the passive sense, then indirection through such a link domain is much like calling a simple caretaker which merely exercises its capability on demand. (Note, however, that the cost of an actual domain-call has been avoided.) On the other hand, from the point of view of its creator, this passive caretaker is a very well-behaved controlled domain, since there is no possibility of its capability being copied or moved.

2.4 Indirection Through Link Segments

Since indirection through link segments created especially for that purpose seems to provide many desirable features for revocation, we now pursue this approach somewhat more vigorously. The discussion is still in terms of TCS, in the sense that we attempt to minimize modifications to the base-level system and construct the revocation machinery "on top of" that system. Although we will later argue that a fairly complex revocation facility should instead be included in the base-level system, it is useful to explore this higher level implementation as a first step.

As mentioned during the discussion of ownership, it is desirable for any possessor of a capability to be able to distribute copies of it while retaining the power to revoke the privileges thus conferred. Thus, if access privileges pass through the hands of several distributors, the corresponding link segments form a chain. Capabilities accessing via that chain are subject to revocation by any of the distributors. Any possessor of such a capability may extend the chain by creating a link segment and storing the capability in it. Retaining a powerful capability for the link segment allows later reduction of the privileges in the capability stored there. If and when all privileges are to be revoked, the link segment can be destroyed.

Thus far, we have made no changes at all to the TCS base-level capability mechanism, but neither have we provided any way for the indirection chains to be used to access the target object. This will require a fairly simple modification of the base-level system, but before describing that modification, it is instructive to observe precisely what goes wrong in attempting to do without it.

In terms of our standard example of A giving B privileges to access X, we find that A, in Figure 2.4-1, having created link segment S_L and stored its capability C_x for X there, must now give to B a capability C_L for S_L. Clearly, B's capability C_L must not allow B to tamper with the capability in S_L, but only to use it as a component of a multi-level address for X. (For example, if X is a segment, B's address for its 5th word, given that C_L is located at location 3 of B's implicit segment S_I, is

$$3|0|5 \equiv <((C_I,3),0),5> \equiv <(C_L,0),5> \equiv <C_x,5> \quad .)$$

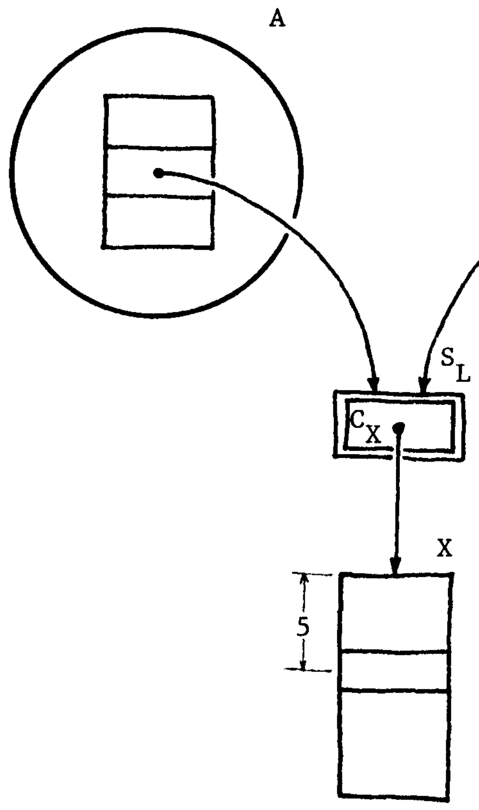
There are four interdependent problems with this attempt to implement link segments on an unmodified capability system:

1) Non-transparency: A domain accessing an object must know how many links are present in the chain leading from its capability to the object (i.e. how many 0's to insert in its multi-level address, as in "3|0|5" above).

2) Ambiguity: A link in the chain is indistinguishable from a target object which happens to be a segment containing a capability in location 0.

3) Subvertability: This is really implied by problems (1) and (2); if the accessing domain accidentally or maliciously specifies a multi-level address which is too short, it can obtain a copy of a capability stored in the chain, thus circumventing subsequent revocation.

4) Loss of selective adjustment in long chains: Only the last link in the chain contains a capability whose



Figures 2.4-1: Example of indirection through a "link" segment

privileges apply to the target object. Each earlier link contains a capability whose privileges apply to the next link in the chain. The only revocation allowed by such a link is total revocation by breaking the chain.

All of these difficulties are avoided by a simple modification to the base-level system, which introduces a new operation on capabilities, and changes the behavior of the base-level system slightly when a capability is encountered to which this operation has been applied.

The new operation allows a capability of type 'segment' to be converted into a capability of type 'indirect' in which all privileges are 'on.' (As we shall see later, this is just a specific instance of a more general mechanism useful for type extension.) The intention is that such indirect capabilities for link segments should be handed out to domains which are being given revocable privileges. For example, in Figure 2.4-1, the capability C_L which A gives to B must be of type 'indirect,' although A's own capability for S_L is of type 'segment.'

Whenever an operation which expects a capability for some object encounters instead a capability of type 'indirect,' the indirect capability is followed; that is, it is replaced by a copy of the capability in (location 0 of) the segment to which it points, with any privileges deleted which did not also occur in the original indirect capability. This step is iterated, as necessary, until the resultant capability is not of type 'indirect,' at which point the operation proceeds as usual.

Thus, each time an object is accessed via a chain of link segments, that chain is automatically followed to the target object unambiguously indicated by the first non-indirect capability encountered. The resultant capability is exercised, but is not otherwise available to the accessing domain, hence the chain cannot be circumvented. The privileges conferred are the intersection of those found during the entire scan of the chain, thus allowing independent revocation by each intermediary domain controlling a link in the chain. In other words, problems (1) through (4) above have been avoided.

It is important to note that an indirect capability is followed _only_ when it is used to access its target object; following is _not_ performed when the capability itself is manipulated (e.g. by the copy or reduce operations).

The indirection feature being described is fundamentally different, not only in design, but in intention, from the multi-level addressing feature of TCS. In some systems, such addressing has also been referred to as "capability indirection." A system in which both of these features were desired would require two separate mechanisms.

Distribution of revocable capabilities using this scheme involves five steps:

1. Creation of a link segment.
2. Conversion of a capability for that segment into an indirect capability.
3. Copying of the distributor's own powerful capability for the object into the link.

4. Reduction of the privileges of the capability in the link to an appropriate level.

5. Distribution to the receiving domain(s) of copies of the indirect capability produced in step 2.

Any later reduction in level of trust can be enforced by re-executing step 4, specifying some reduced set of privileges.

Although this indirection scheme does a reasonable job of capturing the notion that a distributor of a capability should retain the power to revoke the privileges it confers, it gives one the feeling that the desired mechanism is being "simulated," in the sense that the basic action of distributing a capability is provided by a particular non-atomic sequence of operations, rather than being an atomic operation. This has two consequences:

a) It is inconvenient for the user.

b) It may allow other sequences of operations to produce a non-meaningful state.

The former problem can be easily dealt with by providing a simple library procedure to perform the actions required for capability distribution. The latter problem, however, is not so easily disposed of. Suppose, for example, that by accident or design, a domain, in performing step 3 of the procedure, stores not the appropriate object capability, but the indirect capability created in step 2. This is just one way in which circular indirection chains can be created. Such chains, when followed, will cause an endless loop in the base-level system. Of course, one could deal with such a situation by placing an arbitrary limit on the length of an indirection chain to be followed before it is abandoned and an error is signalled, but this is rather ad hoc and inelegant. An atomic operation producing only well formed chains would be much more attractive.

Another problem with this scheme is its relative inefficiency. For one thing, it would generate large numbers of small segments. This could be extremely costly in terms of both space and time, especially in a system using block-oriented rotating magnetic storage and a corresponding paged primary memory. For another thing, the scheme requires the following of a chain of links each time an indirect capability is exercised. This overhead could prove prohibitive, particularly in the case of indirect access to segments. Moreover, any mechanism attempting to capture a computation's set of recently used chains and retain them in fast hardware would be complicated by the fact that every store instruction would have to be regarded as potentially invalidating this "lookback" information by overwriting a link in some chain.

By comparison, if equivalent revocation features were built into the base-level system, they would probably be easier to use, harder to misuse, and more amenable to optimization. This approach is explored in detail in Chapter 3.

2.5 Type Extension

The definition of a large complex system as a sequence of "layers" has been found to be a valuable technique, aiding all stages of design, implementation, testing, and documentation [Di 68b, Pa 72, La 69]. In an object-oriented system, this implies that not all of the various types of objects provided will be implemented, or even known about, by the base-level system. On the other hand, it would be most inconvenient if the naming and protection machinery provided by the base-level system (i.e. capabilities) had to be reinvented by each new layer of the system; this would not only raise serious problems for the implementation, but would also force the users to interface with several parallel mechanisms for storing privileges, passing privileges to other domains, and so on. It is therefore very desirable for the base-level capability machinery to provide capabilities for objects of which the base-level system has no knowledge.

The various base-level facilities involving capabilities can be divided into two categories. In the first category are the facilities involving capabilities themselves: their creation, integrity while stored, copying, erasure, and so on. In the second are the facilities for manipulating base-level objects named by capabilities: fetching from a segment or calling a domain, for example. It is the facilities in the first category which can and should be provided for higher-level objects unknown to the base-level system.

As indicated in section 2.1, a capability provided by TCS contains the _type_ of its corresponding object. The division of the set of all objects into types is a well known and intuitive idea (although, as pointed out by Morris [Mo 72], the difference between the type of an object and the privileges allowing access to it is somewhat indistinct). The set of objects provided by the base-level system falls into some small fixed number of types. The question is: what type of capability is used to name a higher-level ("extended") object? Various answers have been proposed, four of which we will explore.

Approach 1: Representation capabilities. Any given layer of the system runs in an environment provided by the lower layers, hence any object it defines must be represented in terms of lower level objects. We will assume that the representation of each extended object is a single lower level object, since that single object can be a segment containing capabilities for any other objects which are necessary. Thus the most obvious candidate for the capability for an extended object is simply a capability for the representing object. A possessor of that capability could call the layer implementing that extended type to request some operation, and pass the capability to indicate the extended object to which the operation should be applied. Having been passed this capability, the domain implementing the extended operation would automatically have access to the representation of the object.

There are at least three problems with this approach. The first and most important concerns the selection of an appropriate set of privileges to appear in the capability. The difficulty is that the domain implementing the extended object requires essentially complete power to manipulate the representation, while wishing to deny such power to the using domain(s) in order to prevent tampering with the representation. If the same capability is used by both, this is clearly not possible. Hence, the implementing domain, having upon request, created the representation of a new extended object, and thus obtained a fully privileged capability for that representation, must appropriately weaken that capability before returning it to the calling user domain. However, in order to guarantee its own future access to the representation, the implementing domain must do one of two things. Either it must save a copy of the original fully privileged capability for later use, or it must make arrangements allowing it to convert the weaker capability back into the fully privileged one when it later receives it as a parameter to some operation on the extended object.

The first method obliges the implementing domain to maintain a global table containing privileged capabilities for all existing extended objects which its layer has created, and to locate the corresponding entry whenever it receives a weak user capability. This method is reasonable, if somewhat clumsy.

The second method requires some facility similar to Jones' "amplication" [Jo 73], allowing the implementing domain to add specified privileges to capabilities of the type of the representing object. Clearly, the power to amplify capabilities of a given type is a very dangerous power, and must be tightly controlled, since it can completely subvert the inter-user protection of objects of that type if misused. While this is an incomplete subversion of the objects in question, in the sense that they still follow the semantic rules which define their type, it must be regarded as a failure of the corresponding layer, since the correct functioning of a layer includes the protection of its users from each other. Thus, the authorization of amplication must be the responsibility of the layer implementing the type whose capabilities are being amplified. One of the main criteria of layering, however, is that a given layer should have no knowledge of higher layers. Thus, it is not possible for a layer to distinguish between "legitimate" higher layers which need amplification, and untrustworthy domains which would use amplification to gain undesired access to other domains' objects. We thus conclude that privilege amplification by itself is insufficient to solve the problem of assigning appropriate privileges to the using and implementing domains of an extended object, given that the same type of capability is used by both domains. (In conjunction with another complementary mechanism ("constituent rights" [Jo 73]), however, amplification can provide a very powerful type extension facility which is equivalent to one which we will describe later.)

The second problem with the representation-capability approach involves the control of access to the extended object, as opposed to its representation. Privileges are needed in each capability to specify which of the operations on the extended type are authorized to possessors of that capability. This certainly cannot be done by assigning new meanings to the existing privileges, since granting the use of some operation on the extended object would then imply granting some unrelated access to the representation. Hence, multiple sets of privileges are needed. On the one hand, this tends to make capabilities undesirably large. On the other hand, the number of sets of privileges provided places a fixed upper bound on the number of times a base level type can be extended. This situation is especially frustrating since in most capabilities, only one of the sets of privileges will be non-empty.

The third problem with the representation-capability approach is the difficulty of determining, given some capability, the type of the corresponding object. This is caused by the "unofficial" status of extended types in this approach. A given base-level object may have been extended one or more times, but the type fields of all capabilities for it still contain its base-level type. The only indication that the capability is of a given extended type is the presence of a matching fully privileged capability in the previously mentioned table kept by the domain implementing that extended type. Thus, one is not able to ask of a given capability "what is its type?" but only "is it of type T?" for some list of types T. This is a clumsy and costly substitute.

Approach 2: Domain capabilities. This approach is, in some sense, a variant of the previous approach, in which the representation of each extended object is a domain. A using domain has only one privilege in its capability for this representation domain: the privilege of calling it. To perform an extended operation, the user performs such a call, indicating only the operation to be performed; the object to which the operation applies is implicit in the identity of the called domain. Actually, this approach falls outside the framework of our discussion, since it requires independent domains callable by any process (at least if extended objects are to be shared). It deserves mention, however, since it has been used in at least two systems [En 72, Fa 68], and because it attacks the three problems of the representation-capability approach, with somewhat mixed results.

The first problem, that of easily allowing only the imple-plementing domain full access to the object's representation, is bypassed, since each object has, in effect, its own copy of that domain, which can retain a privileged capability for the rest of the representation in some convenient location in its address space.

The second problem, that of controlling access to the extended object, is solved by embedding in the domain information about the operations it is willing to perform. Thus, privileges for extended objects are represented and controlled differently for base-level and extended objects; whenever a less privileged capability for an extended object is desired, a copy of the domain can be made, which is then ordered never to perform the operations being denied to receivers of the less privileged capabilities. This is not as expensive a solution as it might appear, for two reasons. First, the various copies of the domain representing a given extended object can retain in their implicit segments the information specifying the operations they are willing to perform, and can thus share all the other identical components of their address spaces. Second, the capabilities for a given object exhibit a strong tendency to fall into a small number of subsets, each containing capabilities with identical privileges (a tendency which we shall exploit later). Thus, the number of copies of the domain representing a given object tends to be much smaller than the number of capabilities for the object.

The third problem, that of determining the type of a given object, is handled in an interesting if somewhat clumsy way. Clearly, examination of the capability will always indicate the type to be 'domain.' One can establish a uniform convention, however, for associating some arbitrarily chosen unique capability with each extended type, and storing a copy of that capability in some standardized location in each domain (e.g. location 0 of its implicit segment) representing an object of that type. If users are allowed to examine that location, they can then reliably determine the type of each extended object. The main objection to this scheme is that base-level types and extended types are represented differently, which disallows any uniform type-checking mechanism.

There are some other problems peculiar to the domain-capability scheme. Two difficulties arise from the fact that the domains implementing the extended type are associated with the objects of that type, rather than with the accessing processes. One reason for wanting to associate a domain with each process as the "representative" of a given layer is that the local storage of the domain provides a natural repository for information describing the status of that process from the point of view of that layer. This "own" storage is not provided by a scheme which associates domains with extended objects instead of processes [Fa 74]. Some systems have made heavy use of such own storage (e.g. CAL-TSS, Multics); it is not clear to what extent this is intrinsically necessary.

Another minor difficulty with the domain-capability approach is its implicit assumption that all operations on extended objects are monadic. While this is undoubtedly the most common case, examples abound of useful operations which apply to two or more objects ("file-to-file copy"), to some large implicitly defined set of objects ("close all open files") or even to no object at all ("create a file"). Forcing such operations into the mold of a call on a particular object is not only artificial for the user, but can be somewhat inconvenient for the implementor.

Approach 3: Sealed-data capabilities. This approach is motivated by the following observation about the use of representation capabilities in Approach 1: If the using domains are not allowed direct access to the representation of an extended object, and if the implementing domain always replaces the user's weak capability with the corresponding strong one saved in its own table, then the user's weak capability is never actually used to access the representation. This suggests the possibility of changing the type field in the user's capability to contain, not the type of the representation, but some new value associated with the type of the extended object. There are two distinct advantages to this change. On the one hand, it provides an easily visible and unforgeable (given mechanisms to be described shortly) indication of the type of the extended object. On the other hand, it renders the capability useless for directly accessing the representation, thus eliminating the need for a separate set of privileges to control such access, as was required in the representation-capability approach.

From the implementing domain's viewpoint, the creation of a new extended object using this approach could be done by:

1) creating a representation of the object
2) saving a fully privileged capability for the representation in a hash table keyed on IDs
3) constructing a new capability containing the extended type, full privileges, and the ID of the representation,

and returning it to the caller.

When called to perform some operation, the implementing domain can examine the passed capability:

1) checking the type to verify that the object is one that it implements

2) checking the privileges to verify that the requested operation is authorized

3) locating the representation capability in its table and performing the operation on the representation

Clearly, the creation of capabilities for extended objects must be carefully controlled, since a forged capability could deceive not only the users, but also the implementing domain. The creation of capabilities of a given type can itself be authorized by a capability. When this capability and an arbitrary datum are presented to an appropriate new base-level operation, a new capability is returned with the authorized type, all privileges 'on,' and the datum as its unique ID. (As suggested above, this might be the ID of the representation, but could be any value desired by the implementing domain.) Section 2.6 will discuss how such authorizations to create new capabilities can themselves be created and distributed.

The sealed-data approach as described is a quite acceptable type extension mechanism, and has in fact been used in at least one actual system [St 73]. It places each higher layer in much the same position as the base-level system; a capability is regarded as holding an ID sealed in a tamperproof box, which guarantees that the name presented by a user is in fact a valid name given him by that layer. Furthermore, it allows this without forcing re-invention of the sealing mechanism in each new layer. It does, however, require that each new layer implement its own table for converting an ID into a capability for the representation of the corresponding object; this is a partial duplication of the function of the base-level "map" of section 2.2. It is desirable to avoid re-invention of the map, as well as of the capabilities themselves, an advantage possessed by our fourth approach to type extension.

Approach 4: Sealed capabilities. The need for each layer to maintain a table mapping extended object capabilities into representation capabilities can be eliminated if the system simply allows each extended capability to *contain* the corresponding representation capability. The extended capability thus becomes a tamperproof box holding another capability! On the surface, this makes it appear inevitable for capabilities to grow larger and larger as objects are extended repeatedly, a problem already discussed in connection with our first approach to type extension. A carefully designed implementation, however, can avoid this phenomenon, allowing unlimited extension with fixed size capabilities, as we shall see in section 2.7, which discusses the sealed-capability approach in more detail. First, however, we digress briefly to examine some more general questions about type extension.

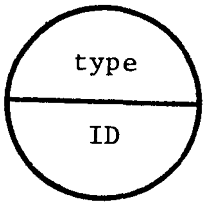
2.6 Hierarchies of Objects and Types

In a non-extendible system, only a small fixed number of predefined types are provided, hence types can be identified by small integers. In an extendible system, a much larger set of types is needed. Two conflicting considerations influence the choice of the size of this set. On the one hand, it is desirable to minimize the size of type identifiers, since these appear in capabilities, where compactness is a great virtue. On the other hand, it is desirable to maximize the total number of types available, to insure that the supply will never be exhausted, especially since type identifiers, like object IDs', can never be reused.

Emphasizing the first consideration results in a system in which the number of types, while much larger than the number which would ever be legitimately used, is still fairly modest (e.g. thousands or millions of types) [St 73]. This leaves open the possibility of a malicious program using up all available types within a few minutes of determined computing. Types in such a system must therefore be viewed as a finite resource, and must be allocated as such. This is possible, but somewhat inconvenient.

Emphasizing the provision of an inexhaustible supply of types results in a system design in which the space of type identifiers, like the space of object IDs, is effectively infinite (i.e. too large to be exhausted during the lifetime of the system). By combining these two infinite name spaces, the HYDRA system [Wu 74, Jo 73] provides an elegant conceptual framework in which types are themselves objects. This is illustrated in Figure 2.6-1, which depicts the set of all objects as forming a three-level tree. For purposes of this figure, only two attributes of each object are of interest. One is its ID. The other is its type, which is

An object:



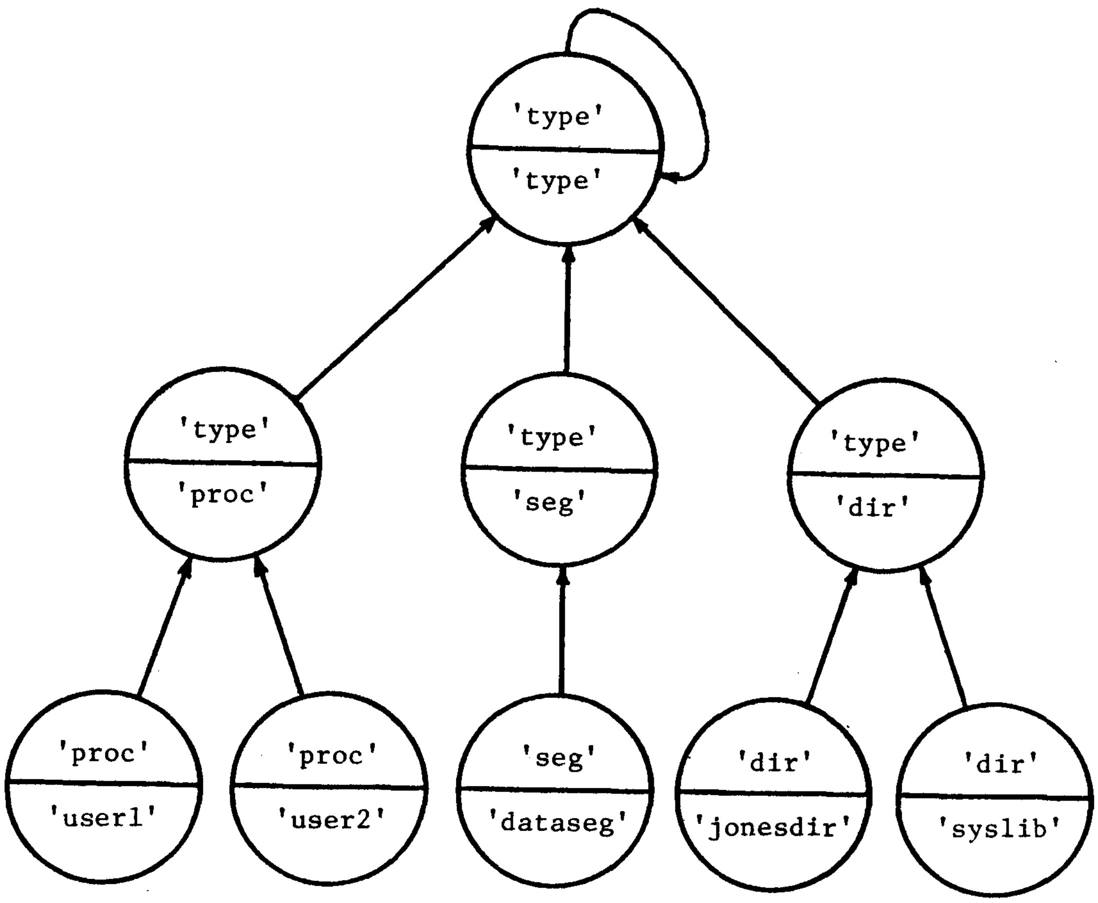


Figure 2.6-1: Three-level object hierarchy

simply the ID of some other object.* Insisting that the other object so identified be of type 'type,' and providing a special root-object with ID 'type' (which is also of type 'type') forces all objects (except the root) to occupy either the second or third level of the tree. The second level contains the types, while the third level contains the non-type objects.

Creation of objects in such a scheme can be described conceptually as a single operation:

$$C_{obj} \leftarrow \text{create\_object}\ (C_{type})$$

where the new object will be a type if C_{type} is a capability naming the root object, and a normal object if C_{type} is a capability naming a second level object. If C_{type} names a third level object, an error is signalled. In practice, of course, such a unified base-level create_object operation cannot replace the specific object-creation operations for the various extended types, since only the corresponding layer has both the authority and the knowledge needed to create and initialize the various components of the representation of a given type of extended object.

The practical disadvantage of the viewpoint just described is the large size of type IDs. Nevertheless, we adopt the HYDRA view of types as being objects. In Chapter 3, we describe a scheme which manages to adopt this point of view, and yet provides an extremely compact representation for capabilities.

There is a second kind of hierarchy among the types in an

*Unique IDs, which are simply long integers, are shown as symbols in Figure 2.6-1 for clarity.

extendible system, which has been described by Morris [Mo 72]. This second hierarchy involves only the types, rather than all the objects, and attempts to characterize the layered nature of the system. Figure 2.6-2 illustrates a simple example, in which segments are assumed to be predefined, and various plausible extended types are shown, each indicating the type of its implementation. This assumes that all objects of a given extended type have the same type of representation, which does not seem unreasonable. One can find examples, however, of situations in which differing characteristics of objects of the same extended type might make different types of representations desirable. In Figure 2.6-2, for example, one might wish to allow long documents composed of a collection of text files, which, according to our conventions, would be represented by a segment containing several text file capabilities. As another example, one might wish to represent a customer list as a sorted file or as a linked list, depending on the frequency of insertions and deletions expected. In the general case then, the types form not a simple tree, but a directed graph without cycles. The latter property expresses the partial order induced on types by the layered structure of the system. Note that for any given extended object, there is only one representing object, hence for a given representing object, the extended objects it represents can form at most a tree. (Of course, in any realistic situation, this tree is only a linear chain.)

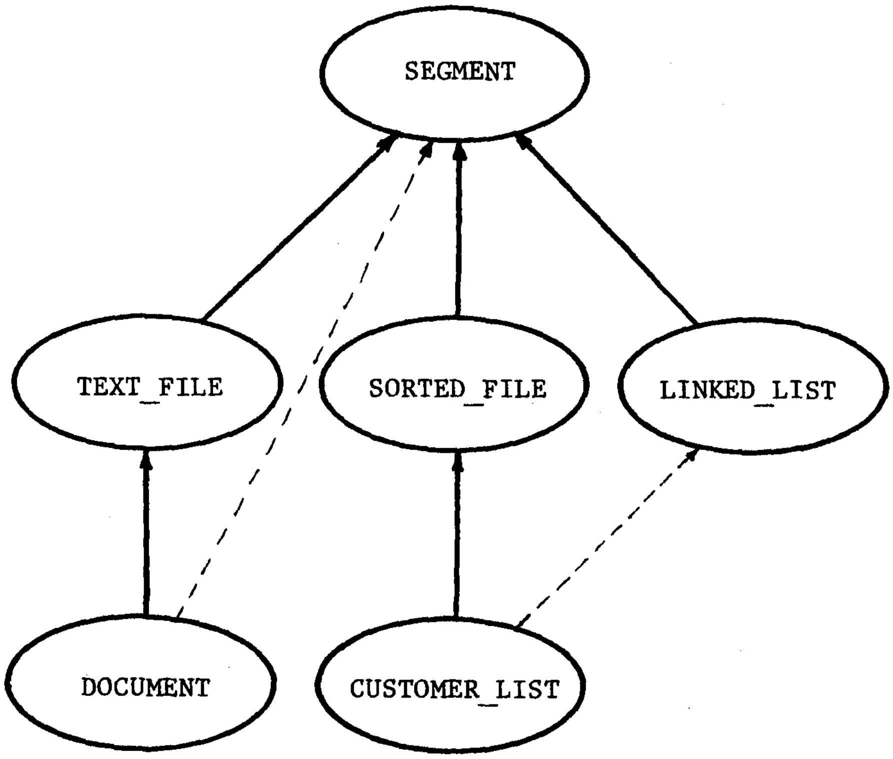


Figure 2.6-2: A type tree

2.7 Type Extension Using Sealed Capabilities

We now return to the last of our four approaches to the naming of extended objects, that using "sealed capabilities." As in the sealed-data approach, the manufacture of extended capabilities must be carefully controlled to prevent forgery. Given the view that types are objects, the appropriate authorization to manufacture a capability of a given type is a capability for that type. A layer can obtain a new type T by executing

$$C_T \leftarrow \text{create\_type ( )} \quad .$$

Subsequently, it can seal any capability C by executing

$$C_s \leftarrow \text{seal } (C,C_T)$$

as illustrated in Figure 2.7-1. C_s will have type T, all privileges on, and a new unique ID assigned by the system.

Later, C can be recovered by executing

$$C \leftarrow \text{unseal } (C_s,C_T) \quad .$$

Note that C_T must be presented to authorize unsealing, thus preventing any random possessor of C_s from obtaining the capability C which is sealed inside.

The implementation of capability sealing as just described requires a fair amount of machinery, such as that to be described in Chapter 3. However, a slightly restricted version of capability sealing can be added to TCS in a surprisingly simple way. In the description below, we assume that a layer wishes to implement

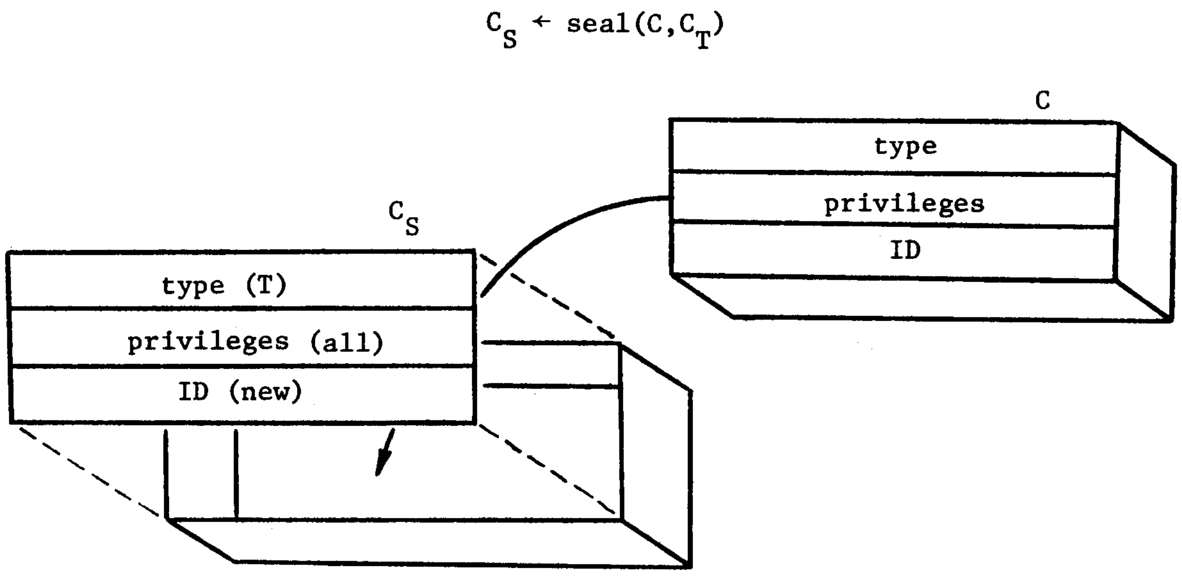


Figure 2.7-1: Sealing a capability

extended objects of type T_x whose representations are of type T_r.

The creation of type T_x is performed by the operation:

$$C_{T_x} \leftarrow \text{create\_type}\ (T_r, P_r) \quad .$$

Note that the type of the representation (T_r) must be specified. This is one of the restrictions necessary for the implementation described below, and forces the set of types to form a tree, as discussed in the previous section. Also, a set of privileges (P_r) must be specified, whose significance will be explained below. The resulting capability for the new type (C_{T_x}) allows the creation of new capabilities of type T_x, containing representation capabilities of type T_r sealed inside.

The creation of an extended object involves the creation of its representation (which results in a capability C_r), followed by the creation of a capability C_x for the extended object, using the operation:

$$C_x \leftarrow \text{seal}\ (C_r, C_{T_x}) \quad .$$

This produces a sealed capability C_x. The second restriction in the scheme is the requirement that C_r contain at least the privileges in P_r. (In practice, this is no problem, since sealing is generally preceded by the creation of the representation, which produces a fully privileged capability C_r.)

Later, whenever the implementing domain receives as a parameter a capability C_x of the new type, it can recover the sealed capability C_r using the operation:

$$C_r \leftarrow \text{unseal } (C_x, C_{T_x}) \quad .$$

Note that the recovered capability C_r has exactly the privileges P_r, which cannot be greater than the privileges in the capability originally sealed. Thus, the layer which implements the representing type need not trust the layer implementing the extension, since the latter can only recover privileges which it had previously.

The scheme just described can be implemented by representing the extended type as shown in Figure 2.7-2. The implementation of sealing now consists of merely changing the type field of C_r from T_r to T_x and turning on all privileges to produce C_x, while unsealing simply changes it back and sets the privileges to P_r, thus recreating C_r again. Note that C_x will thus contain the same object ID as did C_r, rather than a new ID provided by the system. In practice this is not a serious problem.

This implementation clearly allows a given object to be extended one or more times, and still be represented by a standard-sized capability. Variations on this scheme which depend on short type IDs are described by Sturgis [St 73] and Lindsay [Li 73]. Another related scheme is the "constituent rights" approach discussed by Jones [Jo 73], which is essentially equivalent to sealing a segment containing several capabilities. Chapter 3 will describe a scheme which eliminates the restrictions described above, allowing arbitrary sealing of capabilities.

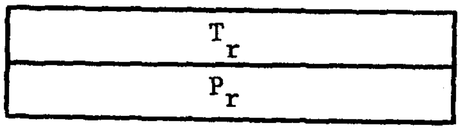


Figure 2.7-2: Representation of a type

2.8 Goals for a New Capability System

This chapter has attempted to set the stage for the proposed capability mechanism of Chapter 3 by sketching a typical capability system, exploring the problems of revocation and type extension in the context of that system, and discussing various relatively minor modifications to such a system attempting to solve those problems. In disucssing these modifications separately, examining both their strengths and their weaknesses, a number of desirable properties have been noted. These are listed below, and are adopted as the goals to be met by the design proposed in Chapter 3.

Goals

1) Revocation should take effect immediately.
2) It should be possible to revoke the various privileges in a capability independently.
3) It should be possible to selectively revoke the privileges of a subset of the capabilities for an object, and this should require no global knowledge of capability propagation.
4) Any distributor of a capability (i.e. not just the "owner" of the object) should be able to revoke its privileges.
5) The users of capabilities should not need to distinguish between revocable and non-revocable capabilities.
6) The cost of revocability should not be excessive.
7) The mechanisms of revocation and type extension should interact correctly.

Chapter 3

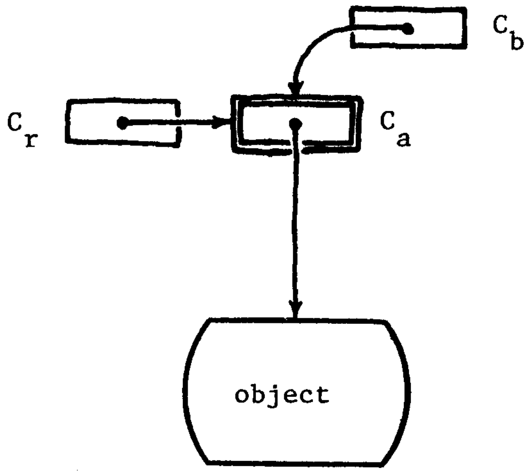
A New Capability System

3.1 A New Capability System

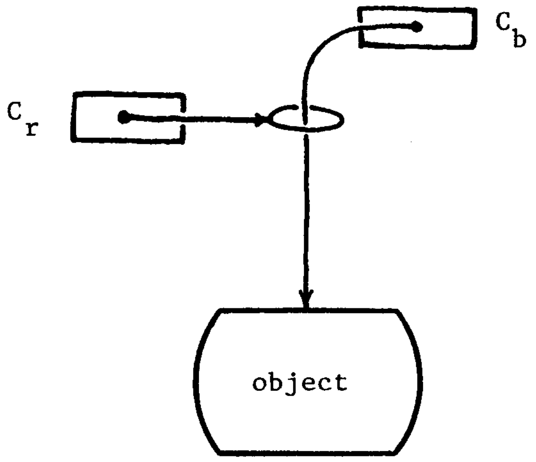
The goal of this chapter is the description of a new capability system (called NCS for short) which meets all of the goals listed at the end of Chapter 2. This requires a fairly substantial departure from the TCS system of Chapter 2. After discussing two abstractions of the "link segment" scheme of Chapter 2, we adopt the family tree model to describe the revocation behavior of capabilities. The mechanism of generalized sealing is then proposed, to provide both revocation and type extension, and the practicality of implementing the scheme is argued in some detail.

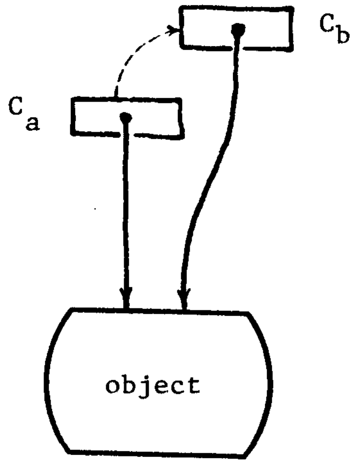
3.2 Design Considerations for Revocation

In the design of the NCS capability scheme presented in this chapter, we wish to retain as many as possible of the advantages of the indirection scheme of Chapter 2, while avoiding its problems. There are at least two approaches which can be taken in attempting to capture the essence of the indirection scheme in a base-level construct, as depicted in Figure 3.2-1a. On the one hand, as in Figure 3.2-1b, one can regard C_a as being merely a part of the mapping from C_b to the object, and C_r as being a special <u>revoker</u> capability which allows that mapping to be broken. On the other hand, as in Figure 3.2-1c, one can regard both C_a and C_b as being capabilities for the object, with C_b being somehow <u>dependent</u> on C_a in the sense that revoking C_a



(a) Indirection scheme





(b) Revoker-capability approach

(c) Dependent-capability approach

Figure 3.2-1

automatically revokes C_b as well.

Taking the former point of view results in a scheme in which the mapping from a capability to an object is itself viewed as being essentially like an object, since one can have a capability for it and thus be authorized to manipulate it. To allow individual privileges to be revoked independently, one must define the mapping as containing, or at least limiting, the privileges of the capability. The establishing of one's future power to revoke a capability should be an atomic operation, as discussed in Section 2.4. For example, the situation in Figure 3.2-1b can be produced by executing

$$C_r \leftarrow \text{revoker } (C_b) \quad .$$

Subsequently, the possessor of C_r can revoke the privileges in C_b by executing

$$\text{revoke } (C_r,P) \quad .$$

In its effect on C_b, this is equivalent to the TCS operation

$$\text{reduce } (C_b,P) \quad .$$

The difference lies in the fact that, unlike reduction, revocation also takes effect in any and all copies of C_b which may exist. The interaction of revocation with copying is clarified in Figure 3.2-2, which shows the situation resulting from executing

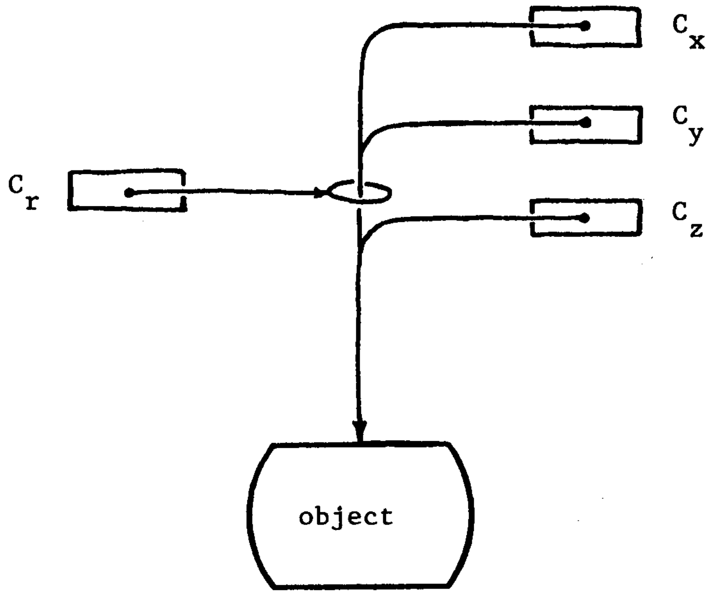


Figure 3.2-2: Interactions of copying
and revoker capabilities

$$C_y \leftarrow C_z$$

$$C_r \leftarrow \text{revoker } (C_y)$$

$$C_x \leftarrow C_y$$

This kind of interaction causes subsequent revocation of C_y to affect C_x but not C_z, which is clearly the desired behavior.

More complicated situations include "subletting," as shown in Figure 3.2-3, in which both the original owner (holding C_o) and an intermediate distributor (holding C_d) retain the power of revocation over the user (holding C_u), and "bill collecting," as shown in Figure 3.2-4, in which the ability to revoke the access of the user (holding C_u) is delegated to a "collection agency" domain, with the owner (holding C_o) retaining the option of later disabling the collection agency if the contract with the user is renegotiated. Note that the latter example takes advantage of the fact that revocability, being authorized by a capability, is itself thus revocable.

The revoker-capability approach just described has a good deal to recommend it, and has in fact been explored in some detail in a system design project at Stanford Research Institute [Neu 74]. However, we pursue here the dependent-capability approach instead. Investigation of the two approaches reveals the following advantages of this choice:

a) It avoids the introduction of special capabilities authorizing revocation, thus simplifying matters somewhat (although a certain amount of complication is unavoidable, as we shall see shortly).

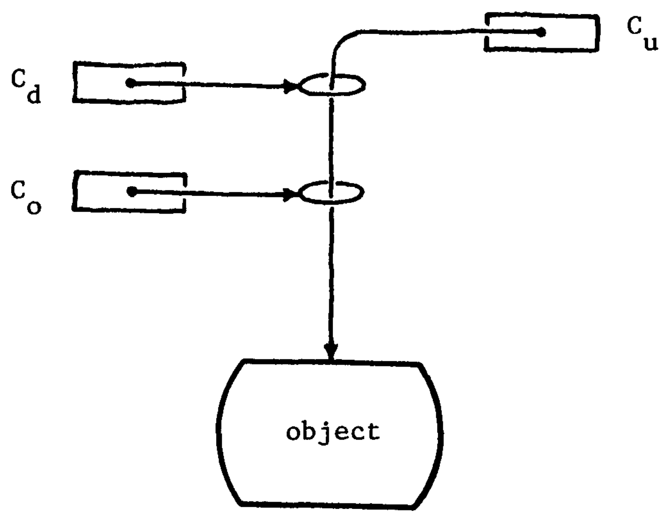


Figure 3.2-3: Subletting using revoker capabilities

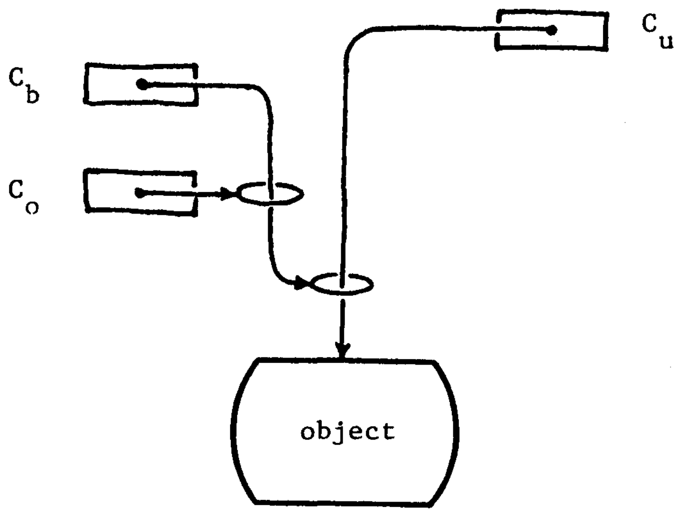


Figure 3.2-4: Bill-collecting using revoker capabilities

b) It avoids treating the capability-to-object mapping as a manipulable object, which significantly reduces implementation costs, but sacrifices the ability to make revocability itself revocable.

c) It can be cast in terms of a mechanism (to be described in Section 3.4) which unifies the notions of revocation and type extension.

It must be admitted that the choice is not entirely clear-cut; in particular, the opposite conclusion might be reached in a context in which revocable revocability was considered important.

One motivation for the notion of dependent capabilities is the observation that a weakened copy of a particular capability can arrive in the possession a domain as a result of either of the following sequences of actions:

a) The privileges in the original capability are reduced to the desired set, and then a copy is passed to the receiving domain.

b) A copy is passed to the receiving domain, and then the extra privileges are revoked from the original.

The essence of sequence (b) is that the granting domain "has second thoughts" and wishes it had used sequence (a) instead. This suggests defining the revoke operation by simply changing the reduce operation to be commutative with copying, in the sense that

$$\text{revoke } (C_a, P);\ C_b \leftarrow C_a$$

and

$$C_b \leftarrow C_a;\ \text{revoke } (C_a, P)$$

produce the same net effect. Of course, revocation cannot be expected to undo any intervening exercise of the affected capabilities hence this commutativity applies only to the state of the protection structures, rather than to the state of the objects being protected. Nevertheless, it is an attractive way of describing the effect of revocation.

Exactly how the revoke operation manages to find all outstanding copies of the capability being revoked is, of course, the central implementation question concerning this scheme. At this level of discussion, however, we simply imagine that a global search is done to locate and revoke the appropriate capabilities.

Given that we require commutativity of copying and revocation there are several possible schemes, corresponding to different assignments of dependency among the various capabilities existing for a given object. Clearly, the commutativity requirement constrains the choice to assignments in which the dependency set of any given capability includes all other capabilities which have been derived from it through one or more levels of copying. We examine three schemes, corresponding to three such assignments.

Scheme 1: The simplest scheme considers all capabilities for a given object to be interdependent, so that revoking privileges from any of the capabilities affects them all. This approach is clearly unsatisfactory in general, for two reasons:

a) All capabilities for a given object are forced to contain the same set of privileges.

b) Any domain possessing a privilege can revoke it from all other domains.

Nevertheless, this approach has one virtue which makes it worth mentioning: it is possible to copy a capability and have the copy retain the revocation powers of the original. This is desirable, for example, when a domain simply wishes to move a capability within its address space.

Scheme 2: A more appealing scheme considers the capabilities for a given object as forming a "family tree" generated by the copy operation as follows:

a) The initial capability (produced at object creation time) occupies the root node of the tree.

b) Whenever an existing capability is copied, the copy occupies a new son node of the node containing the capability being copied.

A typical family tree is shown in Figure 3.2-5. By defining a capability to be dependent on each of its ancestors in the family tree, we maintain at all times the condition that no capability can have any privilege not possessed by all of its ancestors. Thus, revocation affects entire subtrees of the family tree.

This tree-structured dependency solves the two problems encountered with version 1 above, since it allows different capabilities to contain different sets of privileges, and strictly circumscribes the effect of revoking privileges from any given capability. Thus domain A may pass capabilities to domains B and C, such that

a) B and C have different privileges from each other, and from A,

b) A may revoke the privileges of B and C independently,

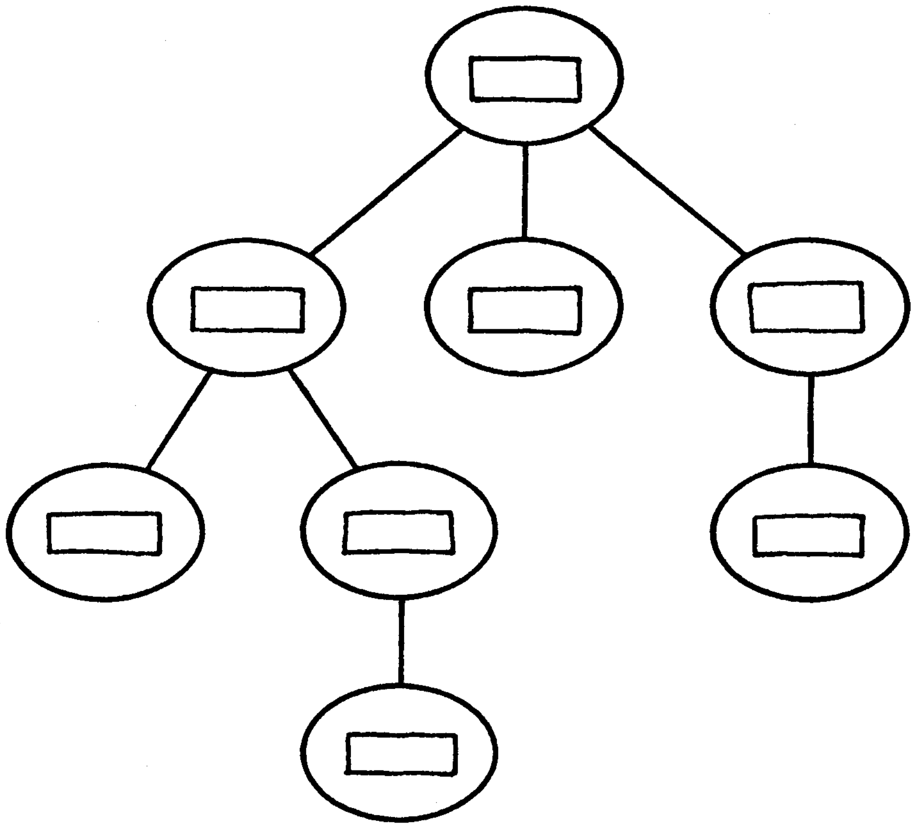


Figure 3.2-5: A typical family tree of capabilities

and c) B and C may not interfere with each other, nor with A, by revoking the privileges.

Unfortunately, by treating copying in this way, Scheme 2 sacrifices the one advantage of Scheme 1: the ability to produce a copy with identical revocation powers. A capability cannot be moved by copying it and discarding the original, since the copy, being a son of the original would lack the power of revocation over other such sons, and would therefore be an inadequate replacement for the original.

The problem is caused by two conflicting notions of what copying is for, suggesting that two different operations are needed.

Scheme 3: By combining the notions of Scheme 1 and Scheme 2, we define a "reduced family tree" of capabilities generated by a pair of copy operations:

$$C_b \leftarrow C_a \qquad \text{(as in Scheme 1)}$$

$$C_b \leftarrow \text{son}\ (C_a) \qquad \text{(as in Scheme 2)} \quad .$$

The reduced family tree is generated as follows:

a) The initial capability occupies the root node.

b) The copy operation produces a new capability occupying the same node as the capability being copied.

c) The son operation produces a new capability occupying a new son node of the node containing the capability being copied.

A reduced version of the family tree in Figure 3.2-5 is shown in Figure 3.2-6. As in Scheme 2, revocation affects entire subtrees.

Thus, while Scheme 1 proposed a set of capabilities, and

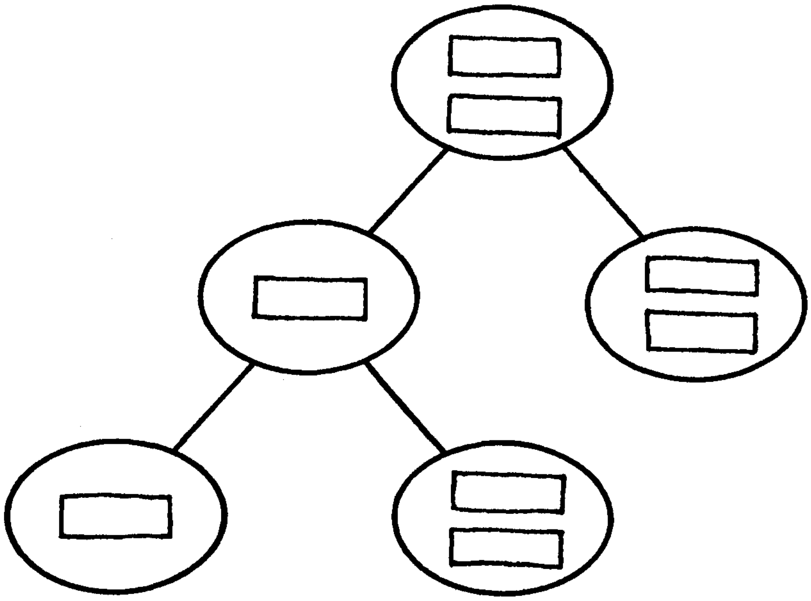


Figure 3.2-6: A reduced family tree corresponding to Figure 3.2-5

Scheme 2 proposed a _tree_ of capabilities, Scheme 3 proposes a _tree of sets_ of capabilities. This is intended to capture the observed tendency of the capabilities for a given object to fall naturally into subsets containing equivalent capabilities (as mentioned in Chapter 2). In this scheme, the capabilities in each family tree node always contain the same privileges, since any change to one of them affects them all. On the other hand, capabilities in different nodes of the family tree can contain different privileges, and interact according to the rules of descendant revocation. This contrasts with a system like TCS, in which any two capabilities may contain different privileges, and reducing the privileges in one never affects the other.

One valid complaint about this scheme is that it forces an early decision as to which capabilities one may eventually wish to revoke. The recommended policy would be to use a revocable capability whenever there was any doubt concerning the trustworthiness of a receiving domain. Indeed, this is the justification for our restriction that capabilities with the same revocation status may not differ in their privileges. It seems intuitively reasonable that any level of trust less than complete trust may be subject to change, especially since incomplete trust is often based on incomplete knowledge. Thus, the same reservations which prompt one to pass a capability with restricted privileges should prompt one to make that capability revocable.

We wish to adopt the reduced family tree as the model of revocation behavior in NCS. The implementation described in Section 3.6 produces exactly this behavior, in addition to a sealed-capability type-extension mechanism. In the implementation, these two mechanisms not only interact strongly, but also display a striking similarity, despite their apparently dissimilar definitions. We therefore present, in Section 3.4, a more general mechanism which subsumes them both. It should be emphasized that this generalized mechanism does <u>not</u> provide any additional privilege revocation features, but functions rather as an interesting descriptive device unifying two seemingly different constructs. We will continue to use the family tree description as well, where appropriate.

3.3 Interactions with Type-Extension

In the design of NCS, we wish to adopt the sealed-capability approach to type extension, as described in Chapter 2. The minor restrictions in the TCS capability sealing mechanism of Section 2.7 will be eliminated, but this is not a major improvement. What is crucial, however, is the proper interaction of type-extension with revocation.

One aspect of such proper interaction has already been mentioned: it must be possible to revoke access to extended objects, as well as to base-level objects. Moreover, such revocation must be handled through the normal base-level revoke operation, without, for example, any need to explicitly notify the layer which implements the object that access is being revoked. Thus, no extra burden is placed on the user of the extended object, although certain mild constraints are placed on the implementing layer, as we shall see in Section 3.5.

Another interaction which must be handled properly is the revocation of capabilities for objects which are representations of extended objects. Since such capabilities can be sealed inside the extended object capabilities (to any depth), the revoke operation, during its hypothetical global search, must be able to look inside the extended object capabilities and remove the appropriate privileges from any eligible representation capabilities it finds there. This requirement rules out such implementations as that described for TCS in Section 2.7, in which a sealed representation capability has no explicit existence, but can be reconstructed on the basis of certain assumptions, the key assumption being that its privileges remain constant, which can be false in a system providing revocation. The important point here is not that a layer implementing an extended type would normally be in the position of having its representation capabilities revoked, but that it must not be possible for the freely available type-extension mechanism to be misused to "hide" capabilities from the revocation mechanism.

3.4 Generalized Sealing

In discussing capabilities, we have sometimes referred to them as being information "sealed in a box." This characterization has been used by Lampson [La 69], Morris [Mo 73] and others, and suggests the obvious generalization of repeated sealing, i.e. boxes within boxes. We have already seen one situation in which such a construct was useful: the sealed capability approach to type extension. In this section, we propose a much more general capability sealing mechanism for NCS which not only allows type extension without the restrictions imposed in Section 2.7, but also provides for revocation which follows the reduced family tree discipline of Section 3.2.

The act of sealing information in a box can have two consequences:

a) Reading of the information is prevented.

b) Modification of the information is prevented.

Morris [Mo 73] has referred to sealing as being *transparent* if only restriction (b) holds, and *opaque* if both restrictions (a) and (b) hold. We wish to generalize this distinction to allow *partially* opaque sealing of capabilities. This is accomplished by using boxes which are partly opaque and partly transparent. The opaque parts of a box have information on them; they cover and override the corresponding parts of the capability sealed inside. The transparent parts of a box allow the corresponding parts of the capability sealed inside to show through, and to thus remain in effect. It is not surprising that this selective "filtering" action can be used to capture the notion of privilege revocation, as we shall see.

The ability to seal things in boxes is carefully controlled, as is the ability to unseal boxes and thus gain access to their contents. Various kinds of boxes are available; the sealing and/or unsealing of a given kind of box is itself authorized by an appropriately privileged capability for a type. In this scheme, a type is simply a template for making boxes. As we will shall see, such templates, when used in a particular way, generate a HYDRA-style 3-level object hierarchy, but this is not an explicit part of our definition of types. The association of boxes with types should not be taken as meaning that boxes are themselves objects, which they are not. A box is merely the "skin" of a capability, and has no independent existence of its own.

The format of boxes is shown in Figure 3.4-1. A type is just a template for making boxes, and a capability is just a box containing something, hence this can also be used as the format of types and capabilities. One can think of the fields as being written as "trit strings" where each digit takes its values from {0,1,transparent}. The fields are all familiar from previous discussions, with the exception of the "capability-ID" field. This field identifies the capability, and serves to distinguish it (and all copies of it) from other similar capabilities, even if their type, privileges and object-ID fields are the same. This is important, for example, during the hypothetical search which performs revocation of privileges.

In spite of the alarming size of these capabilities, we continue to assume that each addressable location in memory is capable of containing one. At the same time, we will take the apparently paradoxical view that each of the four fields in a capability is the full size of a data item which could be stored in the same location as the entire capability. This kind of behavior should come as no surprise in a system which allows capabilities to be nested to any depth without increasing in size.

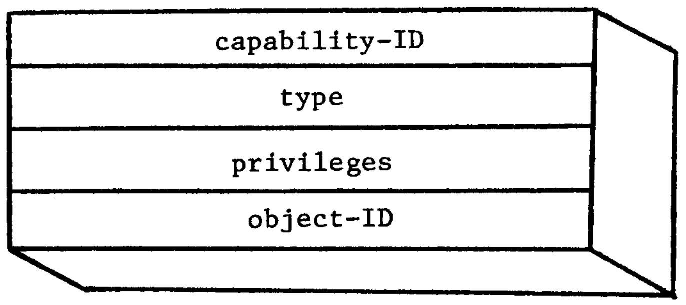


Figure 3.4-1: Format of boxes, types, and capabilities

The seal and unseal operations are fairly simple. Executing

$$C_S \leftarrow \text{seal}\ (C, C_T)$$

creates capability C_S by sealing C in a box specified by the template contained in type T, as authorized by the privilege of sealing in C_T. The box produced is a verbatim copy of the template in type T, with the exception that the capability-ID and object-ID fields, if opaque, will have the same new unique ID written on them. Executing

$$C \leftarrow \text{unseal}\ (C_S, C_T)$$

reverses the process by removing one or more boxes from C_S until it suceeds in removing a box whose type field is opaque. The value of its type field must match that of the template in type T; otherwise, an error is signalled and no value is returned. The capability C_T must contain the privilege of unsealing.

Given the above mechanism, various kinds of templates can be defined, of which we will use three.

The simplest kind of template is shown in Figure 3.4-2. It is completely transparent, and generates boxes we will call "lockers," since their only function is to prevent their possessors from modifying their contents in any way. In particular, lockers are used to control revocation, as will be discussed in the next section. A type containing this template is provided by the system, and a capability for the type, allowing sealing but not unsealing, is publicly available.

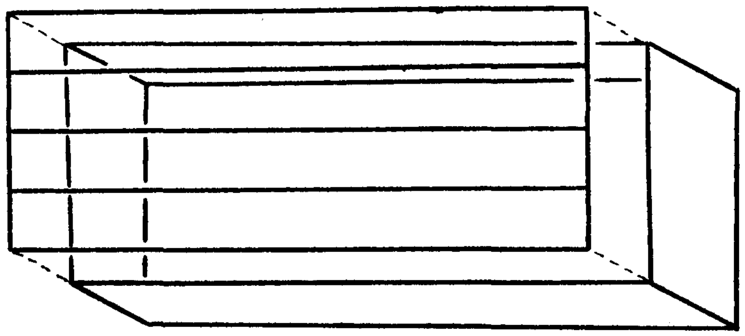


Figure 3.4-2: A "locker"

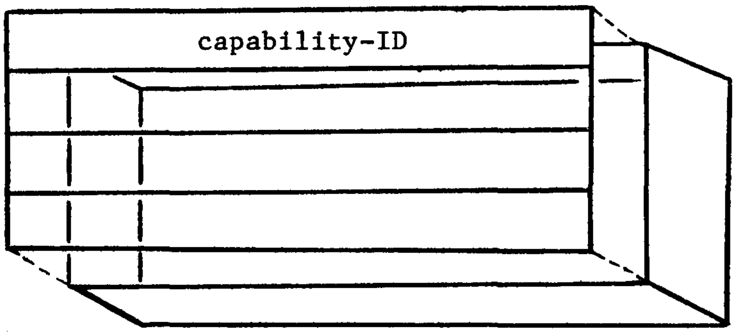


Figure 3.4-3: A "revoker"

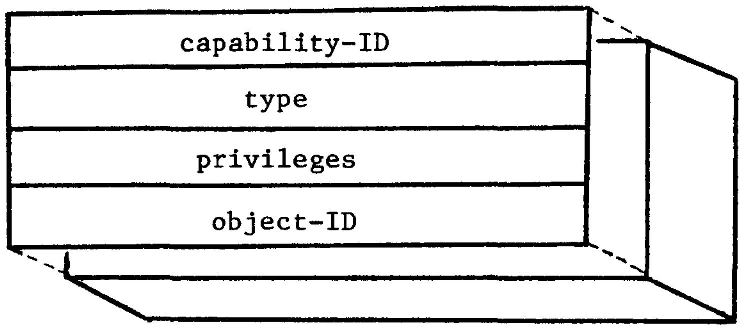


Figure 3.4-4: An "extender"

A slightly more complicated template is shown in Figure 3.4-3. It is transparent except for an opaque capability-ID field, and generates boxes we will call "revokers." (Recall from the definition of the seal operation that each new revoker will thus have its own new capability-ID.) As will be seen in the next section, sealing a capability in a revoker box is equivalent to generating a new son-node in the reduced family tree. A type containing this template is also publicly available for sealing, but not unsealing.

The third kind of template is shown in Figure 3.4-4. It is completely opaque. The value of the type field is just the ID of the type containing the template. Boxes generated by such templates we will call "extenders." Extender boxes provide a sealed-capability type extension facility as described in Chapter 2. Several types containing such templates are predefined by the system, and an operation is provided for creating more such types on demand. These types are _not_ made publicly accessible.

There may be other kinds of templates which would prove interesting or useful, but we will not pursue this here. Instead, we turn to the relationship between the sealing mechanism and the other operations of the base-level system.

As mentioned previously, the base-level operations taking capabilities as arguments can be divided into two groups. Most of them simply "look at" the capabilities as the names of objects which are their actual arguments. A few of them are directly concerned with the capabilities themselves. The treatment of capabilities by the former operations is quite simple: they always rely on the external appearance of a capability, regardless of its internal structure of nested boxes. For the latter operations, the situation is more complex.

In addition to the seal and unseal operations described above, there are four kinds of base-level operations which manipulate capabilities themselves:

a) creation of base-level objects

b) copying of capabilities

c) erasing (overwriting) of capabilities

d) revocation of privileges

Each of these is now described in some detail.

Creation of base-level objects is involved with the capability mechanism in two ways. On the one hand, each new object must be named by an initial capability which is to be returned as the value of the creation operation. The fabrication of this capability can best be described as the sealing of an empty extender box, using a type owned by the base-level system as a template. Thus, base-level object creation depends on sealing.

On the other hand, sealing depends on the previous creation of types, which are base-level objects. Types corresponding to the various base level objects (segments, domains, etc.) are created at system initialization time. At least the "root" type (ID = 'type') must be created "out of thin air," and in fact, all base-level types are presumably created this way (although conceptually, one can think of the base-level system using its own create_type operation, which would in turn use the seal operation specifying the root type as a template).

Copying of capabilities is conceptually simple in this scheme.

The entire capability, including any number of nested boxes, is reproduced exactly, so that the new capability is indistinguishable from the original. Thus, executing

$$C_1 \leftarrow C_2$$

results in two identical capabilities.

The overwriting of a capability with data or with another capability is also simple. The overwritten capability is destroyed, with no particular side-effects except for the obvious possibility that some previously allowable actions are now forbidden.

The most complicated operation in this scheme is revocation, which is performed by executing

$$\text{revoke } (C,P)$$

which revokes from C (and all copies of C) any privileges which are zero in mask P. The outermost box of C is required to be a revoker. Note that the revoke operation, like the TCS reduce operation, is portrayed as modifying an existing capability, rather than producing a new one (cf. seal, unseal). Generalizing the discussions of Sections 3.2 and 3.3, we will hypothesize that the underlying capability machinery performs a global search anytime an existing capability is modified and reflects the changes in all copies of the capability, even those which are sealed in nested boxes.* (These copies are easily recognized by their

*In the design being described, this hypothetical search is exploited only by revocation. Section 3.8 will survey some possible elaborations on the design, two of which would also depend on this search. At risk of repetition, we again point out that this global search is only a descriptive device, and is not actually implemented as such.

capability-ID fields.) The particular modification performed by the revoke operator is the writing of an opaque 0 at each position in the privilege field of C which corresponds to a 0 in the mask P. This is only done, however, if the outermost box of C is a revoker; the revoke operation refuses to write on any other kind of box, and signals an error if this is attempted.

Operations must also be provided for testing the tag of a memory location to see whether it contains a capability, and if it does, for displaying the various fields of the capability. These operations are straightforward and require no detailed discussion.

3.5 Examples of Generalized Sealing

This section outlines some intended uses of the NCS sealing mechanism just described, and reviews the goals listed at the end of Chapter 2, to assure that they have all been met. The description of directories and other specific facilities which can be implemented using NCS capability sealing is postponed until Chapter 4.

There is more than one reasonable way to use the NCS sealing mechanism for revocation, depending upon the exact situation (i.e. the number of domains involved and their relationships to each other). In the example situations below, it is assumed that domain A possesses a capability and wishes to pass it to one or more domains B. In choosing a method of doing this, A controls the possibility of later revocation of the various capabilities passed.

To illustrate the various situations, the sealed capabilities are shown as arranged in corresponding reduced family trees. Recall that sealing a capability in a revoker box corresponds to generating a new son node in the tree.

The simplest situation is one in which A completely trusts B, and simply passes a copy (C_B) of its own capability (C_A), as shown in Figure 3.5-1. The most important example of this is in "system calls," in which A regards domain calls on B as being operations of its "extended machine." As will be seen in Section 3.6, the passing of such non-sealed capability parameters represents a considerable saving. This is very significant, since experience suggests that a great majority of domain calls executed are in fact system calls [SS 72]. There are also logical reasons for passing non-sealed capabilities on certain kinds of system calls, namely those which are part of extended mechanisms for capability storage and/or transmission, such as directories or message channels.

If A does not have complete trust in B, then before passing C_A to B, A should seal it in a revoker box. By keeping one copy (C_R) of the sealed capability, and passing another (C_B) to B, A retains the power of later revoking B's privileges. This situation is illustrated in Figure 3.5-2.

If A wishes to pass revocable capabilities to the several domains $B_1, B_2, \ldots, B_n$, one alternative would be the creation of C_R as above by sealing C_A in a revoker, followed by the passing of n copies of C_R (denoted C_{B_i}) to the domains B_i, as shown in Figure 3.5-3. (Note that this is just the situation

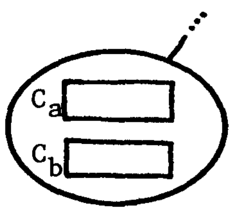


Figure 3.5-1: Passing a non-revocable capability

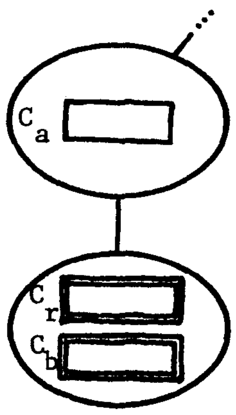


Figure 3.5-2: Passing a revocable capability

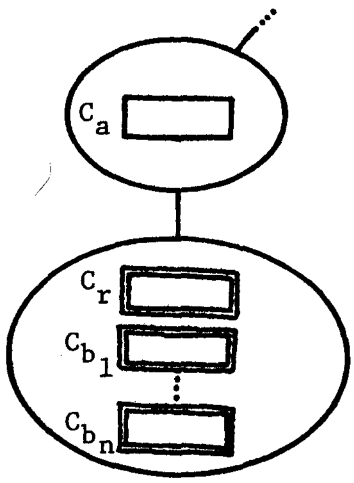


Figure 3.5-3: Passing simultaneously revocable capabilities

which would arise if A passed C_{B_1} to B_1, and B_1, completely trusting $B_2 \cdots B_n$, in turn passed copies to them.) There are two limitations to this use of the mechanism. One is the non-selectivity of A's power of revocation; revoking privileges from any of the domains B_i requires revoking from all of them. The other limitation is the lack of isolation between the domains B_i; any of them is capable of revoking the privileges of all of them, which may be inappropriate.

Both of these limitations can be avoided by simply handling each of the domains B_i separately as in Figure 3.5-4. This allows selective revocation from each of the B_i, and isolates them from each other in case they are mutually suspicious. For example, the various B_i may be the renters of a program owned by A, in which case both of these considerations are important.

On the other hand, there are situations in which A does not need to revoke the privileges of the various B_i selectively, but does wish to isolate them from each other. For example, a professor may wish to grant access to a grading program to all of the students in his class. He certainly wishes to prevent the students from revoking this privilege from each other, but may well have no desire to revoke their privileges independently, especially since this is somewhat costly and requires that A retain and use n different capabilities C_{R_i}. In this situation, A can produce a single C_R by sealing C_A in a revoker box, and can then distribute the capabilities C_{B_i} produced by in turn sealing C_R in a locker box, as shown in Figure 3.5-5. This not only eases simultaneous revocation, but is significantly cheaper,

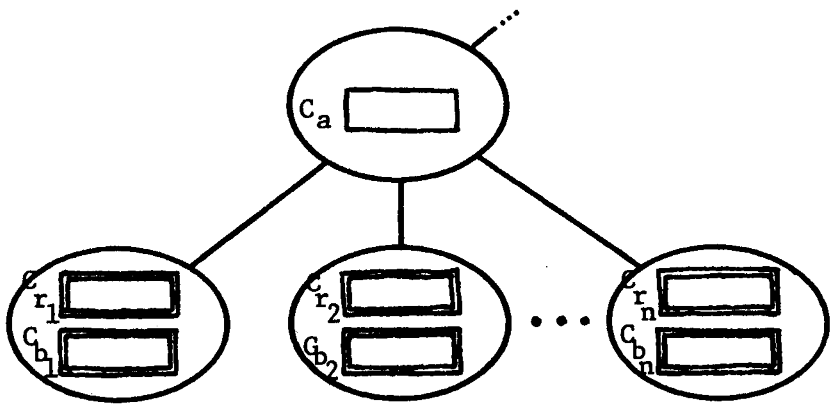


Figure 3.5-4: Passing independently revocable capabilities

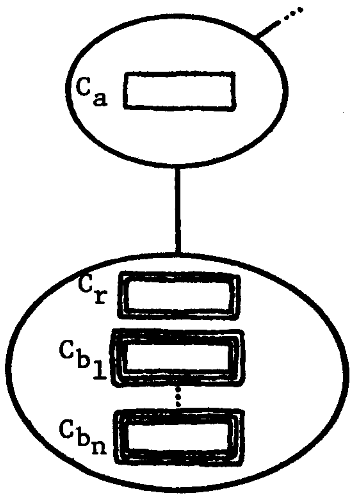


Figure 3.5-5: Passing isolated simultaneously revocable capabilities

given the implementation to be described.

From this discussion, it should be clear that goals 2, 3, 4 and 5 of Section 2.8 are satisfied by the proposed design. Goal 6, that of reasonable cost, will be treated in the next section, which proposes an implementation for sealed capabilities and discusses its efficiency. This leaves only goal 1, that of immediate revocation, and goal 7, that of proper interaction between revocation and type extension. Between them, these two goals generate one fairly subtle problem, which must be discussed before all the goals can be considered satisfied.

It is clear that revocation as defined takes effect immediately in the sense that the privileges of the appropriate capabilities are immediately modified. This is only significant, however, to the extent that the corresponding operations on the object in question are immediately prohibited, which in turn depends on the checking of the privileges by the operations. One can imagine the following kind of scenario, in which revocation is effectively delayed. Suppose that domain A in process P_A passes to domain B in process P_B a capability to access X, which is an extended object implemented by layer L. Suppose that layer L is represented by domain L_A in P_A and by domain L_B in P_B. Assuming that we can say nothing about the relative execution speeds of P_A and P_B [Di 68] the sequence shown in Figure 3.5-6 is one possible outcome, and produces an effective delay in revocation which is visible to A. Strictly speaking, the problem here is caused by the occurrence of step A1 between steps B2 and B3, which should be executed together as a "critical section." Synchronization between

<u>in P_A</u>	<u>in P_B</u>
	B1. B calls L_B to modify X
	B2. L_B verifies that B is authorized to modify X
A1. A revokes B's privilege to modify X	
A2. A calls L_A to examine X	
A3. L_A returns to A the original state of X	
	B3. L_B performs the previously checked modification of X and returns to B
A4. A calls L_A to examine X	
A5. L_A returns to A the modified state of X	

Figure 3.5-6

the base-level system and higher layers is fraught with difficulties, however, hence the following alternative seems preferable: when a layer is about to access the representation of an object, it must first lock all parts of the representation to be touched and then check to see that the requested operation is authorized. In many cases, this interlocking would be necessary anyway; the major change due to revocation is the moving of privilege checking inside of the critical section. (In particular this means that pre-checking of privileges as an integral part of the domain call machinery [St 73, Wu 74] is not very useful in a system in which privileges are revocable.)

In the context of Figure 3.5-6, such checking would delay step A3 until after step B3. The crucial point is that this renders the situation indistinguishable* from one in which step B3 occurred before A1. Thus, although an access may occur slightly after permission to perform it has been revoked, there is no way for a properly written (i.e. timing independent) program to detect this occurrence.

3.6 Implementation of Generalized Sealing in NCS

As in previous discussions, we begin by describing the representations of capabilities themselves. A tagged memory location holding a capability appears to the user to contain a rather large amount of information, but in actuality it contains a short form

*Except for real-time delays.

of the capability, consisting of a "locker bit"* and the ID of the capability as shown in Figure 3.6-1. The other fields are stored elsewhere, and the ID is sufficient to locate them, allowing reconstruction of the complete _long form_ of the capability.

The most important advantage of this approach is that it allows the changeable information (e.g. revocable privileges) in all copies of a capability to be centralized and thus updated without a global search. This is crucial to the practicality of the scheme, and will be discussed in more detail shortly.

This approach also allows the effective storage of an entire capability in a single practical-sized word of a tagged memory. For example, on the terribly pessimistic assumption that a new unique ID is generated every 10 microseconds, the use of 48 bit words would allow the system to run continuously for about a century without exhausting its supply of names. Using a name-space compaction approach and a somewhat more realistic level of pessimism would probably allow the use of 32 bit words without requiring an objectionable frequency of system shutdowns to perform the compactions (i.e. once every few weeks or months, at worst).

An attractive way to store the boxes which constitute the actual substance of the capabilities would be in a global hash table containing small fixed sized entries and keyed on unique IDs. The map, as described in Section 2.2, is just such a structure, which suggests implementing each box as a map entry. This approach yields an integrated structure for the reconstruction and interpretation of nested capabilities from their short forms. The

*This is _not_ the same as the tag bit on the capability, and will be discussed below.

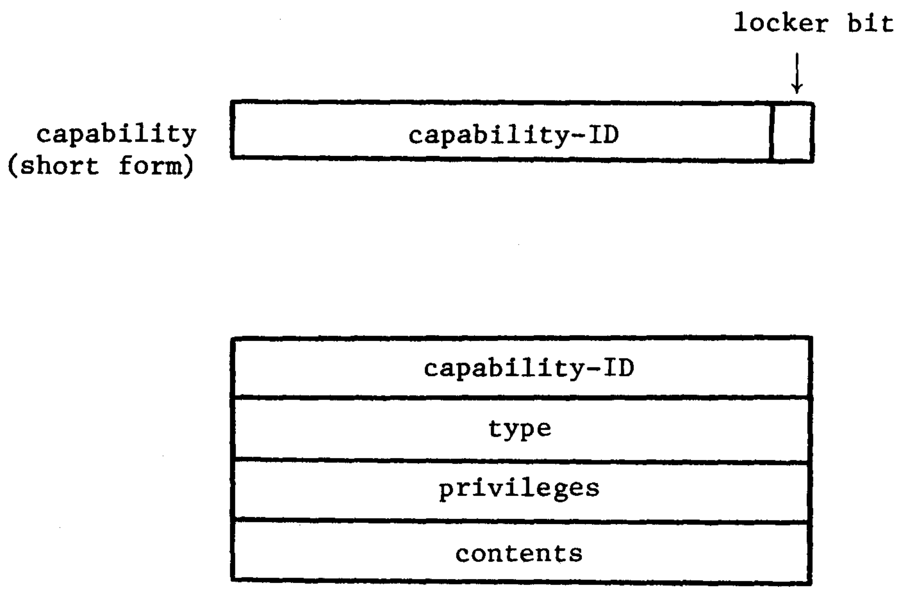


Figure 3.6-1: Format of (short-form) capabilities and map entries

increase in size and complexity of the map machinery, while non-negligible, is not excessive.

The format of a map entry is shown in Figure 3.6-1. The capability-ID, type and privileges fields of the corresponding box are represented directly, while the object-ID field is replaced by a new "contents" field which serves to locate the contents of the box. Map entries for various particular kinds of boxes are shown in Figure 3.6-2.

Base level capabilities, while conceptually the same as other extenders, are represented in a special form. The contents field contains the physical address of the object, hence these map entries correspond to the map entries in a system like TCS. The privilege field would always contain all 1's since revocation does not operate on extender boxes, hence its value can be implicit; the space in the map entry is used to record the size of the base-level object instead.

Normal (i.e. user created) extender boxes are represented similarly, but their contents are capabilities, rather than physical addresses, and they make no use of their privilege fields.

Revoker boxes represent their transparent type and privilege fields as all 1's. In the case of the type field, this value is a constant which is specially recognized by the capability reconstruction machinery. In the case of the privilege field, it is used as a mask, hence any 0's written in it are effectively opaque, as required for revocation.

Note that no map entry format is described for locker boxes. Locker boxes are so simple that they may be implemented in a much

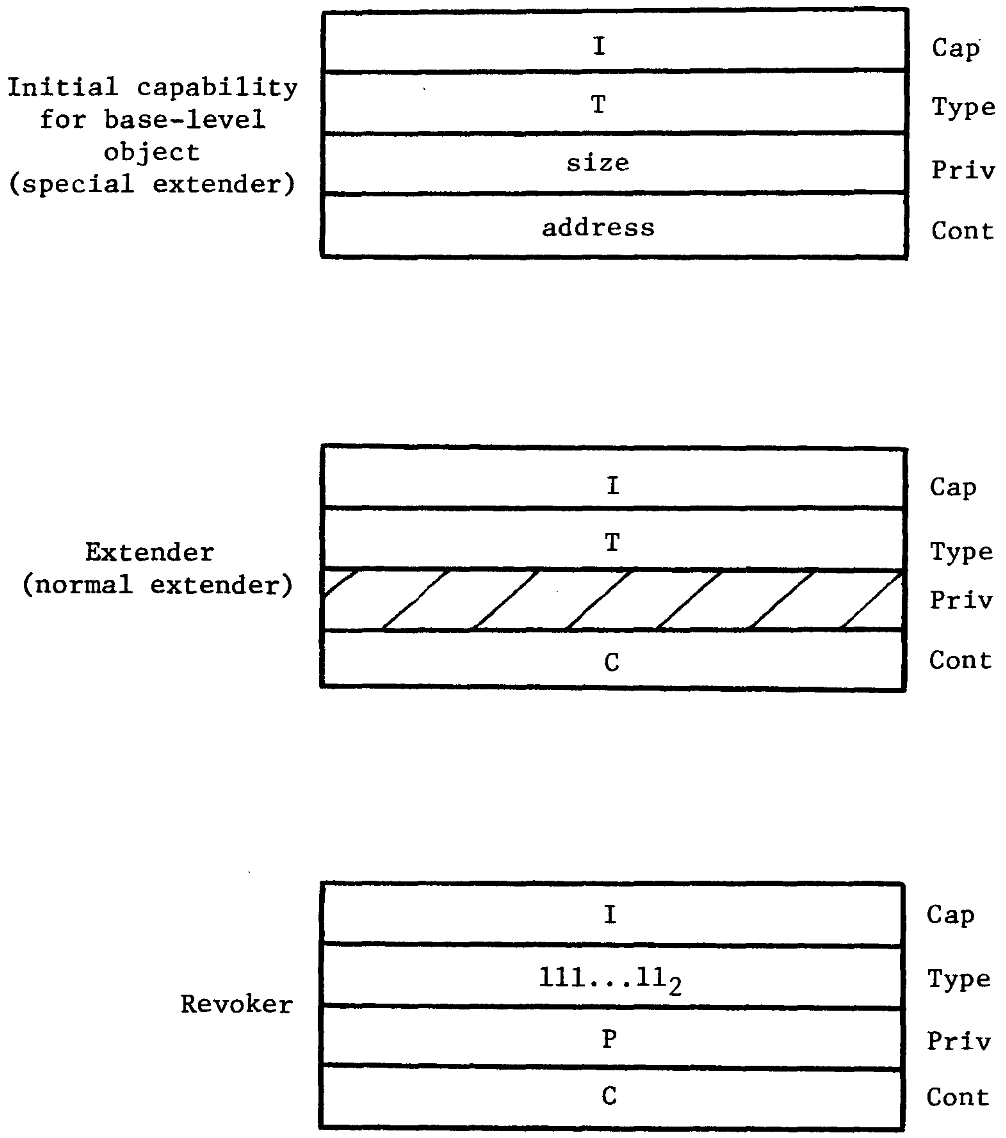

Initial capability for base-level object (special extender)
I
T
size
address
Cap
Type
Priv
Cont
Extender (normal extender)
I
T
C
Cap
Type
Priv
Cont
Revoker
I
111...11₂
P
C
Cap
Type
Priv
Cont

Figure 3.6-2: Map entries representing various kinds of boxes

cheaper way. As shown in Figure 3.6-1, a single locker-bit in the short form capability, rather than a complete map entry, serves to indicate the presence of one or more locker boxes. (Since they are transparent and non-removable, multiple consecutive locker boxes are indistinguishable from a single one.)

Given the described representations of the various kinds of boxes, the seal and unseal operations may be implemented as shown in Figures 3.6-3 and 3.6-4, respectively. The seal operation creates a new map-entry representing the new box and stores in its contents field the capability being sealed. Sealing in a locker box is handled specially by simply turning on the locker bit in the sealed capability. The unseal operation simply returns the contents of the appropriate extender box. (Recall that revokers and lockers can never be unsealed.) Figure 3.6-5 summarizes the various low-level facilities used in the description of these and other operations. These are assumed to be clear from previous discussions, with the exception of capability reconstruction ("Recap") and associative memory lookup ("Cap_find" and "Cont_find") which will be described shortly.

The creation of each new base-level object includes the construction of the "root" map entry representing its initial capability. This map-entry is self sufficient, in the sense that it does not depend on any other map entry for its proper interpretation. On the other hand, a map entry representing a revoker or extender box contains another capability; its one-word contents field holds the short form of the capability, hence its interpretation is dependent upon the other map entry holding the rest of

$C_S \leftarrow \mathrm{seal}(C, C_T)$

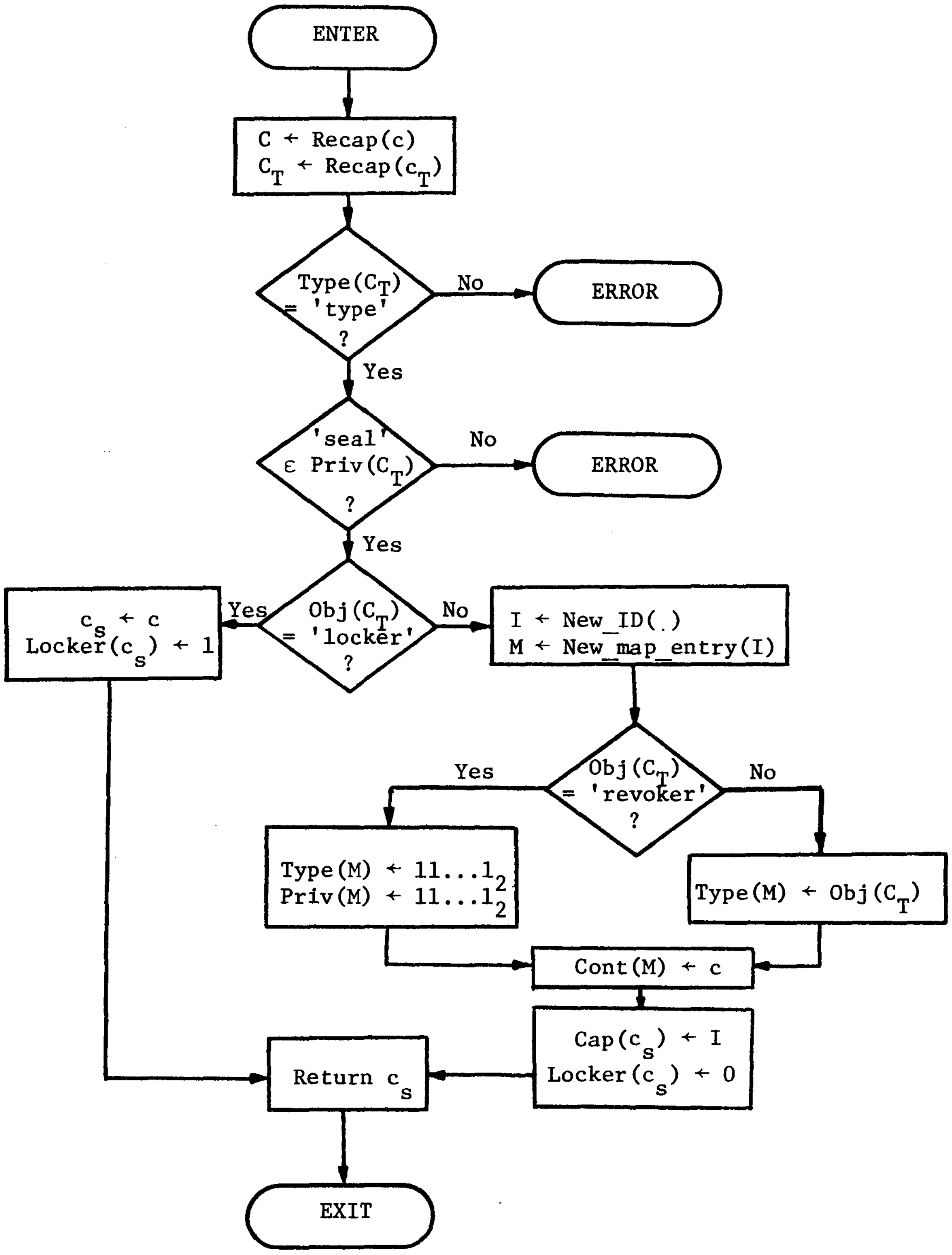


Figure 3.6-3: NCS seal operation

$C_n \leftarrow \text{unseal}(C, C_T)$

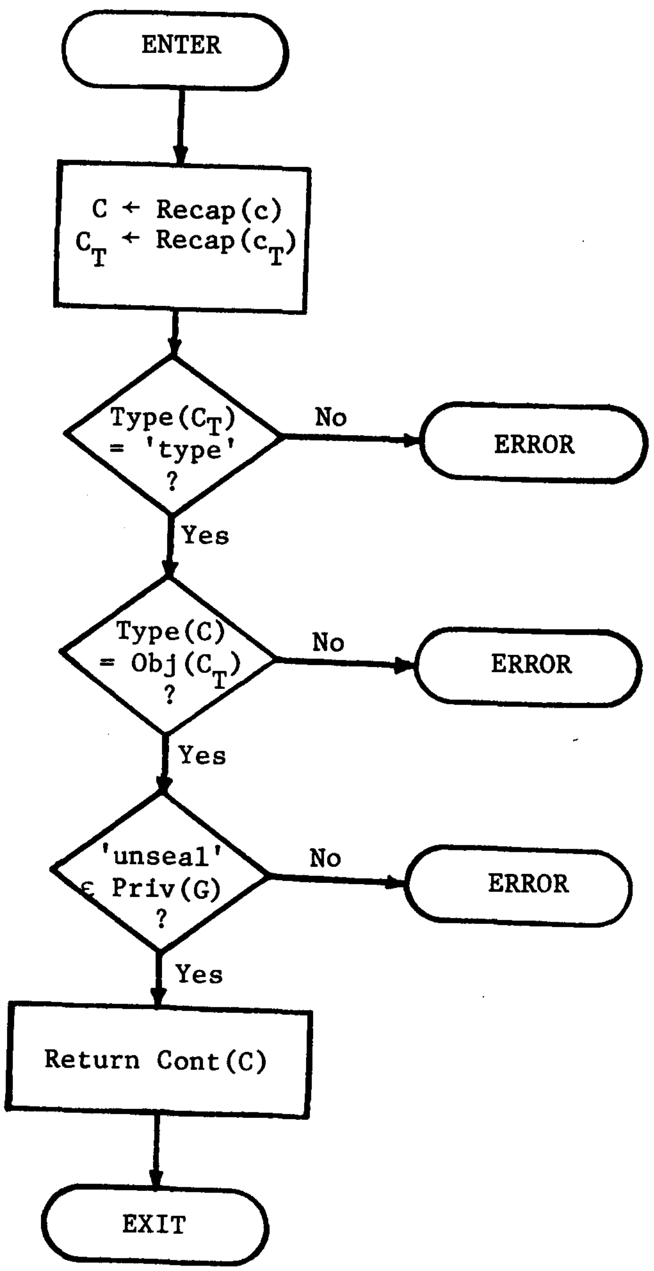


Figure 3.6-4: NCS unseal operation

<u>Fields in various data structures</u> (see also corresponding figures)

Cap (x)	capability-ID
Type (x)	type
Priv (x)	privileges
Obj (x)	object-ID
Size (x)	size
Cont (x)	contents

<u>Unique names</u>

New_ID ( )	generates a new unique ID

<u>Map</u>

New_map_entry (I)	creates map entry with capability-ID = I
Map_entry (I)	finds map entry with capability-ID = I
Delete_map_entry (M)	deletes map entry M

<u>Capability reconstruction</u>

Recap (c)	reconstructs long form of c

<u>Associative memory</u>

Cap_find (I)	find entry with capability-ID = I (else LRU entry)
Cont_find (x)	find entry with contents = x (else LRU entry)

Figure 3.6-5 Low level facilities used by operations

that capability. Thus, repeated sealing of a base-level object results in the generation of a tree of map entries, which combines the functions of the type tree of Section 2.6 and the reduced family tree of Section 3.2. An example of such a tree is shown in Figure 3.6-6, in which a segment is used as the representation of an extended object of type 'directory,' for which various capabilities have been distributed.

It is important to note that while the seal operation generates such tree structures, the unseal operation does not dismantle them. For example, in Figure 3.6-6, if the layer implementing directories unseals C_3 to obtain C_5, the map structure remains unchanged. The mechanism for deletion of unneeded map entries will be discussed later.

In order to reconstruct the long form of a capability, it is necessary to examine the boxes which compose it, starting with the outermost and working inward, until all fields are completely opaque. Given the particular kinds of boxes used in our scheme, this simply entails scanning down a chain of (zero or more) revokers until a non-revoker box is encountered. This reconstruction procedure, shown in Figure 3.6-7, is rather similar to the "following" procedure for indirection chains of Section 2.4. In other figures, the capability reconstruction procedure is referred to in the form

$$C \leftarrow \text{Recap}(c)$$

where c denotes the short form and C the reconstructed long form of the capability. In addition to the visible long form, the reconstruction process also recovers the representation pointer

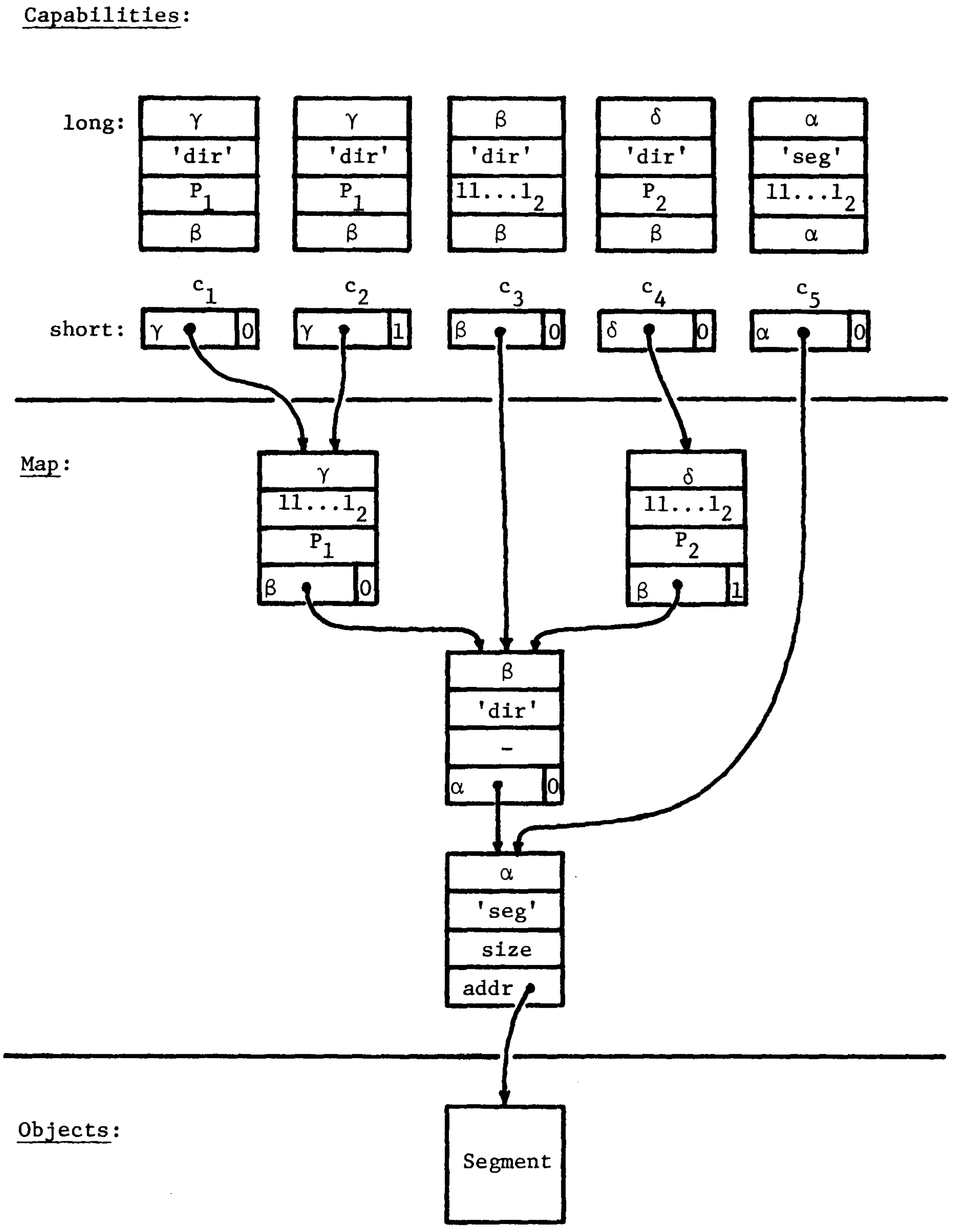


Figure 3.6-6: A map entry tree

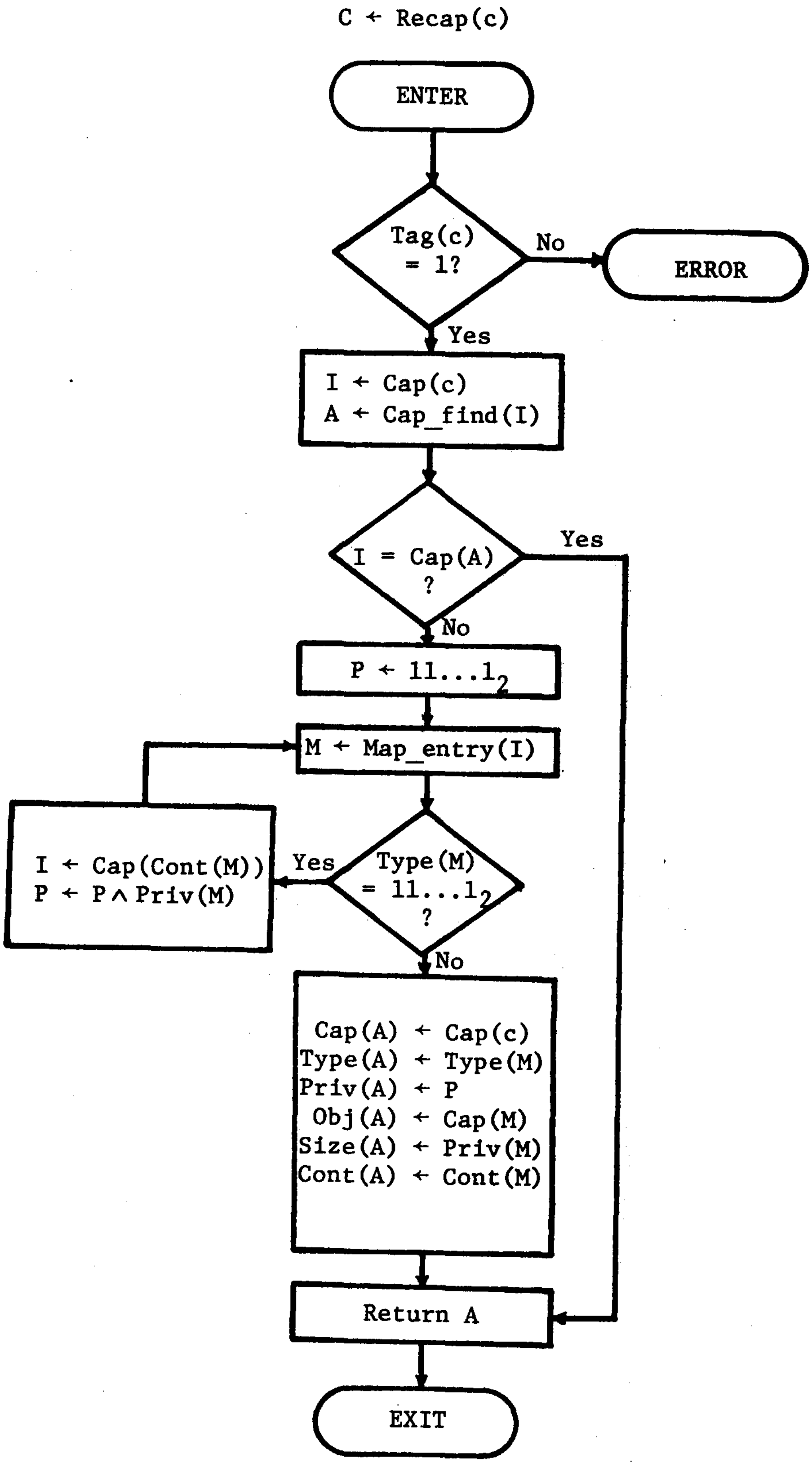
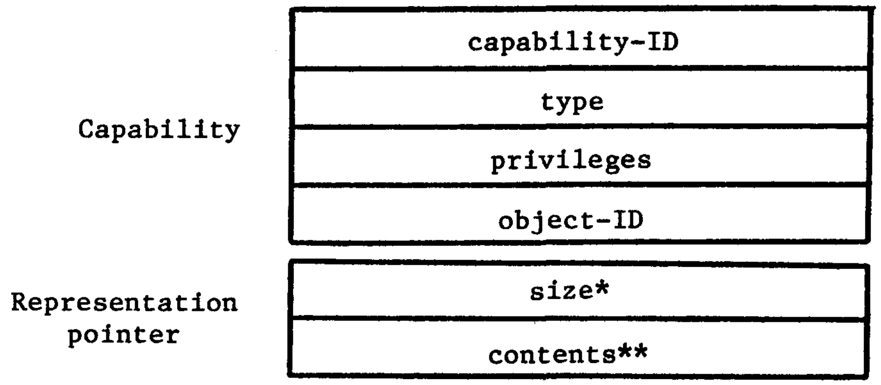


Figure 3.6-7: Capability reconstruction

from the capability to the object, which consists of the short form representation capability in the case of extended objects, and the address and size for base-level objects. Thus, the result of the reconstruction process is a mapping, as shown in Figure 3.6-8.

The cost of the reconstruction process is relatively high, since it involves scanning a chain of map entries, each of which must be located by hashing into the map. The retention of the most active mappings in fast hardware thus becomes even more important than in a system like TCS. The associative memory discussed in Section 2.2 could be used without change to hold map entries from active chains and thus speed up the scan. On the other hand, a 50% increase in the size of the associative memory entries allows them to contain entire mappings, rather than single map entries. On the average, this modification would probably not provide a very dramatic improvement in speed (by bypassing the reconstruction process entirely, rather than merely accelerating it) and might even slightly reduce the efficiency of space utilization in the associative memory (if the average chain length was less than 1.5 map entries). It is desirable, however, since it allows a fixed amount of associative memory space to effectively contain a chain of arbitrary length, thus preventing long chains from severely degrading performance by filling up the associative memory. We therefore specify the associative memory as containing the several most recently used complete mappings. The exact number to be retained would depend on several considerations, ranging from available hardware components to expected usage patterns. Two factors which favor maximizing the number are the relatively high



* Base-level objects only

** Address if base-level object
Representation capability (short form) if extended object

Figure 3.6-8: A mapping
(as stored in the associative memory)

cost of initial loading (= capability reconstruction) and the fact that the retained mappings remain valid through domain-calls and process switching.

In the various figures, the associative memory facilities are represented in the form:

$$A \leftarrow \text{Cap\_find}\ (X)$$

$$A \leftarrow \text{Cont\_find}\ (X) \quad .$$

Each of these finds an associative memory entry whose appropriate field (capability-ID or contents) contains the value X. If no such entry is present, the least recently used entry is found.

The revoke operation is quite straightforward in terms of its effect on the map. Since all copies of a given revoker box are represented by a single map entry, the masking of the privilege field of that map entry automatically revokes the corresponding privileges from all the copies, including those sealed inside other capabilities. The only problem is that some of these latter capabilities may already have been reconstructed and saved in the associative memory, necessitating their removal.

Unfortunately, the names of all such capabilities cannot be determined from the name of the capability being revoked, except by introducing a complicated and fragile backpointer structure into the map-entry trees. One way of dealing with this problem is to completely flush the associative memory on each revocation. This will be satisfactory if the frequency of revocation is relatively low. If revocation is a sufficiently frequent occurrence, however, this will drastically reduce the utility of the associative memory by forcing heavy use of the expensive reloading procedure. A quite satisfactory compromise between total flushing of the associative memory and selective removal of only the affected capabilities is the removal of all capabilities for the same object. This is easily accomplished using the "Cont_find" feature of the associative memory, as shown in Figure 3.6-9. (For simplicity, we have assumed that 0 is not a valid value of the Cap or Cont fields of a mapping, and can therefore be used to disable an associative memory entry.) This semi-selective removal will sometimes force unnecessary reloading of capabilities which were not affected by the revocation, but this will only happen when a capability is revoked and another capability for the same object which is not its descendant in the family tree appears in the associative memory.*

The storage of inactive map entries in secondary memory is much the same in NCS as in TCS. Each TCS map entry corresponds to a complete tree in NCS, but only the active paths in the complete tree need be kept in primary memory. It seems likely that known techniques for localizing list structures in secondary memory [Bo 67] could contribute significantly to minimizing the overhead incurred when an inactive path becomes active and must be brought into primary memory.

*One possible frequent example of this would be revocation of a domain-call parameter upon return from the call. Revocation of the callee's capability would unnecessarily remove the caller's own capability from the associative memory. This could be avoided using a modification suggested by Peter Bishop of M.I.T., in which the mapping produced by the capability reconstruction mechanism would include the length of the chain scanned to produce it. By comparing this value for the capability being revoked and the capability being removed from the associative memory, one could avoid removing tree-ancestors of the revoked capability.

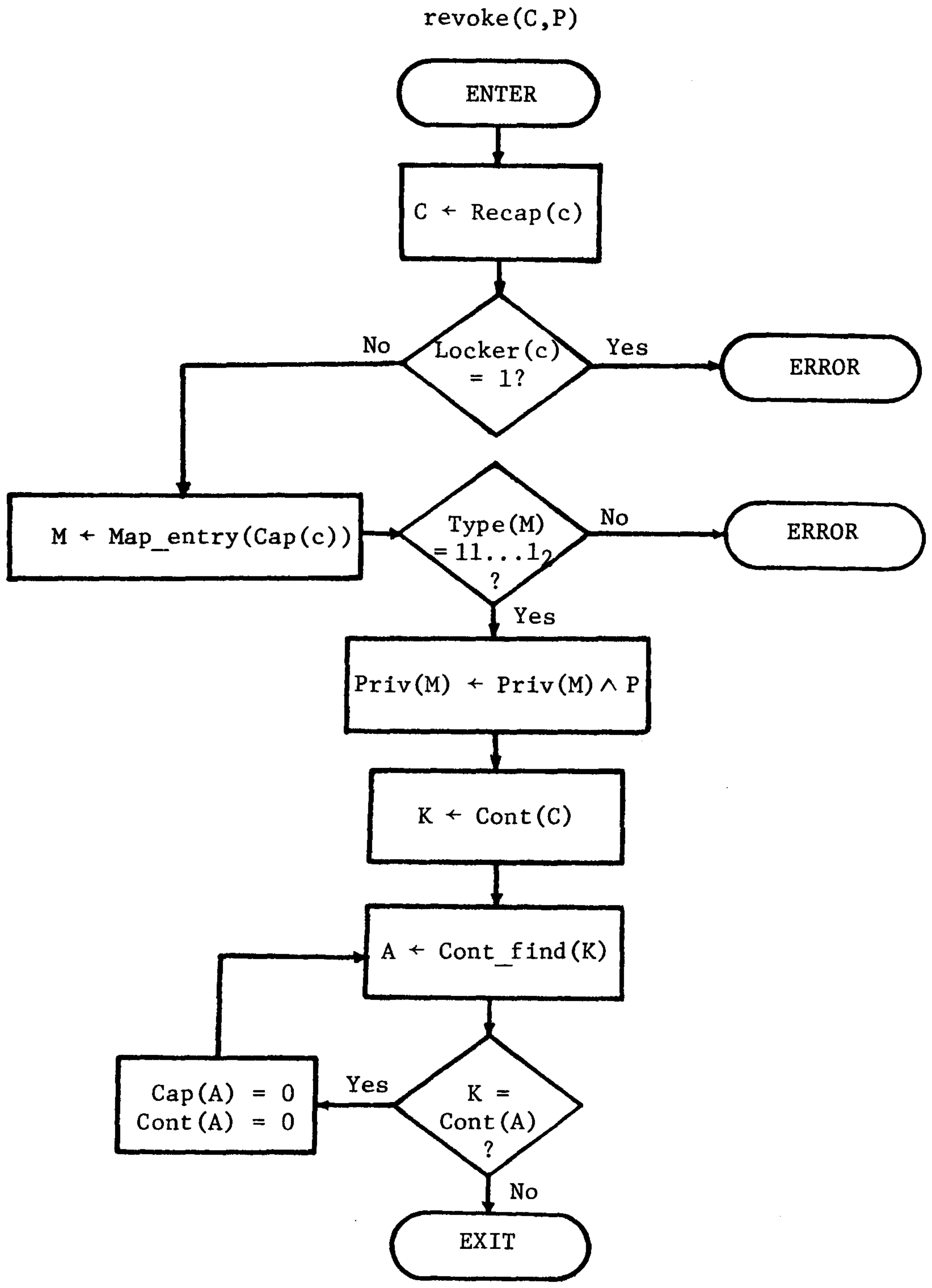


Figure 3.6-9: NCS revoke operation

3.7 Some Implementation Details

In describing an implemented system, it is often desirable to omit or simplify certain details which, while necessary in the implementation, are of little intrinsic interest, and tend to obscure the significant principles of the design. Unfortunately, in arguing the practicality of an unimplemented system like NCS, one is obliged to address such issues. This section is involved with such details relating to the maintenance of the system data structure we have called the map. Readers who find themselves growing bored with the arguments can skip the remainder of this section without significant loss of continuity.

The basic problem with the map as described thus far is the lack of any mechanism to keep it from filling up. For example, by repeatedly sealing a single capability at the relatively modest average rate of once per millisecond, a malicious domain could fill up a 1 million word map in a few minutes. In a system like TCS in which each map entry corresponds to a different object, one might be able to depend on the limitation of other resource usage for the object to limit usage of the map-space resource and prevent its exhaustion. This is clearly not the case in the new scheme, in which creation of map entries does not imply any other resource usage at all.

For this reason, it is necessary to treat map entries as an allocatable resource and thus limit the amount of map space available to each domain via its account. An account's reserve of available map space must be decremented each time a domain it funds creates a map entry, and incremented when the map entry is deleted. This requires that each map entry contain an extra field specifying the account which funds it since this may not be evident at the time at which it is deleted. Since unused map space resides on secondary storage, it is quite inexpensive, hence the allocation given to each account can be sufficiently generous that no reasonable program would ever exhaust it. The limit serves only to contain the damage done by pathological programs.

From the system's point of view, the problem is now solved since each user can harm only himself by extravagant use of map space. This is not really sufficient however; the consequences of such self-inflicted harm must not be too severe. A given account's allocation of map space can be cluttered by an undebugged program, hence some mechanism must be provided for prevention of and/or recovery from such a situation. Prevention cannot reasonably be expected of the base-level system, since it cannot distinguish between legitimate and illegitimate use of map space, hence recovery must be possible. We take the point of view, however, that this recovery need not be particularly easy or graceful, since, as mentioned previously, most use of the sealing mechanism is expected to be made via more civilized facilities rather than directly. The implementation of such facilities will be discussed in some detail in Chapter 4. At this point we are only concerned that such facilities use sealing in an orderly way.

What constitutes orderly use of the sealing mechanism? So far, no method has been described for removing unneeded map entries, hence any use of sealing will eventually fill up the map. The basic question is: when is a map entry no longer needed? There are at least two circumstances in which this is true:

a) Its privilege field is empty.

b) Its contents field points to a non-existent map entry or object.

If either of these conditions holds, the map entry is useless and may be deleted. Condition (a) suggests the revoke operation, upon reducing the privileges in a map entry, should check whether any privileges remain, and if not, delete the entry from the map. Condition (b) suggests that the capability reconstruction mechanism, upon encountering a map entry whose contents field contains such a "dead-end" capability (which we will call an "isolated" entry) should delete it from the map. A map entry whose contents field contains the address of a base level object is deleted when the object is deleted, thus isolating any map entries pointing to it. In general, the deletion of a map entry can cause one or more other map entries to become isolated, and thus be deleted the next time they are exercised by the reconstruction process. In this way, entire isolated subtrees can be gradually eliminated. (The case in which such entries are never subsequently exercised will be discussed shortly.)

Thus, in addition to its normal cleaning-up activities (destroying unneeded objects, etc.), a well-behaved domain should revoke any unneeded capabilities to clean up the map.

Similarly, the problem of cleaning up after the execution of an undebugged domain involves deletion of unneeded objects and map entries, followed by deletion of the domain itself. Problems can arise if the faulty domain has discarded all capabilities for any such object or map entry, which is then lost. A feature solving the lost object problem will be described in Chapter 4, but it would be expensive and cumbersome if used for every map entry. We therefore allow map entries to become lost and require that recovery from this situation be possible. This requires the revocation of all capabilities originally passed to the faulty domain, thus isolating the subtrees of map entries produced by its execution. The lost map entries in these trees will never be exercised, however, since by definition there are no capabilities for them.

For the reason just cited, some mechanism must be provided to exercise lost map entries. Moreover, even for map entries which are isolated but not lost, it would be helpful if their elimination from the map was automatic, since it may be some time before they are exercised. This can be accomplished by adding to the base-level system a relatively simple operation of the form:

exercise (I)

which simply exercises the I-th map entry by reconstructing its capability. A low-priority background process (sometimes called a "daemon" or "phantom") can now be constructed which uses the new operation to slowly sweep through the map eliminating isolated map entries. The rate at which this is done is a tradeoff between minimizing the extra load imposed on the map machinery and maximizing the rate at which map space is recovered. Given generous allocations of map space to the various accounts, the rate could probably be quite low. The exercise operation is not available to the users, since they have no use for it, but it is not at all dangerous, hence the background process need not be trusted by the base level system.

3.8 Possible Elaborations on the Design

There are several directions in which NCS as described in this chapter could be elaborated. We here digress briefly to discuss four examples, arranged in order of increasing difficulty of adding them to the implementation described.

A simple feature which might well be included in an actual system allows examination of the relationship of two capabilities, to determine if one is a descendant of the other in the same map tree. This would be useful:

a) To determine revocability of one capability by another.

b) To determine accountability for unauthorized distribution of a capability.

This checking could easily be provided by an operation which simply scanned from the first capability's map entry to the root (base-level object) entry of the tree, watching for the second capability's map entry.

Another feature, which has been mentioned previously, would be the definition of other useful kinds of boxes in which to seal capabilities. For example, a box in which two or more capabilities could be sealed would eliminate the need for a small segment to act as the root of a compound representation of an extended object. This is similar to the scheme used in the HYDRA system [Wu 74]. On the other hand, its implementation would require variable-sized map entries, thus significantly complicating the implementation of the map.

A third rather interesting possibility is based on the observation that the masking of privileges by the revoke operation is not an intrinsically irreversible process. One could just as easily provide an "unrevoke" operation for restoring previously revoked privileges. Note that in this context, the use of locker boxes takes on a new significance, since it not only prevents inter-user interference, but also prevents the possessor of a capability from restoring privileges which have been revoked from it. The only major implementation difficulty with this feature is the impossibility of automatically deleting totally revoked entries from the map, since they may later have their privileges restored. This would require explicit deletions of map entries, making the appearance of the mechanism more complex. In addition, the whole notion of unrevoking privileges cannot be described cleanly in terms of the family tree model. Nevertheless, this feature could be quite useful, since it allows increased levels of trust between domains without necessitating the inconvenient repetition of the capability distribution procedure. The whole notion of temporary revocation could be quite useful, for example, in the debugging of locking protocols in a complex multi-process data-base system.

The fourth possibility is similar to the previous one in the sense that it attempts to preserve an established pattern of distributed capabilities while changing the meaning of those capabilities. In this case, the change is to allow switching of the contents of an extender box. This would enable a layer implementing an extended object to dynamically change the identity of its representation. Of course, care must be taken to avoid the possibility of circularities in the map; this can easily be done by using the first extension mentioned above to detect the case in which the new representation is a descendant of the extender which is being modified and signal an error.

The extensions described in this section could be added to NCS without excessive difficulty, but for the sake of clarity, the remainder of this thesis will assume that only the mechanisms originally described in Section 3.4 are provided. The facilities described in Chapter 4 would require some modification if any or all of the extensions were in fact included.

Chapter 4

Two Facilities Using the New Capability System

4.1 Possible Facilities Using Generalized Sealing

The purpose of this chapter is to briefly explore two examples of helpful facilities which can be constructed using the NCS generalized sealing mechanism described in Chapter 3. One is an improvement to the base-level domain-call machinery providing selective revocation of capability parameters passed on a call when the corresponding return occurs. The other is an extension providing a new type of object called a directory, which allows storage and distribution of capabilities in a manner which is often much more convenient than that provided by the base-level system.

Other useful facilities could also be defined in a similar fashion. Plausible examples might include:

a) An interprocess communication facility providing extended objects called message channels, capable of transmitting messages containing capabilities valid only until the next message is received.

b) A rental mediation service, guaranteeing to the lessor that privileges will be revoked upon contract expiration, and to the lessee that revocation cannot occur before that time.

These and other possibilities will be left unexplored here. The point is simply that the nested capability scheme allows the construction of an open-ended set of extensions, many of which can also make use of the revocation properties provided.

4.2 Revocable Parameters

There are certain events which constitute natural points at which to distribute and revoke capabilities. The most obvious examples are the occurrence of a domain-call and the subsequent corresponding return. As discussed by Schroeder [Sc 72], the temporary granting of access to parameter objects is a natural and useful feature of calls between mutually suspicious domains. There are other situations, however, in which it is unnecessary or even inappropriate to revoke all capability parameters when a return occurs. In particular, as previously noted, calls to trusted machine-extension domains need not revoke their parameters, which can result in substantial savings. We therefore propose a more general mechanism in which the caller can specify, for each parameter passed, whether it is to be revoked when the called domain returns.

It would probably be possible to provide this improved domain call as an extension rather than an integral part of the base-level system. This would require that all domain-calls and returns (or at least all those which involved any revocable capability parameters) be routed through this extension, which would be both clumsy and costly. We therefore describe revocable parameters as being included in the base-level domain-call mechanism.

In the previous discussion of parameter passing in Chapter 2, we found it unnecessary to specify the details of the copying of capabilities from the caller's address space to the callee's address space. In discussing the modifications necessary to provide revocable parameters, we continue in the same fashion, describing the implementation of parameter passing in terms of the get_parameter and put_parameter operations used in the discussion of TCS in Section 2.2.

When a domain call occurs, the caller controls parameter revocation by passing a Boolean vector R as an extra parameter, each element of which specifies whether the corresponding parameter should be revoked upon return. The call thus has the form:

$$\text{Call } (C_G, P_1, P_2, \ldots, P_n, R)$$

where R[i] controls the revocation of P_i.

Revocation of parameters is implemented using the same push-down stack which saves the return gate used to reactivate the calling domain when the callee returns. Thus, instead of just a gate capability, each domain-call corresponds to a packet of information as shown in Figure 4.2-1. The first item is N_R, which is the number of capability parameters to be revoked, and the last item is the return gate. Between them are the N_R capabilities which will be revoked when the return occurs. Figure 4.2-2 depicts the domain-call operation, and resembles Figure 2.2-2 which shows the TCS version. The differences comprise the steps necessary to save the extra information in the stack. Each revocable capability parameter is sealed in a revoker box; one copy of the sealed capability C is passed to the callee, and another is retained in the stack. The discipline followed is thus that of Figure 3.5-2; sealing of the callee's parameter in a locker is not necessary, since it is not received by any other domain. Figure 4.2-3 depicts the domain-return operation, as compared with the TCS version in

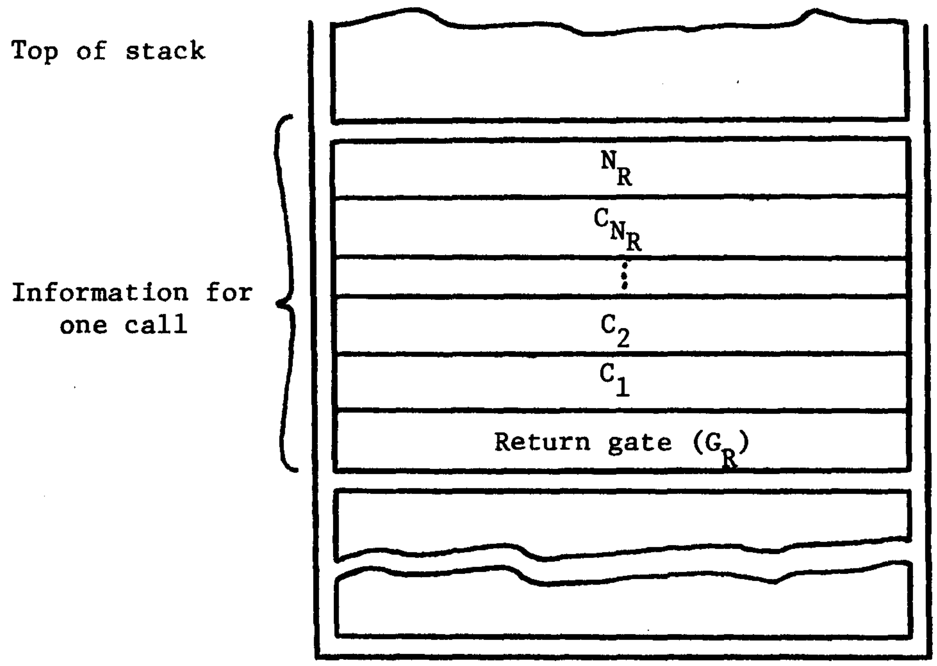
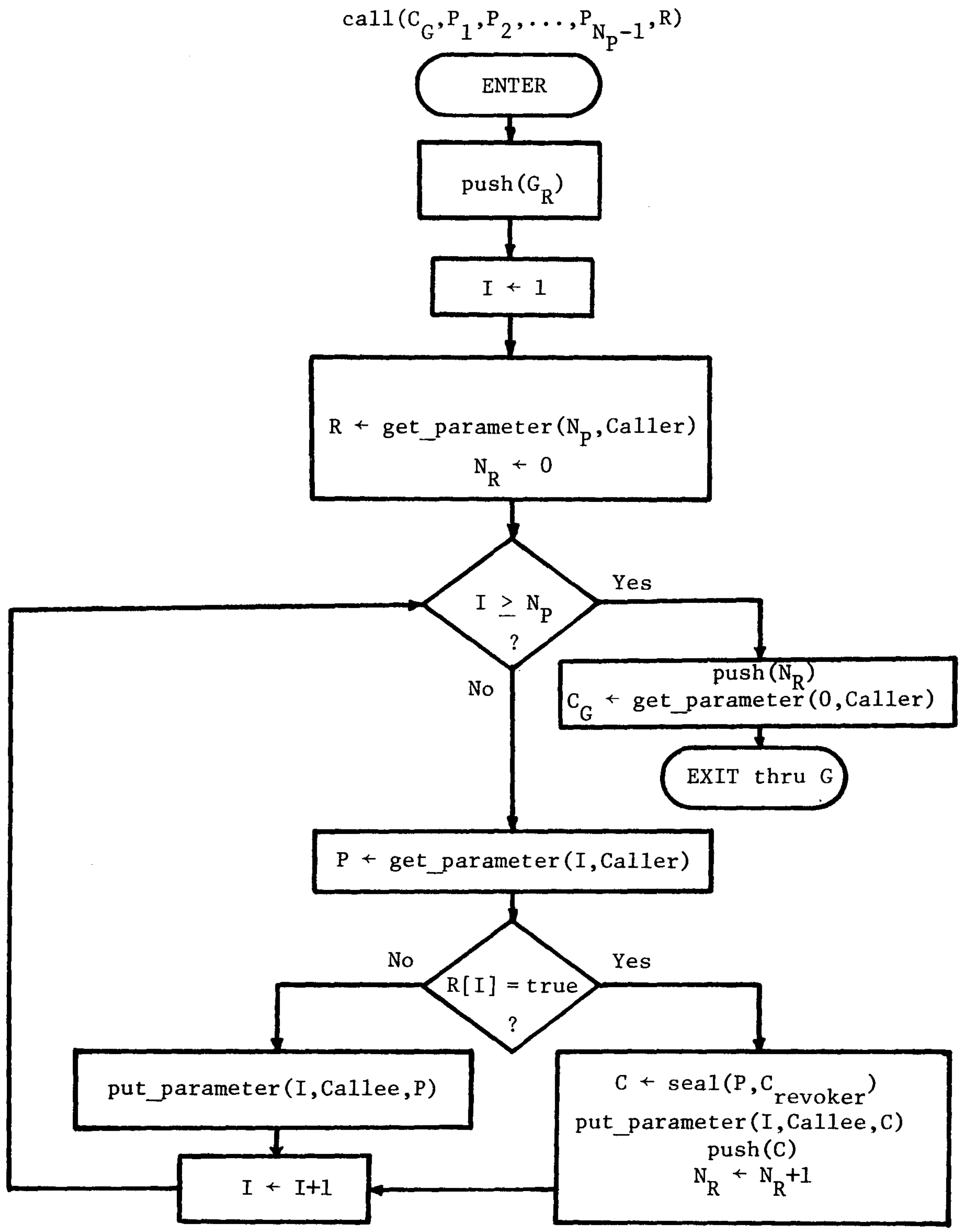


Figure 4.2-1: Parameter revocation data in stack



call(C_G,P_1,P_2,...,P_{N_P-1},R)
ENTER
push(G_R)
I ← 1
R ← get_parameter(N_P,Caller)
N_R ← 0
I ≥ N_P ?
Yes
push(N_R)
C_G ← get_parameter(0,Caller)
EXIT thru G
No
P ← get_parameter(I,Caller)
R[I] = true ?
No
Yes
put_parameter(I,Callee,P)
C ← seal(P,$C_{revoker}$)
put_parameter(I,Callee,C)
push(C)
N_R ← N_R+1
I ← I+1

Figure 4.2-2: NCS domain-call operation

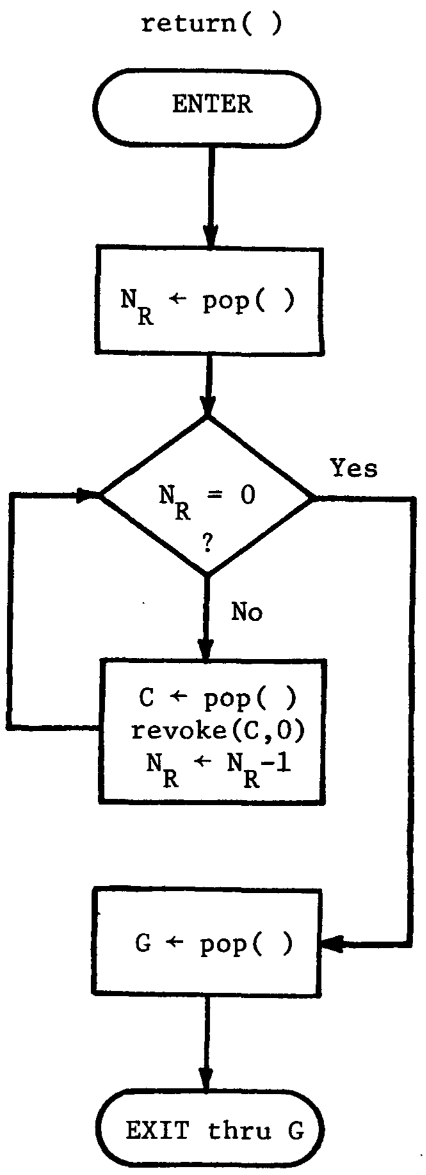


Figure 4.2-3: NCS domain-return operation

Figure 2.2-3. The added steps use the information in the stack to revoke the appropriate capabilities from the callee before retrieving the return gate and returning control to the caller. Note that the revocation is total, and thus releases map entries in an orderly way, as discussed in Section 3.7.

4.3 Directories

The notion of a directory, catalogue, or name-table mapping symbolic object names into some form of internal object pointer has appeared in most operating systems. The idea of a large collection of directories arranged in a tree-structured hierarchy originated mainly with the Multics system [Da 65], and has been adopted in several other systems since that time [St 73, Co 72, Ri 74].

A directory consists of a variable number of entries, each containing a different symbolic name and a pointer to an object (plus other information to be discussed shortly). The assumption that a unique directory entry is created with each object,* combined with the fact that directories are themselves objects, induces a tree-structured hierarchy on the set of all objects in existence at any time. The internal nodes are the directories and the leaves are the objects of other types. Concatenating the names of all entries along the path from the root directory to a given object yields the tree name of that object which uniquely identifies it.

The global tree-structured view of the universe of objects

*Except the pre-defined "root" directory.

can be useful in several contexts, such as system backup and recovery, accounting, and, as described below, in solving the "lost object problem," but it is often more convenient in other contexts to modify this view in two ways:

a) To allow the establishing of several directory entries for the same object.

b) To allow general _path names_ which can be interpreted as starting in any directory, rather than only the root-directory.

Both of these features can be added without disturbing the underlying tree-structure, as long as the extra entries ("_links_") in (a) can be distinguished from the original entries ("_branches_") when this is desired. This treatment of links as being full-fledged directory entries, contrasts with the Multics approach in which links are merely a re-naming device and have no protection significance. We choose this approach to facilitate subletting of rented objects.

In addition to naming, the directory system is useful for purposes of access control. Attaching an _access list_ to each directory entry aids in the orderly distribution of privileges to access shared objects. Each entry in the access list contains a pair

(lock, privileges)

which allows any possessor of a key matching the lock to obtain the corresponding privileges. (Of course, the specification of the access list, like the creation and deletion of entries, represents an access to the directory itself, and must also be controlled.) The simplest example of a lock would be a user name. A more sophisticated version of this is the "principle identifier" used in Multics [Sa 74], which is a kind of three-dimensional user name with more complicated rules for matching locks with keys. An even more flexible scheme will be described below. Note that in all such schemes, a user may <u>not</u> invent his own key(s), but may invent any locks he chooses and apply them to his objects, as discussed by Lampson [La 69].

In non-capability-based systems, directories are usually implemented as base-level objects [Or 72, Ri 74], since their access lists are generally used as the system's primary protection facility. In a capability-based system, however, directories can be implemented as a higher-level extension, providing symbolically named "pigeon holes" for the storage and dissemination of capabilities [Fa 68]. This is an attractive organization, since it removes from the base-level system all handling of symbolic names and the corresponding variable-sized data structures. From the point of view of the base-level system, the directory layer is simply another user domain, although, of course, it must be regarded as a trusted machine extension by normal user programs which store their capabilities in directories. The desirability of providing both directories and capabilities in the same system is convincingly argued by Lampson [La 69].

The directory layer described below provides for storage of any number of capabilities in each directory, one per entry. Attached to each entry is an access list authorizing a domain to obtain a sealed copy of the stored capability by executing

$$C \leftarrow \text{lookup } (C_D, \text{Name}, C_K)^{*}$$

where C_D is a capability for the directory (authorizing lookup access), Name is a character string, and C_K is a <u>key</u> capability. The unique ID of the key capability is matched against the locks in the access list of the entry and the corresponding privileges are returned in C. Subsequent reduction of the privileges authorized to holders of key C_K will retroactively reduce the privileges in C, using the underlying revocation machinery. (Various conditions, such as failure to find an entry with the given name, or failure to find a lock in the access list which matches the key C_K cause errors to be signalled and no capability to be returned.) The use of freely distributable capabilities as the keys authorizing directory lookups allows the users to flexibly and economically establish any group authorization scheme desired by simply passing keys to each other. Neither the base-level system nor the directory layer need take any explicit notice of such groups [La 69, St 73]. More complicated facilities such as path name lookup [Da 65], multiple directory searching [Or 72, St 73] and automatic lookup on first use of a symbolic name [Da 68] could be implemented in terms of this basic lookup primitive; these will not be discussed here.

In such a directory system, there is no intrinsic distinction

---

*In terms of base-level operations, this would be written

$$C \leftarrow \text{call } (C_G, C_D, \text{Name}, C_K)$$

where C_G is a capability for a gate into the directory layer corresponding to the lookup operation.

between the various directory entries containing capabilities for a given object. For the reasons cited previously, however, it is useful to distinguish one of the entries as a branch and consider the others to be links. In particular, one can solve the lost object problem by guaranteeing that the branch exists for at least as long as the object. This is accomplished by creating the object and the branch simultaneously, and having the directory system, upon removing the branch from the directory, delete the object (if it still exists).

The use of branches to solve the lost object problem is relatively straightforward in the case of base-level objects and directories. By performing the creation of all such objects through calls on the directory layer which also create a directory branch, one can insure the existence of a branch for each new object. When the branch is removed, the object can be destroyed by the directory layer, either internally (in the case of directories) or by calling the appropriate operation (in the case of base-level objects).

In the case of extended objects, however, the situation is more complicated, for two reasons:

a) It is inappropriate for the directory layer to have embedded in it any knowledge of (e.g. calls on) higher layers.

b) New higher level extended types can be defined at any time.

These considerations render impossible the creation of such objects via the directory layer, and necessitate a more circumspect approach to their deletion when a branch is removed.

When a higher layer creates an extended object X and wishes to take advantage of the directory system to keep X from becoming lost, it can do so by executing

$$\text{make\_branch}\ (C_D,\ \text{Name},\ C_X,\ C_G)^{*}\quad .$$

This creates an entry in the directory indicated by C_D. The entry has name Name and contains C_X, a capability for the new object. In addition, the entry holds C_G, a capability for gate G into the caller (i.e. the layer implementing the object). When the branch is later removed from the directory, the directory system guarantees to execute

$$\text{call}\ (C_G, C_X)\quad .$$

The gate G should correspond to the deletion operation for objects of the extended type, hence this is equivalent to

$$\text{delete}\ (C_X)\quad .$$

Of course, it is the responsibility of the layer implementing X to insure that this call does in fact result in the deletion of X. The directory layer's only concern is that it must be prepared for anything which may happen between the time it performs the call

*Repeated use of the make_branch operation specifying the same object X would cause the directory structure to fail to be a tree. This might be of concern to layers at or above the level at which X was implemented (although it certainly would cause no trouble for the directory layer). The layer implementing the object could protect itself from this situation if the make_branch operation were modified to require an extra parameter C_T, a capability for the type of X, as authorization to make a branch for X.

and the time the callee returns. This could include various types of errors, blocking of the process, and even further calls on the directory layer. The straightforward way to handle this is simply to have the directory layer complete its part of the branch removal and then exit to the object deletion operation via a jump-call as described in Section 2.2.*

It might appear that the calling of the higher layer object deletion operation by the directory layer violates the ordering constraints of layered system construction. This is not really the case, however, since this call does not represent any knowledge of the higher layer embedded in the directory layer. Such "blind" upward calls are quite similar to hardware "traps" or "exceptions."

The other directory layer operations of interest are:

make_link (C_D,Name,C_X)

remove_entry (C_D,Name)

set_lock (C_D,Name,L,P)

$C_K \leftarrow$ create_key ( )

create_directory (C_D,Name)

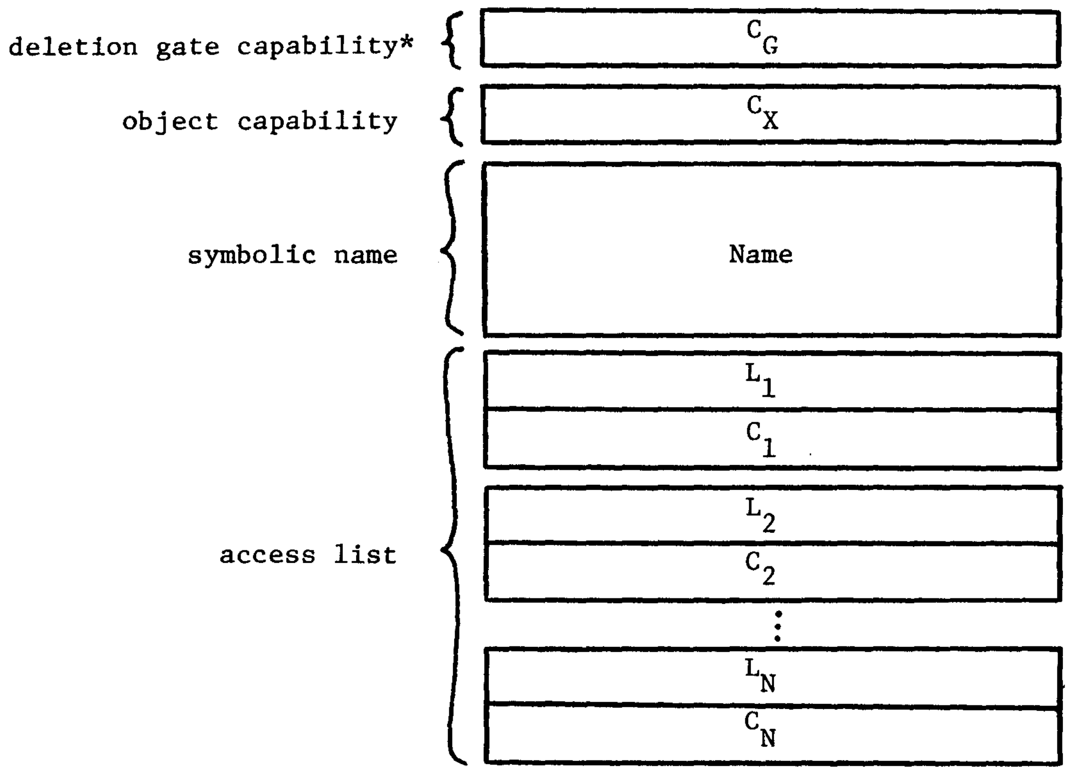
delete_directory (C_D)

The make_link operation establishes a new entry in directory D, containing C_X and named Name. The remove_entry operation removes a link or a branch. In the latter case, it performs object destruction as described above. The set_lock operation establishes a new lock on the named entry in directory D. The lock is L (i.e. it can be opened using a key with capability-ID = L)

*We ignore the extra complications involved if object deletion is allowed to fail.

and it confers the set of privileges P. The create_key operation simply returns a capability of type 'key' with a new unique capability-ID. The create_directory operation establishes a new empty directory as a son of directory D (i.e. pointed to by a new branch in D with name Name). The delete_directory operation deletes the directory D. This requires removal of all entries from D, including any branches for other directories which must thus be deleted, and so on. In other words, the entire subtree rooted in D must be traversed and deleted. This complication is best postponed until a higher level utility program, hence the directory layer can simply refuse to delete a non-empty directory.

The implementation of directories as described is relatively straightforward. Each directory is represented as a segment, containing entries formatted as in Figure 4.3-1. The original capability C and the entry name are present when the entry is first created, along with the deletion-gate capability in the case of a branch. Subsequent use of the set_lock operation proceeds as shown in Figure 4.3-2. First the lock is added to the access list if not already present, together with a capability to hold the privileges corresponding to the lock. This capability is created by sealing the original capability C_X in a revoker box. Then the privileges in the capability are revoked down to the desired level. Note that in the case of applying the set_lock operation to an already existing lock, any outstanding capabilities previously obtained via that lock using the lookup operation will also have their privileges revoked. Finally, if the revocation was total



*in branches only

Figure 4.3-1: A directory entry

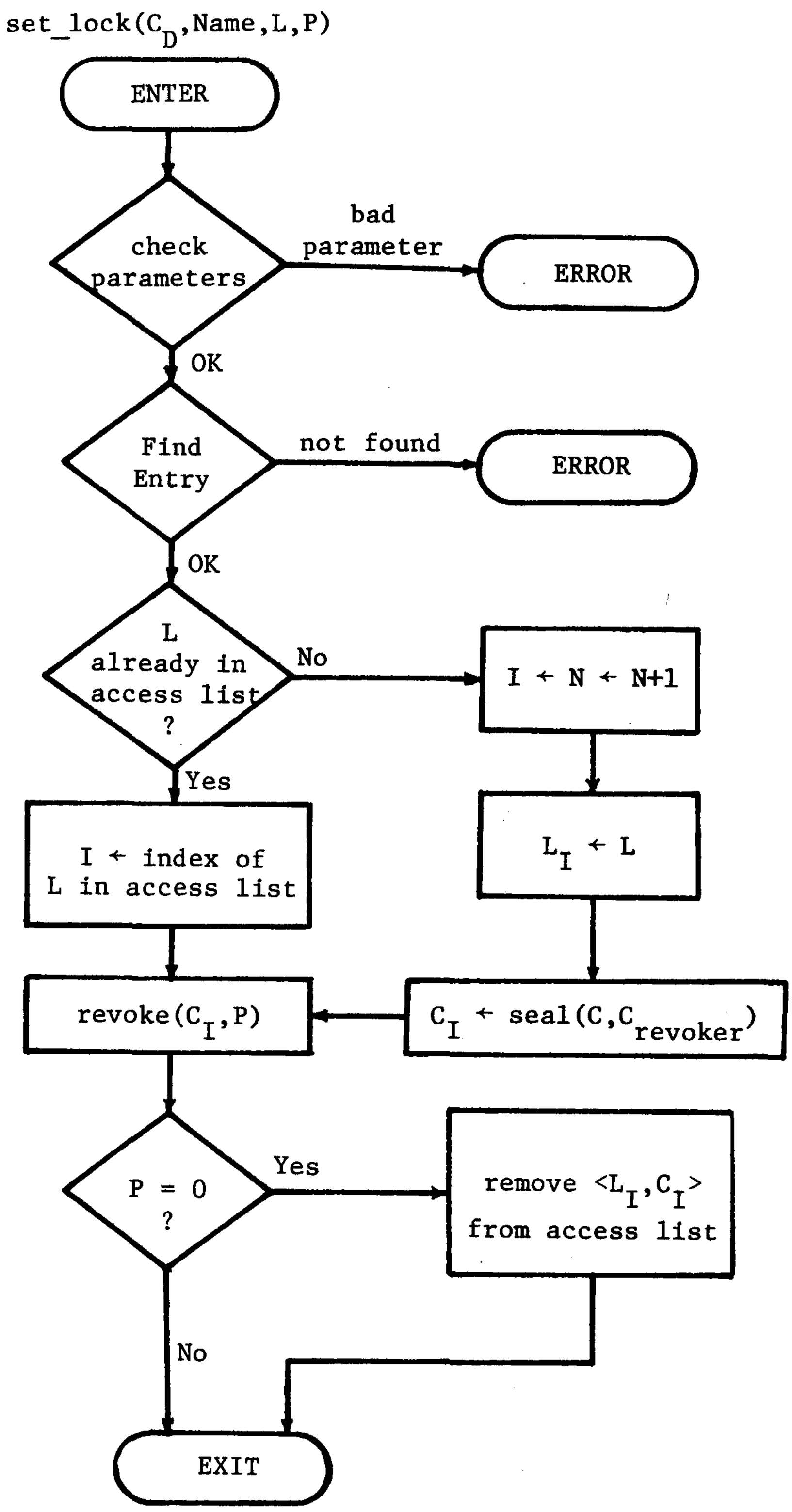


Figure 4.3-2: The set_lock operation

(i.e. $P = 0$), the lock is deleted from the access list. (Such total revocation is also performed on each lock in the access list when the entire directory entry is removed. This is another example of orderly use of the underlying map machinery, as discussed in Section 3.7.)

The lookup operation, upon finding the named entry, searches the access list for a lock matching the proffered key. If one is found, the corresponding capability is sealed in a locker box and returned to the caller. Thus, the net result of the set_lock and lookup operations is distribution of capabilities following the discipline of Figure 3.5-5.

The create_key operation is quite simple to implement. It would be nicely captured by the simple sealing of an empty extender box. Lacking this facility, the directory layer can simply seal any handy capability, since only the external appearance of the new key capability is significant.

The directory layer just described is probably the best example of the kind of useful extensions which can be constructed using the NCS nested capability mechanism. It provides extremely useful features for the users of the system, yet its implementation is rendered relatively simple by the power of the underlying base-level naming and protection facilities.

Chapter 5

Summary and Conclusions

5.1 Summary

This thesis has discussed integrated naming and protection mechanisms for computer systems, providing protected names called capabilities which both identify an object and authorize access to it. A major advantage of capabilities is the flexibility provided by their being freely copyable. A corresponding disadvantage in existing capability systems has been the difficulty of revoking previously distributed capabilities. The main result of this thesis has been the design of a capability system providing both free distribution and orderly revocation of capabilities. Various approaches to this problem were discussed in Chapter 2, culminating in a set of goals to be met by a new design. The generalized capability sealing mechanism of Chapter 3 was shown to meet these goals, providing selective revocation of capabilities, as well as a flexible type extension facility. A possible implementation of the design was discussed in sufficient detail to demonstrate its practicality. Various possible elaborations on the design were also discussed. Chapter 4 described two facilities applying revocable capabilities to the needs of users in specific ways.

5.2 An Area for Further Research

In terms of the facilities provided, the naming and protection mechanisms described in this thesis appear to be a sound basis upon which to build a secure and flexible user environment. In particular, the provision of revocable capabilities eliminates one of the main objections often made to capability-based designs [Sc 72], thus making the proposed design applicable in a wider class of situations. One could thus characterize the thrust of this thesis as an attack on the _flexibility_ aspect of the protection problem.

On the other hand, the thesis does not make any direct attack on another more general aspect of the protection problem which one might call the _comprehensibility_ of protection mechanisms. Experience indicates that protection mechanisms which are confusing to users are likely to be misused, or even go unused [Sa 74, Sc 72]. Even the user who correctly applies a confusing protection feature may feel no great confidence that it enforces his intentions. There are at least three ways in which protection systems can be confusing:

a) They can be based on a disorderly set of separate but interacting mechanisms.

b) The relevance of the mechanisms to specific situations can be obscure.

c) The correspondence between global state of the protection machinery and the desires of the users can be difficult to assess.

A fair amount of progress has been made on problem (a). The early proliferation of _ad hoc_ protection mechanisms was a major motivation for the original development of capabilities [DVH 66], as well as later more abstract treatments by Lampson [La 71], Jones [Jo 73], and others. On the other hand, strict minimization of the set of primitives will not necessarily clarify the description, especially since it may exacerbate problem (b). For example, our unification of privilege revocation and type extension in a single mechanism, while interesting in itself, may or may not represent a net increase in the comprehensibility of the design.

Problem (b) is caused by the gap -- often quite broad -- between the concerns of the human users and the mechanisms provided by the protection system, in terms of which they must express those concerns. Of course, the user need not deal only with the protection primitives of the system; various extensions, such as those mentioned in Chapter 4, can be provided. These do not go far, however, in attempting to capture the interactions between users seen in the larger social context. This is due in part to the imprecision of many legal and social principles, resulting from their implicit reliance on the reasonable judgement of the parties involved, a characteristic sadly lacking in most computers. Much work remains to be done in mapping such principles into the protection primitives of computer systems [Ro 74, Pe 74, Tu 74].

Problem (c) is perhaps the most difficult of the three. During our discussion of capability mechanisms, we emphasized the desirability of allowing distribution and revocation of capabilities without requiring global knowledge of such propagation on the part of the participants. Such global knowledge is sometimes desirable for its own sake, however. Moreover, even if the entire state of the protection machinery is visible (which can itself raise serious questions of privacy), the full significance of that state cannot be assessed without knowledge of the levels of trust and suspicion between the various possessors of access privileges. This appears to be a very fundamental problem, and it is not clear what approach (if any) will prove fruitful in dealing with it.

5.3 The Future of Protection

Much work remains to be done in the area of protection. In the long run, protection will contribute to the development of generally available computer utilities in at least three ways:

a) By facilitating the development of extremely large software systems, such as sophisticated service programs, and the operating system of the computer utility itself.

b) By protecting the investments of users who develop large proprietary programs and/or data bases, thus providing a suitable marketplace for such services.

c) By enforcing social controls on the dissemination of stored information.

Given the difficulty and importance of the problems to be solved protection promises to be an active area of research for many years to come.

References

[BCD 72] Bensoussan, A., Cingen, C.T. and Daley, R.C., "The MULTICS virtual memory: concepts and design," Communications of the Association for Computing Machinery, Vol. 15, No. 5 (May 1972), pp. 308-318.

[Bo 67] Bobrow, D.G. and Murphy, D.L., "Structure of a LISP system using two-level storage," Communications of the Association for Computing Machinery, Vol. 10, No. 3 (March 1967), pp. 155-159.

[Bu 61] Burroughs Corporation, "The descriptor -- a definition of the B5000 information processing system," Detroit, Michigan (1961).

[CC 69] Computer Center, University of California, Berkeley, Cal-TSS Users Guide (1969).

[CV 65] Corbato, F.J. and Vyssotsky, V.A., "Introduction and overview of the MULTICS system," AFIPS Conference Proceedings 1965 Fall Joint Computer Conference, Vol. 27, pp. 185-196.

[Co 72] Cosserat, D.C., "A capability oriented multiprocessor system for real-time applications," ICC Conference, Washington, D.C. (October 1972), 8 pp.

[Da 65] Daley, R.C. and Neumann, P.G., "A general purpose file system for secondary storage," Proceedings AFIPS 1965 Fall Joint Computer Conference, Vol. 27, Pt. I, AFIPS Press, Montvale, N.J., pp. 213-230.

[Da 68] Daley, R.C. and Dennis, J.B., "Virtual memory,processes, and sharing in MULTICS," Communcations of the Association for Computing Machinery, Vol. 11, No. 5 (May 1968), pp. 306-313.

[DF 65] David, E.E. and Fano, R.M., "Some thoughts about the social implications of accessible computing," AFIPS Conference Proceedings 1965 Fall Joint Computer Conference, Vol. 27, pp. 243-247.

[DVH 66] Dennis, J.B. and Van Horn, E.G., "Programming semantics for multiprogrammed computations," Communications of the Association for Computing Machinery, Vol. 9, No. 3 (March 1966), pp. 143-155.

[De 65] Dennis, J.B., "Segmentation and the design of multiprogrammed computer systems," Journal of the Association for Computing Machinery, Vol. 12, No. 4 (October 1965), pp. 589-602.

[De 68] Dennis, J.B., "Programming generality, parallelism, and computer architecture," Proceedings IFIP 1968, North Holland, Amsterdam, pp. C1-7.

[Di 68] Dijkstra, E.W., "Cooperating Sequential Processes," in Programming Languages (F. Genuys, ed.), Academic Press (1968), pp. 43-112.

[Di 68b] Dijkstra, E.W., "The structure of the THE multiprogramming system," Communications of the Association for Computing Machinery, Vol. 11, No. 5 (May 1968), pp. 341-346.

[En 72] England, D.M., "Architectural features of System 250," Infotech State of the Art Report on Operating Systems (1972), 12 pp.

[Fa 68] Fabry, R.S., "Preliminary description of a supervisor for a machine oriented around capabilities," ICR Quarterly Report 18 (August 1968), ICR, University of Chicago.

[Fa 74] Fabry, R.S., "Capability-based addressing," Communications of the Association for Computing Machinery, Vol. 17, No. 7 (July 1974), pp. 403-412.

[Fe 73] Feustal, E.A., "On the advantages of tagged architecture," IEEE Transactions on Computers, Vol. C-22, No. 7 (July 1973), pp. 644-656.

[Fr 74] Frankston, R.M., "The computer utility as a marketplace for computer services," Project MAC Report MAC-TR-128 (1974).

[Gr 71] Graham, G.S., "Protection structures in operating systems," M.S. thesis, University of Toronto (1971).

[Gr 72] Graham, G.S. and Denning, P.J., "Protection - principles and practice," Proceedings AFIPS 1972 Spring Joint Computer Conference, Vol. 40, AFIPS Press, Montvale, N.J., pp. 417-429.

[Gr 73] Gray, J.N., IBM San Jose Research Laboratory, private communication.

[Ha 70] Brinch Hansen, P.~~B~~., "The nucleus of a multiprogramming system," Communications of the Association for Computing Machinery, Vol. 13, No. 4 (April 1970), pp. 238-250.

[HEW 73] U.S. Department of Health, Education, and Welfare, "Records, computers and the rights of citizens," Report of the Secretary's Advisory Committee on Automated Personal Data Systems, Washington, D.C. (July 1973).

[HP 73] Hoare, C.A.R. and Perrott, R.H., _Operating Systems Techniques_, Academic Press, New York, N.Y. (1973).

[Jo 73] Jones, A.K., "Protection in programmed systems," Ph.D. thesis, Carnegie-Mellon University (1973).

[La 69] Lampson, B.W., "Dynamic protection structures," _Proceedings AFIPS 1969 Fall Joint Computer Conference_, Vol. 35, AFIPS Press, Montvale, N.J., pp. 27-38.

[La 69b] Lampson, B.W., "An overview of the CAL timesharing system," Computer Center, University of California, Berkeley (1969).

[La 71] Lampson, B.W., "Protection," _Proceedings 5th Annual Princeton Conference_, Princeton University (March 1971), pp. 437-443.

[La 73] Lampson, B.W., "A note on the confinement problem," _Communications of the Association for Computing Machinery_, Vol. 16, No. 10 (October 1973), pp. 613-615.

[La 74] Lampson, B.W., "Redundancy and robustness in memory protection," _Proceedings IFIP 1974_, North Holland, Amsterdam, pp. 128-132.

[Li 73] Lindsay, B.G., "Suggestions for an extensible capability-based machine architecture," International Workshop on Computer Architecture, Grenoble, France (June 1973).

[Mo 72] Morris, J.H., "Authentication tags: the proper division of hardware/software responsibility" (1972), unpublished.

[Mo 73] Morris, J.H., "Types are not sets," ACM Symposium on Principles of Programming Languages, Boston, Mass. (October 1973).

[Ne 72] Needham, R.M., "Protection systems and protection implementations," _Proceedings AFIPS 1972 Fall Joint Computer Conference_, Vol. 41, AFIPS Press, Montvale, N.J., pp. 571-578.

[Neu 74] Neumann, P.G. et al, "On the design of a provably secure operating system," Working Paper, IRIA International Workshop on Protection in Operating Systems, Paris (August 1974).

[Or 72] Organick, E.I., _The MULTICS System: An Examination of its Structure_, The MIT Press, Cambridge, Mass. (1972).

[Pa 72] Parnas, D.L., "On the criteria to be used in decomposing systems into modules," _Communications of the Association for Computing Machinery_, Vol. 15, No. 12 (December 1972), pp. 1053-1058.

[Pe 74] Peuto, B.L., "Comparative study of real estate law and protection systems," Ph.D. thesis, University of California, Berkeley (1974).

[Po 74] Popek, G.J., "Protection structures," Computer, Vol. 7, No. 6 (June 1974), pp. 22-33.

[Ri 74] Ritchie, D.M. and Thompson, K., "The UNIX time-sharing system," Communications of the Association for Computing Machinery, Vol. 17, No. 7 (July 1974), pp. 365-375.

[Ro 74] Rotenberg, Leo J., "Making computers keep secrets," Ph.D. thesis, M.I.T. (1974), Project MAC Report MAC-TR-115.

[Sa 66] Saltzer, J.H., "Traffic control in a multiplexed computer system," Ph.D. thesis, M.I.T. (1966), Project MAC Report MAC-TR-30.

[Sa 74] Saltzer, J.H., "Protection and the control of information sharing in MULTICS," Communications of the Association for Computing Machinery, Vol. 17, No. 7 (July 1974), pp. 388-402.

[Sc 71] Schroeder, M.D., "Perfomance of the GE-645 associative memory while MULTICS is in operation," Proceedings Workshop on System Performance Evaluation, Cambridge, Mass. (1971), pp. 227-245.

[Sc 72] Schroeder, M.D., "Cooperation of mutually suspicious subsystems in a computer utility," Ph.D. thesis, M.I.T. (1972), Project MAC Report MAC-TR-104.

[SS 72] Schroeder, M.D. and Saltzer, J.H., "A hardware architecture for implementing protection rings," Communications of the Association for Computing Machinery, Vol. 15, No. 3 (March 1972), pp. 157-170.

[St 73] Sturgis, H.E., "A postmortem for a timesharing system," Ph.D. thesis, University of California, Berkeley (1973), Xerox PARC Technical Report 74-1.

[Tu 74] Turn, R., "Privacy and security in personal information databank systems," Rand Report R-1044-NSF (1974), Rand Corporation, Santa Monica, Calif.

[Wu 74] Wulf, W. et al, "HYDRA: the kernel of a multiprocessor operating system," Communications of the Association for Computing Machinery, Vol. 17, No. 6 (June 1974), pp. 337-345.