A Microprogrammed Operating System Kernel

Andrew James Herbert

St. John's College

A dissertation submitted for the degree of Doctor Of Philosophy in the University Of Cambridge, November 1978.

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of any work done in collaboration.

Thesis Summary

"A Microprogrammed Operating System Kernel"

Andrew James Herbert

St. John's College

The subject of the thesis is the design and implementation of an operating system kernel for the Cambridge Capability Computer (CAP). The kernel of an operating system is its most primitive level of facilities and forms the foundation stone around which the rest of the system is structured.

The particular emphasis of the CAP kernel is concerned with protection - the control of access to information. The kernel uses the notion of capabilities to provide a flexible and controlled mechanism for the sharing of information within a computer system. The protection mechanisms include provision for the efficient control of access to memory as well as facilities for handling abstract resources like files and virtual peripherals. The kernel allows the introduction of new types of resources in addition to the basic set of hardware resources to permit user extension of the system. Attention is given to the problem of recall of privilege or revocation in capability systems and the kernel includes operations for both permanent and temporary revocation of particular access rights to information in a selective manner.

In the past many of these functions have only been found in kernels implemented in user-level software which are frequently cumbersome and inefficient. An examination is made of why this should be and how efficiency and simplicity can be gained by a microprogrammed implementation. The thesis draws on the experience of a number of software kernels to discover the various design decisions that have to be made and the techniques that may be used to implement a successful kernel.

The feasibility of the design arrived at by considering these issues is demonstrated by describing its implementation on the Cambridge Capability Computer in terms of the primitives provided and the internal organisation of the proposed kernel. In an evaluation, the kernel is examined in the light of the analysis of other kernels to point out its strengths and weaknesses and to gain insights into the utility of the design as a practical operating system kernel.

PREFACE

I am indebted to my supervisor, Dr Roger Needham, for his encouragement of my research and also to other members of the CAP Project, especially Dr Andrew Birrell, Dr Robin Walker and Martyn Johnson, for their helpful advice and constructive criticisms during all of the stages of my work.

The Computer Laboratory Of The University Of Cambridge, headed by Professor M.V. Wilkes, furnished many facilities, including use of the CAP computer and, in conjunction with St. John's College Cambridge, provided funds enabling me to visit the United States Of America in my final year which proved to be a valuable and stimulating experience.

The CAP Project as a whole and my own work in particular were supported by the Science Research Council during the period that the research was carried out.

Finally I wish to acknowledge my debt to my wife Jane, both for her support and forbearance while I was engaged in research and more recently for her help in typing this thesis.

CONTENTS.

CHAPTER ONE.

INTRODUCTION.

1.1. Overview.

The subject of this thesis is the design and implementation of a microprogrammed operating system kernel. The kernel of an operating system is the most primitive level of the system and forms the foundation of the rest of its structure and so it is important that the kernel is well-designed and efficient if the entire system is to be successful.

An operating system can be considered as a mapping between rudimentary hardware resources and the advanced facilities of an abstract machine. The duties of a kernel include the provision of mechanisms for performing this mapping and also making abstract resources such as files and virtual memory available to users through some addressing and access control schemes. The cost-effectiveness of an operating system can be judged by weighing the benefits of an elegant abstract machine, such as ease of use, against its cost, both in terms of development and the machine cycles consumed in its operation. The effectiveness of an entire system will be greatly impaired by any weaknesses or inefficiencies in the kernel and this is the motivation for investigating the structure and organisation of operating system kernels.

The major concerns of a kernel are protection (the control of access to information by programs running within a computer system), multiprogramming, I/O control and fault handling. This thesis is primarily concerned with the protection aspects of kernel design and the influence of protection on the other kernel functions. A protection architecture which permits flexible and controlled sharing of information by all programs, including those that make up the operating system will be described in the latter part of this thesis.

Many operating system kernels have been implemented in software rather than microprogram. Normally they execute in a

highly privileged special state or supervisor mode in which powers are available that allow the contents of processor descriptor registers, and so on, to be modified. The close relationship between the kernel and the underlying hardware means that it is usual to find that the kernel has been written as a large monolithic assembly code program because of the difficulties of writing compilers to generate compact fast code that interfaces in a straightforward fashion with the hardware primitives used. In consequence, such kernels are difficult to verify and debug, even if great care is taken to ensure that code is written and tested in a well-structured and organised way.

In general, a software kernel preserves a lot of information about the state of the hardware so that it may decide what action to take in response to a kernel call by a program. It is therefore necessary for the kernel to carry out all operations that involve modifying hardware protection registers and the like; even quite trivial functions have to be directed through the kernel so that changes to the state of the hardware can be recorded. This will involve the considerable overhead of establishing kernel calls, which might include preserving and restoring the state of executing processes, and checking the arguments presented against tables of privileges before any function can be carried out. There is also a temptation, if an operating system service does little else beyond a series of kernel calls, to build it into the kernel and this further compounds the problem. Much of the clumsiness of a software kernel comes from a lack of intimacy with the hardware of the underlying machine that carries out the operations programmed in the kernel and this leads us to consider the use of a microprogrammed kernel because of the closer association between microprogram and the basic hardware.

Microprogramming is a long established technique for organising computer hardware [Wilkes 51]. Microprogramming has many properties that aid the development of machines which include complex operations in their instruction repertoire. A microprogrammable machine consists of a simple, fast microprogram processor with rudimentary logical, control and arithmetic

facilities that executes microinstructions held in a microprogram memory. The microprogram emulates user instructions by activating other parts of the machine in the correct sequence and arranging for the passage of data through the machine's registers. If the user machine is to be reasonably fast, the microprogram must be fetched from microstore and executed at high speed. In the past, when the cost of very fast memory was high, microprogram memories were very small, but now with falling hardware costs owing to large scale integration (LSI) technology, quite substantial memories of sufficient speed can be obtained for a reasonable price. This enables critical components of the software of a system to be put into microprogram to reap benefits in the areas of high level facilities and efficiency; for example, the GEC 4000 range of computers [GEC 72] have a microprogrammed nucleus that is responsible for the control of multiprogramming, synchronisation and inter-process communication.

There are other advantages apart from efficiency to be gained from microprogramming. A microprogrammed operating system kernel has a much greater degree of intimacy with hardware addressing and protection functions, interrupt handling and so on than a special state supervisor for a conventional machine. This makes it easier for the kernel to provide a powerful machine to users without the expense of kernel calls in software, because the microprogram is better placed to carry out access checking and argument verification cheaply as part of the normal hardware instruction decoding and addressing operations. Kernel functions can be encoded as a single instruction at the user level which means that interfaces tend to be simple in terms of a few arguments in registers and kernel calls are uniform with the hardware instruction and addressing formats. On the other hand, in a software kernel it is easy to make interfaces very complex and confusing for users.

It is an unfortunate fact of life that microprograms are on the whole harder to write and less easy to understand than assembly code programs. This is because microinstructions specify the operation of basic hardware components rather than the higher level logical functions expressed by machine instructions.

Furthermore, debugging is not straightforward as it requires use of the raw hardware for hands-on access which is wasteful of computer resources, whereas software systems can be tested concurrently on a time-shared system with other programs and can be written in high level languages. There is often insufficient room in microstore to include both a development microprogram and a debugging system, so that a lot of testing has to be carried out with just the aid of register display lights and control switches. To some extent these difficulties can be avoided by the use of a microprogram processor simulator, although it must be ensured that the simulation faithfully duplicates the hardware because microprogrammers are fond of taking advantage of hardware peculiarities and side-effects to save on instructions or time. If oddities are not carefully duplicated in the simulation, there is a strong likelihood that a microprogram will behave wrongly on the real machine.

Even in these days of large semiconductor memories, microprogram memories are usually of modest size, often only a few thousand bytes, which greatly restricts the amount that can be included in a microprogrammed kernel, whereas a software kernel may be many hundreds of thousands of bytes long. On the other hand, if there is microprogram memory space available, it is tempting to put more and more into the kernel on the grounds of efficiency, with the result that the kernel becomes cumbersome, unwieldy and much harder to debug fully and test. The key to microprogramming operating system kernels is the correct identification of those primitives that rightly belong in the kernel and those that should be left to software.

Microprogrammed machines may not be as fast as those built in hard-wired logic because of the overheads of fetching and decoding microinstructions. This is the penalty paid for the ability to implement highly complex functions and to change the nature of the machine by modifying its microprogram. Some speed can be recovered by providing functions such as user instruction decoding, address translation and protection checks in raw hardware and leaving the microprogram to handle more difficult things, at a loss of some flexibility in the range of

architectures that can be presented at the user level.

1.2. A Wider View of Protection.

The bulk of this thesis is concerned with mechanisms for protection [Lampson 71, Jones 73, Graham 68, Saltzer and Schroeder 75], which can be defined as the control of access to information by executing programs. Protection is just one facet of the overall issue of computer security [Ware 67, Anderson 73, Branstad 73, Hoffman 73]. In recent times, computers have been used more and more as repositories for large volumes of information of a confidential or proprietory nature, shared by a large community of users who are unwilling to trust one another not to steal or access private data. The social and legal implications of retaining information within a computer have generated a requirement for the formulation of policies governing the security of computer systems [U.S. Department of Health, Education and Welfare 73] and to implement such policies there is is a need for a technological framework within which it is possible to discuss and judge the security of a computer system.

Aspects of security include hardware reliability, secure communication between terminals and computers, authentication of access to machines and the physical security of the computing system hardware and ancilliary equipment such as magnetic tapes and discs. These external issues of security can be characterised by the property that it is not possible to achieve total security in any of them; instead the measures taken to enhance security can be judged only in terms of the cost-effectiveness of reaching some level of quantified optimism about the degree of security attained. The notion of the work factor involved in breaching the security measures is often used as an indication of the amount of effort that must be expended to defeat security, and in many cases to discourage deliberate attack it is sufficient to ensure that the cost of the work done exceeds the value of the information that is illegally obtained. A system is only as strong as its weakest component and if the security of the system is undermined at any point all other aspects of it including protection, are put at risk.

If an artificial view of the real world in which external security can be guaranteed completely is adopted, it is possible to describe protection mechanisms which enable positive and absolute statements to be made about the security of information within a computer system in terms of which programs may access information and change it. This approach is useful despite its divergence from reality because, if protection can be established within a computer system, it is only necessary to concentrate upon the external aspects of security, safe in the knowledge that the system cannot be subverted from within. The protection state of a computer system can be represented by an access matrix [Lampson 71] whose columns correspond to items of information and whose rows refer to programs as shown below:

program	item			
	A	B	C	D
a	R	R	-	RW
b	R	-	-	R
c	RW	-	-	-
d	RW	-	RW	R

Program a has R (read) access to items A,B,D and also W (write) access for D, but is unable to access item C at all. Indeed, item C can only be accessed by program d. Obviously, in a real life system, an access matrix is vast, with entries for many items of information and programs. In the design of a protection system it is necessary to look for some suitable representation of the information in the matrix. There are two main approaches: (i) access control list systems such as MULTICS [Organick 72] in which each item has associated with it an access control list that encodes the information in a column of the matrix, stating which programs are allowed access to the item, and (ii) capability systems [Dennis and Van Horn 66, Lampson 69 and Needham 72] where each program is given a set of tickets stating which privileges are possessed by the program in respect of each item; that is to say, a program's set of capabilities is an encoding of a row in the access matrix.

It is fairly straightforward to arrange that information belonging to one program cannot be accessed by another, but in

general it should be possible to allow information to be shared by programs with possibly differing degrees of privilege; for example, in the access matrix above, the item A is shared by programs a, b, c, d with read access, but only d has the ability to write to it. In a closed community, programs and data might be shared with no restrictions, in a spirit of free cooperation, but if the computer is shared by a general public with competing interests, users will not trust one another and the sharing of information will take on the nature of commercial bargaining in which each party to a transaction is suspicious of the others. Schroeder calls this mutual suspicion [Schroeder 72]. A protection mechanism must be able to support controlled sharing in this sort of environment. For example, it should be possible for one user to allow another to have use of a program performing some service, while at the same time not allowing him read access to the binary code of the program, so that it may not be stolen or misused.

The level of trust between users will change with time and the owner of a privilege may subsequently wish to restrict use of it, perhaps because a user has defaulted on the payment of a rental. The action of recalling a privilege is known as revocation. Revocation may be temporary, and privileges might subsequently be restored when circumstances change. Sometimes it is only desired to recall a particular privilege while leaving other privileges for the object undisturbed, and this is known as partial revocation.

To summarise, protection is just one aspect of computer system security and protection mechanisms must be able to support a variety of protection policies that reflect the relationships between users of a system and the information it retains.

1.3. A Framework for Discussion.

For this discussion, it is convenient to regard an operating system as an abstract machine that defines operations which can be carried out on a set of abstract objects such as files, I/O streams and virtual memory. The operating system is responsible for the mapping between basic hardware resources and the the set

of objects. Each object has an attribute known as its type which specifies the set of operations that can be meaningfully applied to it.

There are many types of object. Processes are the most important from the point of view of protection because a process is the unit of execution and represents the locus of execution of a virtual processor through a procedure or sequence of code. The state of a process is held in a process base or state vector. Processes can synchronise and exchange information with one another by use of an inter-process communication system. At any time a process will have a set of privileges describing the information to which the process is allowed access.

Only capability-based systems are considered in this thesis. A privilege is represented by a capability that specifies both an object and an access code describing the privileges the capability confers. A capability is not an object; it is a ticket of permission that cannot be forged or corrupted. Capabilities are stored in memory marked in some way to distinguish them from ordinary data and it is not possible for arbitrary programs to mark items in memory; otherwise, it would be possible for users to manufacture bogus capabilities. Processes can share access to a common object by having copies of a capability for the object, each containing identical information about the substance of the object, although the access codes might differ if the various processes have unequal rights of access. A process can share its privileges with others by distributing copies of capabilities and if one of these capabilities is subsequently revoked, all copies derived from it must also be revoked.

The set of privileges owned by a process at a given time form a protection domain. The privileges of the process can be altered either by transferring capabilities in or out of the current domain, or by switching to another domain. In capability systems, domains consist of a set of capabilities that form an environment, one component of which is the code associated with the domain. Execution is switched between domains by use of a domain call primitive which has a domain capability, specifying the environment of the called domain, as its argument. After a domain

call, execution continues with the privileges of the called domain made available and those of the calling domain made inaccessible. Control is always transferred to a predetermined starting address in the procedure of the new domain so that the domain's privileges cannot be misused by jumping into its code at some random point. The process may call other domains for which there are capabilities in the current domain's environment. When execution in the domain has finished, a domain return operation is performed which resumes control in the original calling domain immediately after the point at which it was left, with the privileges of the called domain disabled and those of the caller restored. The domain call mechanism is very similar to the notion of a subroutine in a high level language and the similarity extends to inter-domain communication by a parameter mechanism which permits capabilities to be transmitted between protection domains.

An important property of a domain architecture based on capabilities is that it is non-hierarchical and can be used to model situations of mutual suspicion because the capabilities of different protection domains can be disjoint. This means that the only privileges that can be acquired by a called domain from its caller are those passed as arguments in the domain call. Similarly, a calling domain has no influence on the privileges of the called domain and can only gain capabilities that are returned as results.

So far, domains have been characterised passively as repositories for capabilities and code in which a process can execute. However, it is often useful to consider a domain in a more active sense as the exercisor of the privileges bound into it whenever a process executes within the domain and the term 'domain' will be used for either interpretation provided that any ambiguity can be resolved by the context of its use.

The memory of a domain will be assumed to consist of a number of segments [Dennis 65] each of which consists of contiguous addressable items. To be able to protect small data structures effectively, a protection mechanism must be capable of protecting many small segments only a few words in length as well as larger ones. It is also necessary to be able to generate capabilities

for only some portion of a segment and this can be achieved by holding refinement information in a capability for a segment to select some sub-segment of the total segment referenced by the capability.

1.4. Extensibility

In the design and construction of large software systems there is need for a suitable design methodology to describe the relationships between all of the components of the system so that its complexity can be reduced to manageable proportions. Perhaps the most promising scheme is that of layering in which the system is constructed as a base level or kernel surrounded in an onion skin like manner by a series of extension layers. Each layer enriches its environment by adding to the features provided by the inner layers to produce an enhanced environment for higher layers. The CAL-TSS operating system was designed as a sequence of protected layers and the technique proved successful in aiding the construction of the system [Lampson 76].

The primary rule of the methodology is that knowledge about higher layers must not be built into lower layers. This is so that, in conjunction with the obvious precaution of protecting lower layers against interference from higher layers, there will be a structure in which modifications to, and malfunctions of, higher layers cannot affect the correct functioning of lower layers in the system.

From a top-down point of view the construction of a layered design can be seen as the successive decomposition of a complex system into simpler functions until eventually, in the kernel, they can be directly mapped onto hardware operations. On the other hand, a bottom-up view shows each level of extension as transforming some pre-existing system into a more complete environment by adding useful new features and facilities. This latter view is the most appropriate in the case of extensions written by users to tailor the system to suit their requirements, although to a considerable degree the exact distinction between systems programs and user programs becomes blurred in a layered system.

In the object oriented approach, extensions are viewed as defining new types of object and providing the operations that manipulate them. Each layer in the system can be thought of as providing a new abstract machine whose operations are constructed out of the operations of the abstract machines running in inferior layers. The objects provided by extension layers are described as extended objects and they are represented in terms of objects administered by lower layers. In general, representations are concealed from the users of extended objects who see these objects as atomic items. The manipulation of extended objects in terms of modifications to their representations is the duty of software running in the level providing the extension. An obvious constraint is that the layer providing the abstraction of a new extended object must not be able to subvert the layers providing the types of object it uses to implement its own types.

In an operating system kernel, the sort of abstract objects expected are files, directories, I/O streams and so on. The base level protection mechanisms of the kernel must be capable of extension to provide access control and naming functions for these objects in a manner uniform with that used for with hardware resources. The features of a kernel for coping with extended objects are known as type-extension features.

1.5. Scope of the Thesis.

The work described in this thesis follows up work on the CAP project which led to the building of a microprogrammable processor for investigating a powerful and efficient memory protection architecture based on capabilities [Needham 77, Needham and Walker 77]. The CAP project successfully demonstrated the usefulness of capability-based protection in the construction and debugging of an operating system. The protection mechanisms of the CAP machine were all microprogrammed, which greatly contributed to their simplicity and efficiency.

By concentrating purely upon memory protection, the CAP microprogram lacked some of the more advanced features of software kernels such as HYDRA for C.mmp [Wulf et al. 74] and CAL-TSS [Sturgis 73, Lampson and Sturgis 76] in the area of type-extension

and revocation of access. The aim of the research leading to this thesis was to investigate the possibility of providing a kernel in microprogram which was comparable with the software kernels, but retained the effectiveness of the CAP memory protection architecture, especially with respect to efficiency. It was also decided to investigate some other addressing and naming strategies in the new kernel both because they aided the introduction of new facilities and also to compare the various mechanisms used in protection systems. The work involved designing a kernel and implementing it in microprogram for the CAP machine as a substitute for the memory protection microcode.

The CAP has a modest amount of microprogram memory (4K of sixteen bit words) which has to hold code for emulating basic instructions and organising I/O in addition to protection mechanisms, so the range of facilities that can be considered is fairly limited, although not too severely. The CAP processor includes hardware for instruction decoding, virtual address translation and carrying out access checks, which helps both to reduce microprogram length and to increase efficiency.

This thesis falls into two parts: the first seven chapters deal with the design of protection kernels and the remaining six chapters describe the implementation of a kernel for the CAP machine, whose design evolved from a consideration of the issues discussed in the first part. This latter part is entirely original work, whereas the first part compares and analyses work by others which is acknowledged appropriately in the text.

Chapter Two is devoted to a description of the design aims and guiding principles involved in kernel design. Chapter Three deals with the ways in which capabilities can be associated with objects, and Chapter Four explores the relationships between capabilities and addressing mechanisms. Chapters Five and Six deal with type-extension and revocation and in Chapter Seven there is a discussion about protection domains and processes.

Part Two starts at Chapter Eight in which there is a detailed description of the CAP hardware which provides the basis for Chapters Nine to Eleven that describe the following aspects of a

kernel for CAP: basic mechanisms, type-extension and revocation scheme and a process and protection domain architecture. Chapter Twelve gives a detailed description of the internal organisation of the kernel microcode. The final thirteenth chapter reviews and evaluates the kernel and looks towards future research.

CHAPTER TWO.

KERNEL DESIGN PRINCIPLES.

2.1. Protection Mechanism Design.

From their experience with the construction of protection systems, many people have proposed guidelines for the design and construction of successful systems and these are enumerated by Saltzer and Schroeder [75] in their paper on protection. In the following paragraphs these principles will be examined to see how they influence the design of a protection kernel and, in the next section, particular attention will be paid to their implications for a microprogrammed kernel. Many of the design principles may may be described as common sense and they are applicable to all aspects of system design, not just to protection.

It is universally acknowledged that a simple small design is better than one which is large and complex because it is much easier to check the correctness of its implementation (by simple line-by-line inspection of its code, for example) and also because it is likely to be more efficient. If the number of functions carried out by a mechanism is small, it is a simple task to exhaustively test all of its operations to verify their accuracy. An economy of mechanism has benefits for the users of the system too, as the amount of information they have to learn will be small, and this increases the likelihood that they will understand the mechanisms and use them effectively. To the system designer this principle suggests that he ought to consider carefully the primitives he is going to supply and ruthlessly remove any facilities which are redundant or unlikely to be used. An analogy may be drawn from the language Algol68 [Van Wijngaarden et al. 76] whose basic constructs are designed to be 'orthogonal' in the sense that to achieve a particular effect it is obvious which construct is appropriate. It is clear that the benefits of a compact design must be traded against any loss in the number of functions provided by of the system or inefficiency caused by the lack of an important operation.

In a protection system it is important to arrange that the default relationship between programs and information is that of 'no access permitted' until an appropriate privilege has been duly checked. This is true of capability systems because every access to an object must be accompanied by a capability that is inspected before the access can proceed. The explicitness of privilege in this sort of arrangement makes it easy to find out which objects a program can access by looking at its stock of capabilities. Furthermore, if the default state is to refuse access, any error in a program has less chance of causing harm by spoiling information to which it has no right of access.

In Chapter One it was noted that the degree of privilege possessed by a program will change with time and it is therefore not sufficient to perform access checks statically or just once only on the first access to an object, because subsequently the execution of a program running in parallel may cause an access code to be modified by revocation of a capability. This implies that there must be complete mediation, which is to say that every access to an object must be verified independently of whether or not the access has succeeded in the past. This imposes a system-wide view of protection independent of considerations of the structure and constitution of objects and provides the motivation for designing protection mechanisms as a low level component of a system. However, the implementor of a kernel does not have the full benefit of protection machinery and has to rely on weak hardware support for protection. For this reason, a kernel should contain as little as possible in the way of facilities beyond basic protection primitives so that as much of the system as possible is built upon the protection mechanism as its foundation to obtain a greater level of ruggedness. By contrast, in software kernels like HYDRA and CAL-TSS it is common to find that the kernel is a substantial operating system in its own right and makes little use of its own protection services. The designer of a kernel should aim to implement the basics of his protection machinery as a simple, small core which is then used by every other component of the kernel to access objects so that the kernel does not have private (and therefore suspect) protection,

naming and addressing schemes. Access control must persist at all stages in the life of a system from initialisation through normal running to shutdown. For example, it should not be possible for a user to tamper with the information that is to be loaded at the next system start up while the system is running, otherwise he could modify the system to pass him privileges illegally the next time it is reloaded.

A protection system that relies on the concealment of a secret or password to ensure the non-subvertibility of its mechanisms should be treated with suspicion because the security of the system will not be assessable in absolute terms. Passwords belong to the external realms of authorisation of access and it is only possible for a protection mechanism to prevent unacceptable access to secrets; it is not safe to assume that secrets can neither be guessed nor deduced from the observation of external events. Reliance should not be placed on the concealment of system code as a protection mechanism either, because if the protection system is free of secret algorithms, it can be reviewed independently by sceptical users without fear of abuse. This builds up the trust held in the system by its users and if the protection mechanism is audited independently by several people, the probability of outstanding errors remaining within it is decreased.

The ruggedness of a protection mechanism is greatly increased if it requires the presence of two 'keys' to open a 'lock' because no single mishap can lead to a breach of security. The major example of the separation of privilege in protection schemes is that of type managers which are programs responsible for looking after all objects or resources of a particular type. A type manager usually has privileges relating to the class of objects as a whole, such as being able to create objects in the class or to alter their representation, whereas the privileges to use particular members of the class are distributed amongst the users of the system. The only time at which a particular object can be modified is when two keys in the form of a privilege for a particular object and a privilege for the class of objects to which it belongs are brought together when the user responsible for the object passes a privilege for it to the type manager.

Possibly the most important design principle is that of minimum privilege, or as it is known in security conscious evironments, "the need to know", which is to say that a program should only have access to the information strictly necessary to carry out its function. In a capability system, this means that a protection domain should contain just those capabilities essential to completing its task and that a complex series of operations should be divided out amongst a set of domains, each element of which performs a simple, well-defined job and has exactly the privileges required to carry it out. Putting firewalls into a system in this way limits the propagation of damage after an error because only the few objects accessible to the erring domain can be harmed. It is also easier to locate errors because any failure can be directly accounted to the domains that have access to the information damaged by the error. A common practice is to associate a protection domain with each distinct data structure or abstraction so that a domain is rather like a module of the sort proposed by Parnas [72]. This organisation has the advantage that all of the operations for an abstraction are located in one place and it is easy to get interlocks right and to ensure the consistency of internal tables. However, for each particular service provided by the module, its domain is over-privileged because it carries around privileges for all of the services of the module. To some extent, this over-privilege can be overcome by use of the separation of privilege described earlier or by the use of templates, as found in HYDRA [Jones 73], that modify the privileges of a domain according to the access codes of capability arguments presented to it. A major consequence of minimum privilege for the designer of a protection mechanism is that it must be capable of efficiently supporting the interaction of many small, independent domains and this demands that the cost of a domain call operation be small.

Small protection domains tend to contain very simple data structures and in consequence it must be possible to protect very simple objects and segments just a few words in length. The notion of the 'grain' of a system is used to indicate to what degree it is reasonable to distinguish between items of data and

protect them individually. In general terms compared to most software kernels, the CAP memory protection system has a very fine grain of protection and this greatly contributes towards the ruggedness of the operating system built upon it [Needham 77].

Jones [73] proposes a yardstick known as a suitability factor, which indicates how closely a protection mechanism will allow the principle of minimum privilege to be attained. For every domain in the system, Jones defines an accuracy measure which is the ratio of the number of privileges exercised by a domain to the total number of privileges it owns. Clearly in the state of minimum privilege, this ratio will be one and will fall away to zero as the degree of over-privilege rises. The suitability factor of the entire system is defined as the average accuracy measure across the system and Jones shows that for a capability system with a non-hierarchical domain structure, it is possible to approach very close to the ideal value. This has also been demonstrated in an analysis of the CAP system [Cook 78].

Most conventional computer systems have very weak protection mechanisms and in consequence, users have little experience of taking full advantage of a well-protected system. So, for this reason, it is essential that protection mechanisms should be straightforward so that it is easier for users to remember protection techniques and how to employ them. To a great extent, this principle can be met by keeping to a simple, compact design in which all of the basic primitives are distinct and easy to understand. The psychological acceptability of a protection system depends upon users being readily able to employ the functions of protection machinery to suit their particular requirements.

A cornerstone of the HYDRA project has been the separation of policy from mechanism [Levin et al 75]. For example, a process scheduler in an operating system may be driven by interrupts alone or may allocate fixed duration time-slices, and there are many different ways of organising priority queues in response to different modes of operation. Ideally, these policy matters should be parameterised so that common kernel primitives can serve all possibilities. This is an important design principle because,

in the lifetime of a system, it is likely to be put to a variety of uses that may not have been apparent at the time that the kernel was designed and it is clearly undesirable to have to modify the kernel for every new application of the system. Separating policy from mechanism also holds advantages if it is required that the system should be able to provide two different services simultaneously, such as a transaction-based information retrieval system and a general purpose time-sharing system, because the same mechanism can be used for both services which may have different sets of policy parameters.

2.2. Microprogrammed Kernel Design.

It has been a long standing principle throughout the life of the CAP project that the set of facilities provided by the microprogram should be a self-contained whole that does not rely on the integrity of the software built upon it, rather than a microprogrammed extension of systems software. This is so that the verification of the basic protection machinery can be accomplished simply by inspecting the microcode free of any considerations relating to other software. It is likely that there will be awkward and complex interfaces between microprogram and software if the kernel is split between them because of the difficulty of keeping state information in step between the two levels. On those occasions when the microprogram is unable to cope, for example on a virtual memory fault, the only acceptable means of communication to the software is by raising an interrupt and the microprogram must not make any assumptions about whether or not the software will handle the condition correctly. Whenever the software wishes to negotiate with the microprogram, it must present its arguments in a form which the microprogram can check against appropriate capabilities before going ahead. Thus for example it would be inadmissible for the microprogram to accept an absolute address or to manipulate an object in the absence of a suitable capability.

The need for simplicity in a microprogrammed kernel is much greater than in a software kernel because of the lack of space for long sequences of code and the untidy nature of microinstructions and their side-effects. A lot can be gained by sharing as much

common code as possible in the kernel between its primitives, for example, to evaluate capabilities, modify representations of objects and carry out access checks. The advantage of this is that it is only necessary to verify the correctness of a particular function once and also there is a saving of program code. Naturally, this must be balanced against the overheads of microprogram subroutine calls and any inefficiencies introduced by calling routines that handle general cases rather than using possibly shorter code to handle particular simple cases. This can often be circumvented by careful design of kernel subroutine entry points and parameters.

To avoid confusion in software or accidentally permitting breaches of security, the microprogram must check all of the arguments of a kernel operation before it goes on to modify any data structures so that a protection violation cannot occur during the execution of the primitive and leave things in an inconsistent state. This means that kernel functions must be restartable and on a restart all arguments must be checked from scratch, because an excursion into software caused by an interrupt is liable to result in the modification of the state of the machine.

Complete mediation is possible by ensuring that the microprogram always uses virtual addresses to access data structures through the addressing and protection mechanism and not by remembering evaluated absolute addresses, so that an error in the microprogram or a bad argument causing the kernel to make an illegal access will be duly trapped and reported as an access violation. This increases the ruggedness of the kernel and greatly aids debugging at the cost of an increased overhead in accessing information owing to the protection checks. If the microprogram has some device for optimising efficiency by retaining evaluated capabilities and representations of objects, it must detect when such an optimisation is no longer valid, perhaps because a capability has been overwritten in store. It is in the area of these optimisations that there are most likely to be mistakes that will allow unauthorised access to privileges.

CHAPTER THREE.

CAPABILITIES AND NAMING.

3.1. Names and Objects.

In Section 1.3 the essential contents of a capability were said to consist of an access code describing the privileges conferred by the capability and information to identify uniquely the object which the capability protects. In an early capability scheme due to Fabry [68], the information simply consisted of the representation of the object; for example, a segment capability contained the absolute base of the segment and its size. This is rather unsatisfactory because information about the structure of objects is not centralised and leads to difficulties if the representation of an object needs to be modified. A typical example occurs in virtual memory management: a segment can be relocated in store by altering its absolute base address and it is essential that all capabilities for the segment refer to its new position. To do so in Fabry's design involves searching through all of the capabilities in the system to locate those to modify. This tedious task is easily avoided by holding the representation data in some central tables and retaining a pointer into the tables within a capability. This pointer is known as the *name* of the object.

If the naming mechanism only accepts names that are embedded in capabilities, names can be kept free from forgery or corruption. The name of an object serves to identify it uniquely from all other objects known to the system. As names are found in capabilities, the issues of naming and protection are very closely related; in particular, naming mechanisms have a considerable influence upon the nature of the protection system that can be built around them.

Naming schemes may be divided into two categories: *nested* and *global* [Lauer 74]. In a nested naming scheme the name of an object is only meaningful within one node of a hierarchical tree of name spaces. In each name space there is a table giving

information about all the objects which exist within it and an object is defined in terms of selecting objects in an immediately superior name space in the tree, apart from at the top level where representational information is found. Thus, the entries in a particular table will contain names belonging to its higher name space. The bit pattern of a name has a different significance in every name space and it is necessary to translate names if they are passed between name spaces.

A global naming system is characterised by having a central table of object representations and names which are pointers into the table and have the same significance everywhere. It is usually the case that each object has a name different for all time from every other name, known as its unique identifier. Uniqueness implies that there is a single entry in the table for every object and that modifying this entry will affect all capabilities for the object throughout the system.

3.2. Nested Naming Schemes.

The major example of a protection architecture based on a nested naming scheme is the CAP system [Needham and Walker 77]. The system supports a hierarchical tree of processes and each node of the process tree acts as a coordinator to the processes immediately descending from it. These processes in turn are responsible for coordinating their sub-processes and so on. A typical process hierarchy is illustrated in Figure 3.2-1. There is a name space associated with each process and capabilities within a process contain short (sixteen bit) names that point into a table of objects available to the process, known as the Process Resource List (PRL) which contains information about the representation of the process's objects. For segments, the PRL at the top level, known as the Master Resource List (MRL), holds absolute base addresses and sizes. A segment entry in a PRL lower down the process hierarchy contains the address of a capability for the segment in the immediately superior address space. These data structures are illustrated in Figure 3.2-2. There is a mechanism by which segment capabilities can contain refinement data so that a junior process can have access to sub-segments of segments at higher levels with the same or reduced access.

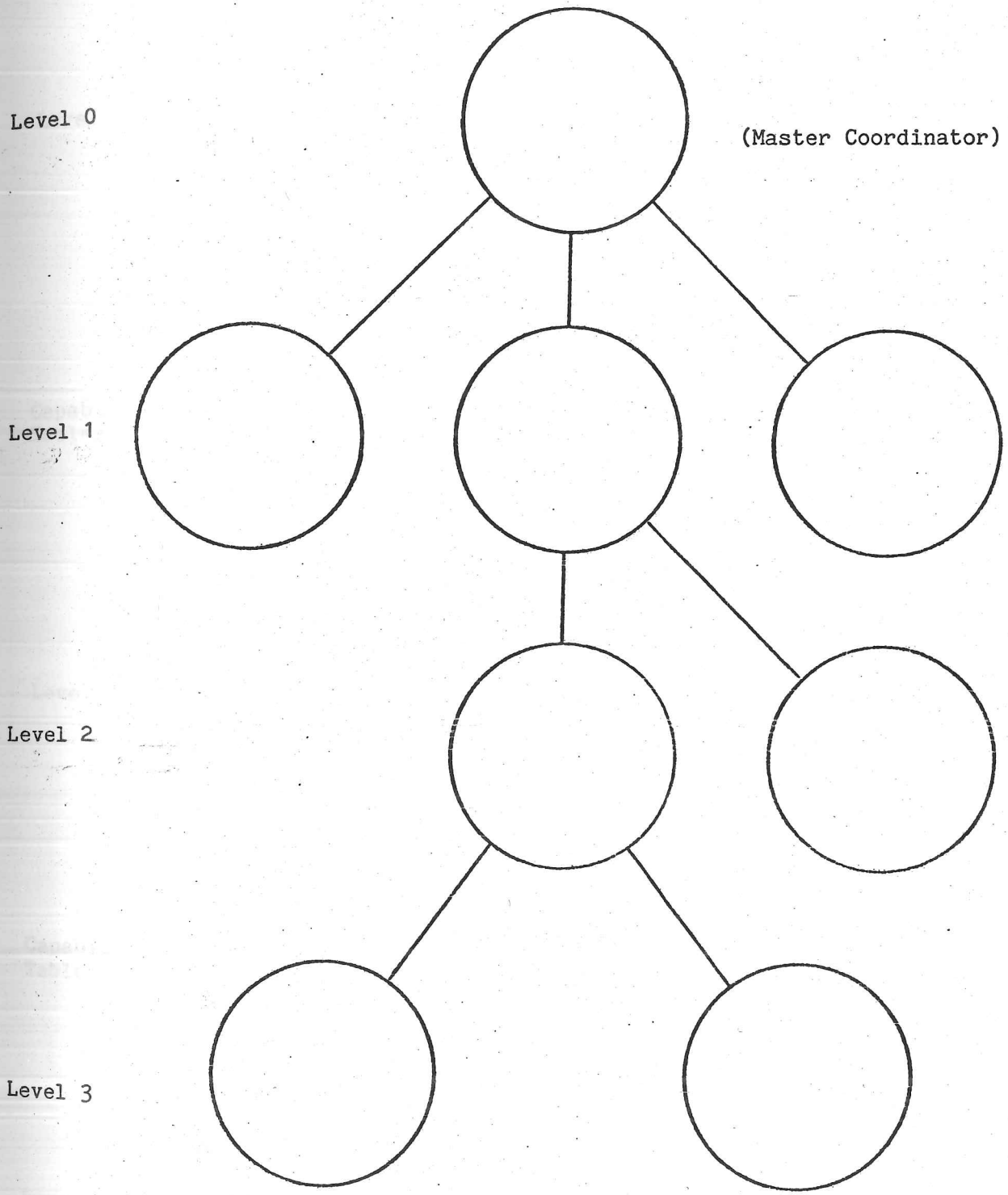


Figure 3.2-1 CAP Hierarchical Process Structure

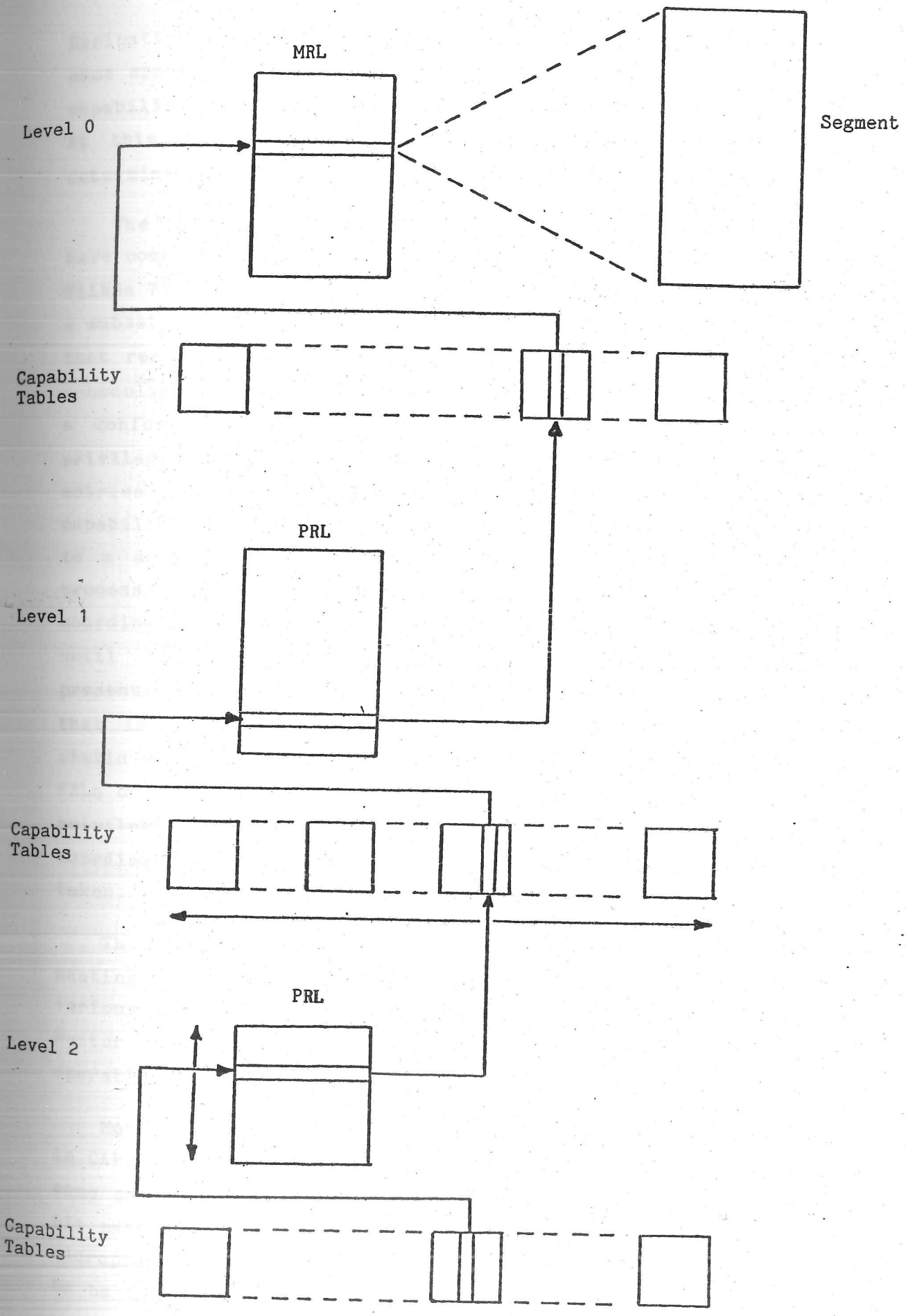


Figure 3.2-2 CAP Capability Evaluation

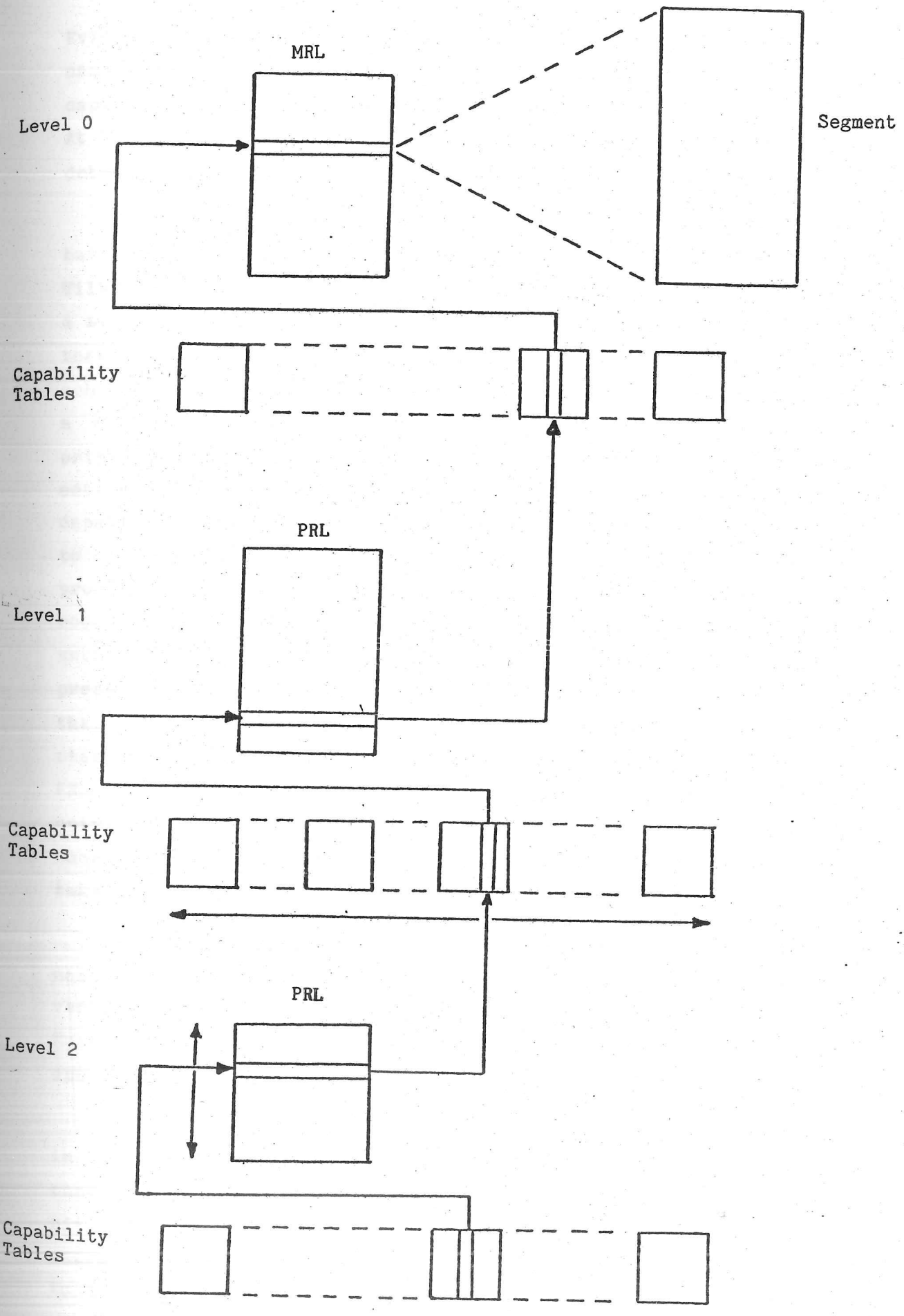


Figure 3.2-2 CAP Capability Evaluation

Evaluating a capability involves ascending through a hierarchy of name spaces, following indirections from capabilities to PRLs, to capabilities in higher name spaces until an MSL entry is reached. At this point, the position of the segment in memory can be determined and data within it may be accessed.

The rationale of the CAP scheme is that a coordinator should have complete control over the processes it schedules [Needham and Wilkes 74] and for this reason the privileges of a sub-process are a subset of those of its coordinator. In fact the only privileges that really belong to a coordinator are those that relate to the scheduling of processes. As it stands the CAP system suffers from a confusion between the control of time and the control of privilege which may be directly attributed to the fact that PRL entries define the representation of objects by addressing capabilities at the next level, rather than by pointing directly to a superior PRL. The reason for the capabilities of a CAP process leading to capabilities in the address space of its coordinator process is because CAP processes have no existence until they are actually running. A process is started by presenting to the microprogram a data segment of the coordinator that is to become the PRL of the new process and there is no static way of deciding which segments in the machine are potential PRLs or not. This means that the only possible place at which the privileges of a process can be gathered together is within the coordinator process from which the apparatus of the new process is taken.

The CAP architecture will support an indefinite depth of nesting, subject to hardware constraints, although in practice, various considerations lead to the adoption of just a single Master Coordinator and one level of sub-processes in the CAP operating system.

Moving capabilities around within a process is straightforward in CAP: all of the protection domains in a process belong to the same process-wide name space so it is sufficient just to copy the bit pattern of a capability whenever it is moved and CAP provides microprogrammed instructions for this purpose. If a capability is to be transmitted between processes having a common coordinator,

the transfer involves establishing a PRL entry in the receiving process identical to the PRL entry in the sending process and then copying the source capability to its destination slot with the name field translated to point at the new PRL entry in the receiver. Moving capabilities between processes that do not share a common coordinator is more involved: firstly a node in the process tree that embraces the name spaces of the communicating processes must be located; then the capability to send must be evaluated as far as this common node and finally, the capability must be allocated space in all of the intervening name spaces in the tree down to the receiver. There is no microprogram support provided for these operations so they must be performed by software. The complication of inter-process communication between different levels of the hierarchy has lead the CAP operating system to only permit messages containing capabilities to be despatched between sub-processes of the Master Coordinator. As message passing is implemented by software and involves the translation of names between name spaces it is considerably slower than the microprogrammed orders that may be used to communicate between domains within a process [Cook 78].

The CAP inter-process message system will only perform the transfer of segment capabilities with the result that it is not possible to send capabilities for objects containing names, such as protection domains. A CAP protection domain is known as a protected procedure and it is defined by an ENTER capability (named after the ENTER instruction which is used to change protection domains) that points to a PRL entry that in turn points at up to three other PRL entries for capability tables holding the capabilities that form the protected procedure. Figure 3.2-3. shows the structure of a ENTER capability. To transfer an ENTER capability between processes it would be necessary to make new copies of these tables so that all of the capabilities within them can be edited to index the correct offsets in the destination PRL when the procedure is transferred and space has to be allocated in this PRL for all of the segments accessible from the procedure. If the protected procedure included capabilities for other protected procedures, these too would have to be unravelled. Even if it were possible to pass protected procedures between

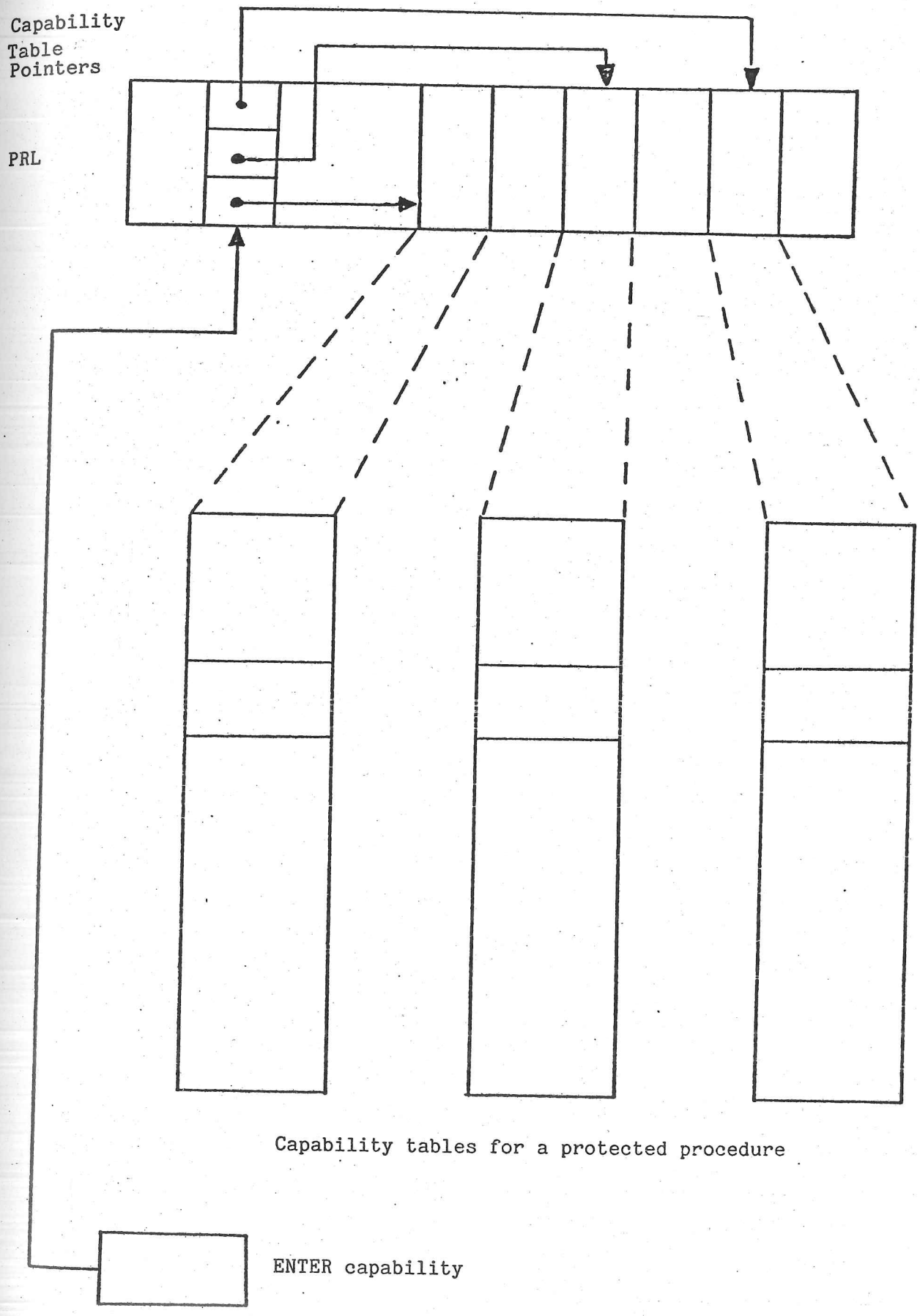


Figure 3.2-3 A CAP ENTER Capability

processes, there are other problems concerned with parallel execution in a single protection domain which will be discussed further in Chapter Seven.

In conclusion, the advantages of a nested naming scheme mainly arise from the efficiency of name look-up by simple indirection and compactness of short names, together with the simplicity of the object tables found in each name space. In return, there are problems concerned with passing names between name spaces and managing objects whose representations are distributed around the system in the several name space tables.

3.3. Global Naming Schemes With Forever-Unique Names.

All of the naming problems mentioned above may be avoided by the use of global names which are independent of domain and process architectures, so there is no difficulty in passing a capability around by copying its bit pattern. The HYDRA system [Wulf et al. 74] uses global naming and ensures that names are unique in space and time by deriving them from a fast clock that will never stop during the entire lifetime of the system. Every object ever known by HYDRA is given a unique name which remains associated with the object, even after it has been deleted. To cope with the number of objects that will exist during the system's life, unique identifiers are long (sixty-four bits), as opposed to the smaller sixteen bit name field of CAP capabilities. As well as being vast, the HYDRA name space is also sparse because of gaps owing to objects that have been deleted, and the intervals in which the clock runs but no new names are generated. For this reason, associating names with entries in the central table must be done by hashing. The entire hash table is too massive to retain in primary memory and it is paged from a fixed head disc. A small hash table in memory, the Active Global Symbol Table holds map entries describing objects that have been used recently and a low priority process slowly scans the table and arranges that it only contains information corresponding to objects that are in current use. If an entry for an object is not found in the active table, an entry must be found for it from the Passive Global Symbol Table on disc. The process of hashing in the active table is quite slow even compared to the time taken to traverse the

naming hierarchy in the CAP system and if an object is not in the active table, the time taken to find it is considerably longer because of real-time delays while the disc is accessed. Thus, although the mechanisms of HYDRA are conceptually simpler than those of CAP they are much more expensive in terms of time and computation.

A major advantage of a unique identifier scheme is that it is possible to preserve capabilities directly in a filing system. In a nested scheme this is not possible as a capability may be subsequently retrieved in a name space different from that in which the name it contains is valid. In the CAP system, filed objects are given a unique System Internal Name and when a capability is filed, its name field is translated from a local name to a System Internal Name [Needham and Birrell 77]. For this reason capability segments are not filed, as to do so would require the translation of all of the names in the capabilities within the segment. CAP capabilities are preserved in filing system directories and it is the responsibility of the directory manager program to perform translations between local and System Internal Names. With HYDRA this is not necessary, as the unique identifier in a capability is always valid and has the same meaning throughout the system at all times. However, the integrity of the unique naming scheme depends upon the object table being kept scrupulously up-to-date and consistent; it must be retained without corruption over a system break and the table management software must guarantee that the table is never left in an ill-defined state. The same remarks apply to the internal name table in CAP, but that table only has to be updated whenever a capability is preserved in the filing system and the overhead of keeping this table up-to-date on disc is less severe than in HYDRA, where the table is modified more frequently in response to operations on all objects and not just those in the filing system.

The advantages that a global naming system has over a nested naming scheme for transferring capabilities between domains and processes is a strong influence on the level of type-extension features found in a protection system. In most type-extension schemes, abstract objects are represented by a data structure that

contains information about the lower level components of the object which is frequently in the form of capabilities. In a nested naming system the problems faced in moving these structures around are similar to those provoked by trying to pass ENTER capabilities in CAP. If a type-extension mechanism is to be useful, it must be possible to protect a large number of objects of varying levels of complexity which can be transmitted easily between protection domains and processes. For these reasons, those protection architectures that wish to support abstract objects are normally based on global naming schemes, so that capabilities and names may be passed around in a free and flexible manner. Furthermore, if a filing system is to allow protected objects to be preserved, the expense of translating run-time names to filing system names within the representation of objects may prove too great and unique identifiers are most commonly used as global names to avoid this overhead.

3.4. Other Global Naming Mechanisms.

It is possible to have global naming schemes that do not rely on forever-unique identifiers. A system considered by Watson [78] uses global names that are only unique for a run of the system, that is, whenever the operating system or machine is stopped and subsequently restarted, identifiers are issued afresh from their origin. This approach relies on the observation that most computer systems are halted at frequent intervals for routine maintenance, lack of work or because of hardware malfunction. Usually these events are separated by days or weeks rather than years, so the identifiers in Watson's scheme need not be as long as those in a forever-unique scheme, with the advantage that capabilities are smaller and less work is required to hash names. Hashing is still the only method applicable for organising the global object table because, even in a few days, its size would become immense if measures were not taken to keep it compact.

In this scheme it is not possible to place capabilities in a filing system without translating names into some internal unique form because preserved names will become invalid whenever the system restarts. On the other hand, it is no longer necessary to go to great lengths to keep the table intact over a system break

since the names it contains are volatile. As the table will no longer require backing up on disc, it can be looked after by microprogram because most table operations like identifier look-up will not need the advanced facilities (such as paging from disc) used by the HYDRA global symbol table machinery. Some software might still be required to locate and remove garbage from the table and perform other high level operations, but otherwise it would not be unreasonable to expect the microprogram to provide primitives for evaluating capabilities, distributing them, simple type-extension (including object creation and deletion) and so on. Unique identifier look up by hashing, however, can lead to a waste of machine cycles when collisions occur in the hash table and a search must be continued. In particular, looking for an entry for an object that has been deleted may occupy the kernel for a long time, causing a degradation of efficiency.

The Plessey System 250 [England 74] circumvents the cost of hashing short term names by arranging that names in capabilities directly index a global name table which is called the System Capability Table (SCT) as shown in Figure 3.4-1. The Plessey system only provides memory protection and does not support any form of extended objects, so all objects in the SCT are segments. The operating system for the Plessey machine detects and recovers any slots in the SCT that are occupied by segments that are no longer accessible from active capabilities so that they may be given to segments that are created subsequently. Names in the Plessey system are still global, but they are only unique in the sense that at any time a name is only associated with a single segment, although at another time the name might refer to a different segment if the original has been destroyed. Finding free slots in the SCT requires the use of a garbage collector that periodically scans all of the capabilities that are active in the system to determine which SCT slots are not referenced. The frequency at which the garbage collector runs and the size of the SCT must both be carefully tuned to avoid wasting space in the table and also to prevent the system locking up if the SCT is full. Names in the Plessey system are just sixteen bits long, which greatly contributes towards having short capabilities and the principal advantage of this scheme over Watson's proposal is

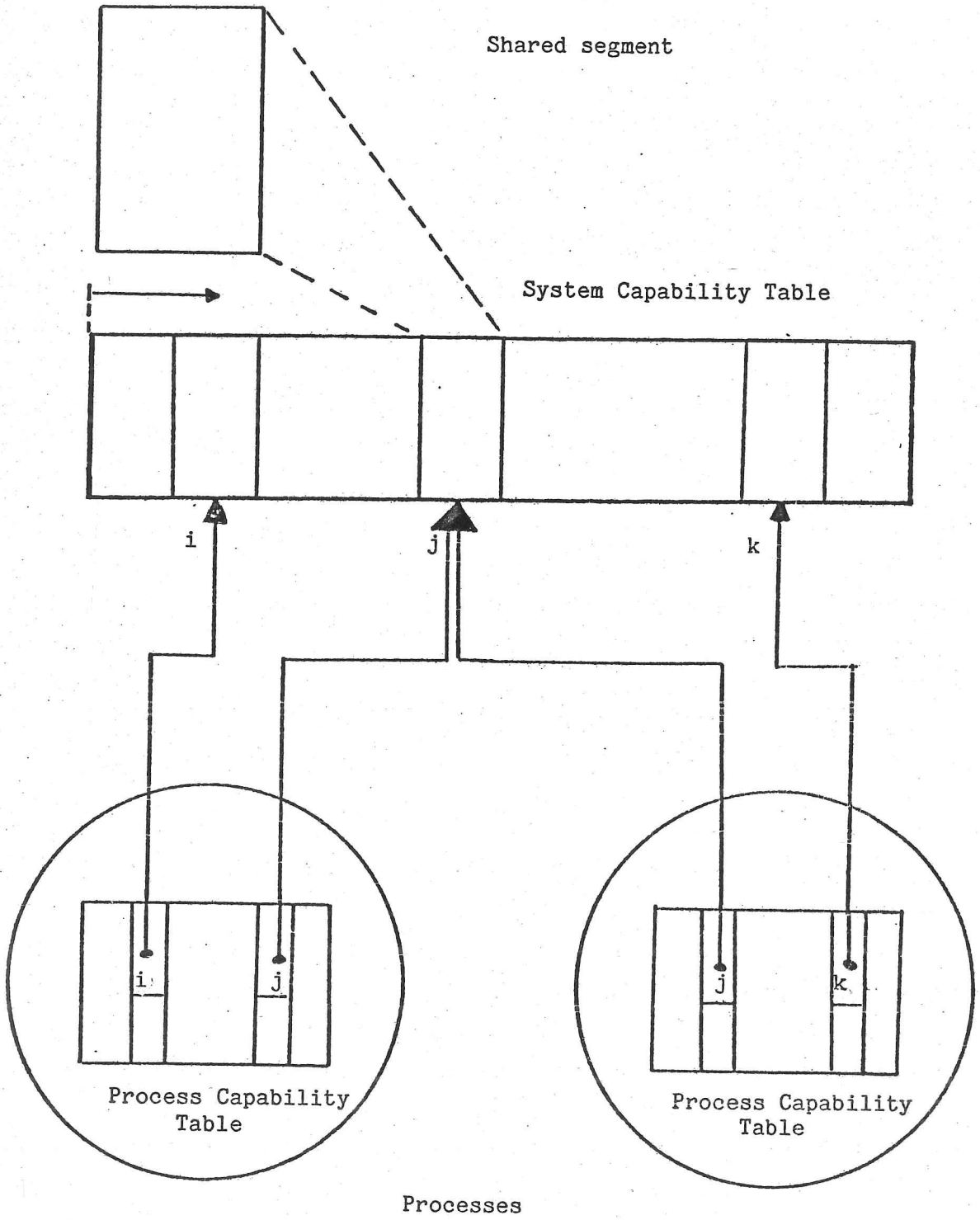


Figure 3.4-1 Plessey System/250 Naming Scheme

that proceeding from names to table entries only involves following a simple indirection and avoids the expense and complication of hashing.

The essential point to notice about the architectures proposed by Watson and Plessey is that, unlike the active global symbol table of HYDRA, the tables in memory are not caches for a larger data structure owned by the kernel. It might be the case that a higher level unique name table (such as the CAP System Internal Name table) exists, but the primitive naming and protection mechanisms know nothing about it and the management of forever-unique names is not a kernel function. The main advantage of adopting this view is that the global name table need only be of moderate size and resident in memory so that kernel naming and protection mechanisms can be implemented in simple and efficient code. The price paid for this facility is the need for translation between filing system names and run time names. In HYDRA the active global symbol table is purely a cache for the passive symbol table kept on disc and it is the duty of the HYDRA kernel to maintain both data structures, which is one reason why the HYDRA kernel is slow and unwieldy and has to be implemented in software rather than microprogram.

A compromise suggested by Lampson and Sturgis [76] to gain the benefits of short names whilst retaining a forever-unique name system is to make capabilities hold both a short run-time name and a long forever-unique name. Operations on the global name table in memory are carried out using short names to address slots within it and a quick check is made to ensure that a unique name held in the slot matches the unique name held in the capability being exercised. If the unique names do match, the operation is allowed to proceed, otherwise a trap is generated and the operating system can use the unique name to find or construct an entry in the table for the object, and the short name field of the faulty capability is then made to be the offset of the new slot. In essence, a short name is a 'hint' to the position of an entry for an object in the map. With this sort of organisation the map can function as a cache for a unique name table that is kept on backing store without involving the kernel in disc operations.

The kernel uses short names as pointers and this avoids the expense of hashing and leaves the management of forever-unique names to higher level components of the system. While this technique may seem to offer an ideal compromise between short names and unique names, there are many pitfalls to avoid. Capabilities are very long because of the need to hold both short and long names, and space must be found in the central object table entries for long names as for well as details of representations. It is necessary to provide code to manage both the small resident map and the larger permanent structure which have different naming conventions, and the interactions between the tables and the algorithms for managing them must be carefully considered to avoid problems of inconsistency, over-complication and loss of efficiency.

In general terms, all of the different management strategies for global name tables represent a compromise between the usage of space and time, so it is unreasonable to expect any single mechanism to be ideal. Instead it is necessary to consider the desired behaviour of a system and to adopt the techniques most suited to it.

CHAPTER FOUR.

CAPABILITIES AND ADDRESSING.

4.1. Capability Structure and Organisation.

The natural place at which to start considering capabilities and addressing is with the nature and substance of capabilities themselves: a capability is evaluated from the contents of a data structure in memory which serves to define both the object protected by the capability, the privileges the capability confers, and in some systems (CAL-TSS for example), information about the type of the protected object. It is useful to be able to refer to the data structures themselves as capabilities although, in the strictest sense, it is the result of evaluating the data structures that yields capabilities. In this thesis, the term 'capability' is used with both meanings provided that it is possible to resolve any ambiguity from the context.

Because they contain names, capabilities must not be either forged or corrupted if protection is to be guaranteed. It is therefore necessary to have some method for distinguishing capabilities from ordinary data so that they can be recognised and only authorised capability operations carried out upon them. There are two common techniques for performing this discrimination: firstly, each item of information in memory may be tagged with a bit saying whether or not the item is a capability, and secondly, memory may be partitioned into disjoint capability and data regions.

Tagging has been successfully employed by the Burroughs B5000 computer system and its descendants [Burroughs 61] and has been extensively investigated by Feustel [73]. The protected items in these systems are 'descriptors' rather than capabilities, but the differences between them are of no immediate concern except for one point: descriptors typically tend to be smaller than capabilities as they contain less information. The impact of this becomes apparent in the light of current trends to reduce the size of addressable items in memory. In the past, machines with items

of thirty-two, forty-eight and even sixty bit items were common, but nowadays the eight bit character, or byte, is becoming universal and proposals have been made for bit-addressable memories. In such addressing organisations a large object such as a capability is implemented as a contiguous sequence of locations of memory usually addressed by the offset of the first element in the frame. If tagging is to be used, it would seem that at first sight two tag bits are required, with the significance 'first item of a capability' and 'subsequent item of a capability' respectively, so that it may be ensured that capabilities are correctly manipulated, but clearly the overhead of associating two extra bits with a small item of, say, eight bits is wasteful and expensive.

A simple way of avoiding the expense is to insist that capabilities can only be stored starting at addresses that are a multiple of the length of a capability and that capability addresses must locate one of the predetermined capability frames; this only requires a single tag bit but complicates software because of the need to align capabilities which sacrifices many of the advantages of being able to access small items. A full discussion of tagging hardware for a capability machine can be found in Redell [74] together with some proposals for a scheme which is economic in terms of the number of tag bits, yet permits items to be arbitrarily laid out in store.

The generality of being able to mix capabilities and data freely in a tagged memory regime poses some system problems: some part of the protection system must be responsible for creating new capabilities and destroying unwanted ones and to do so it must be possible to write arbitrary bit patterns in capabilities, although the use of this privilege must be protected to ensure the integrity of the rest of the protection machinery. This operation conflicts with the setting of a capability's tag bit and some escape mechanism must be provided to overide tags which, in most tagged machines, is available only in a special or privileged state that allows <u>any</u> capability to be modified. This latter privilege is more sweeping than that which is actually required and it is not possible to limit selectively the capabilities which

may be affected.

A further problem arises from the observation that an operating system is obliged to know the location of capabilities and other protection data structures; for example, it may be necessary to scan all the capabilities belonging to a protection domain to find lost objects, or to detect garbage in internal tables. If capabilities are freely distributed throughout a domain's memory (as tagging would allow), the scan would have to include every item in memory that could possibly contain a capability. In any system, and even more so in one that has a large backing store, this task would be exceptionally expensive in both processor time and virtual memory traffic.

The difficulties encountered in a tagged architecture may be avoided by partitioning capabilities and data. In a partitioned system the access code of a capability for a segment will belong to one of two categories: capability type access or data type access. To perform data operations such as addition or shifting on items in a segment, it is necessary to present a capability bearing the appropriate data type access code such as read, write or execute and for a capability operation, a capability presenting a capability type code such as read-capability or write-capability must be used. Thus the interpretation of the contents of a segment depends upon the capability used to gain access to it and it is usual to refer to a segment for which capability access is held as a capability segment; otherwise, if the access is of data type, it is referred to as a data segment. In this scheme, the part of the operating system concerned with altering the contents of capabilities would have a capability giving data access to segments that are elsewhere accessed with capability access and because the ability to modify a particular capability is itself controlled by a capability, it is possible to control the privilege.

The software for managing capabilities in a partitioned architecture does not have to scan the entire memory of the system to find all capabilities, instead it is only necessary to consider capability segments, that is, those for which there is a capability with a capability type access code in existence. It

may be expected that there will be far fewer of these segments than data segments.

The disadvantage of partitioning is that a certain amount of generality is lost: it is not possible to have data structures represented by segments that contain a mixture of capabilities and data and there are occasions when the lack of this feature is a nuisance. Consider, for example, a directory or catalogue for a filing system: the directory cannot be implemented as a segment containing both capabilities for the objects filed within it together with data representing file names and access control information, whereas in a tagged machine the directory could easily be made from a single segment. In a partitioned architecture it would have to be implemented as two segments, one for data and one for capabilities, which is inefficient as it requires two transfers to bring all of the directory into store. The HYDRA system [Cohen and Jefferson 74] employs partitioning but tries to recapture generality by providing 'universal' objects, holding both data and capabilities, that are formed from two segments, one of which holds the data part of the object and one for its capabilities. The implementation of the object as two segments is concealed from the user, but it is not possible to interleave capabilities and data arbitrarily inside the object as the two sorts of information are addressed in different ways. By careful allocation of disc space it can be arranged that the two segments of a universal object are adjacent on disc and can be brought into store in a single transfer. The CAP and Plessey 250 systems also partition capabilities and data but neither has any facility for mixed type segments.

An important consequence of adopting a partitioned architecture is that, unless ordinary orders recognise capability and data type access codes, they cannot be used to move capabilities around between capability segments which means that a special suite of capability orders must be provided.

4.2. Capabilities and Virtual Address Translation.

In a segmented addressing architecture, an address contains two fields, one of which selects a descriptor for one of the segments in a virtual address space and the other indexes a particular item within the segment. Capabilities can be usefully employed as descriptors because they are protected from forgery or corruption and can be passed between address spaces to permit sharing, without the need to resort to complex linkage tables such as those found in MULTICS [Organick 72]. Capability segments can take the place of descriptor tables and each capability within the tables will define a segment associated with some virtual addresses. Two address spaces sharing access to some object will have similar capabilities for it in their (descriptor) capability segments. A capability used as a descriptor provides a bridge between virtual address spaces and the naming machinery because an address nominates a capability which in turn, provided there are no access violations, yields the name of an object which is the key for obtaining its representation.

There are two ways of using capability descriptors: explicitly by loading them into capability registers or implicitly by making the virtual address translation mechanism evaluate descriptor capabilities automatically.

The Plessey System 250 [England 74] is a capability register machine in which all of the capabilities available to a protection domain (or package in System 250 nomenclature) are held in a single capability segment, the Central Capability Segment, which is local to a process. The capabilities in this table hold names that point at entries in the global system capability table and access codes. The System 250 processor makes available to users eight capability registers which hold evaluated (segment) capabilities in three fields: absolute base address, size and access code. The data structures of the System 250 are shown diagrammatically in Figure 4.2-1. Capabilities are loaded explicitly in the registers by instructions of the form 'load capability register r with capability i' which causes the i-th capability in the central capability segment to be evaluated and then to be made available in the r-th capability register,

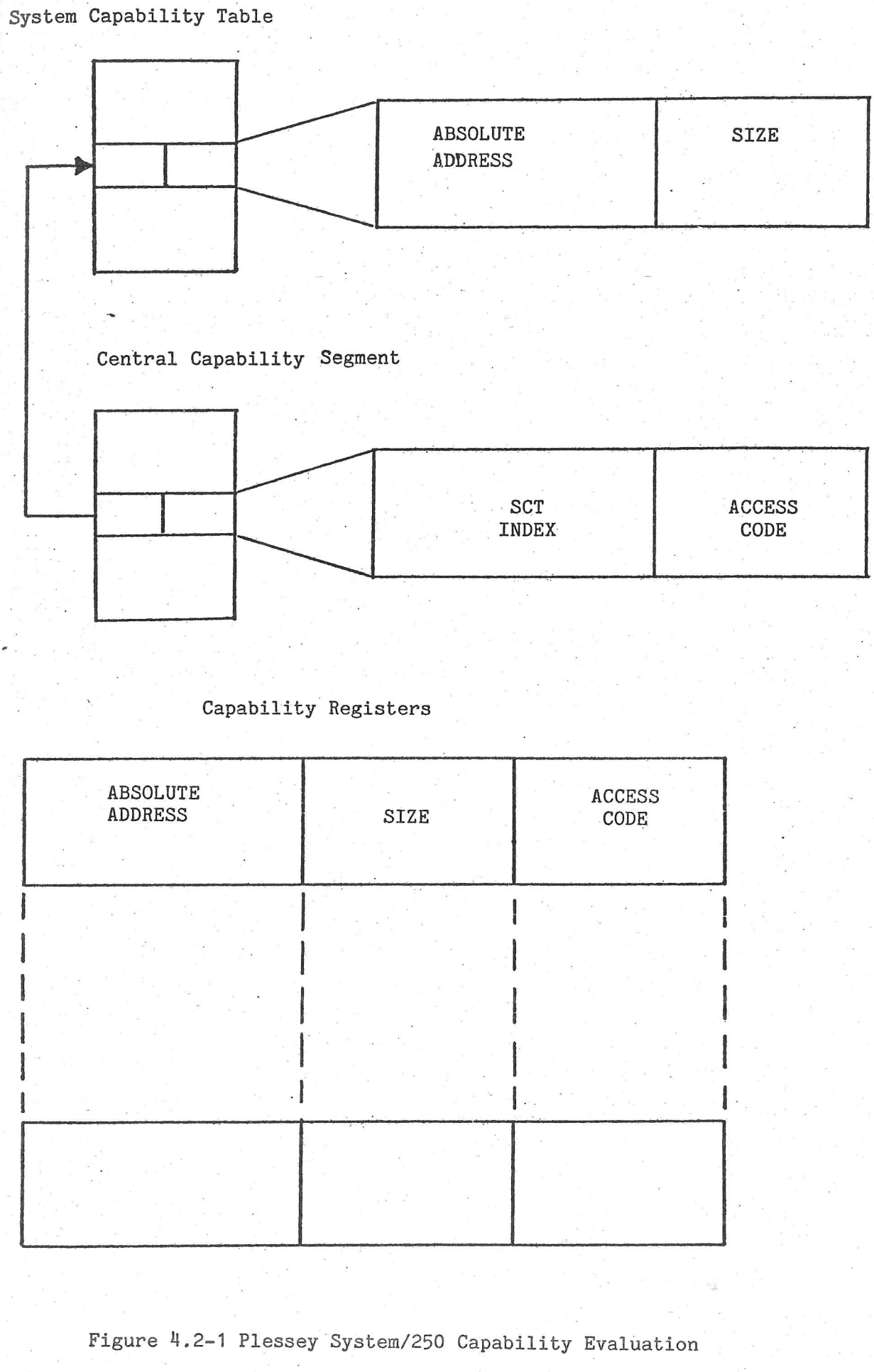


Figure 4.2-1 Plessey System/250 Capability Evaluation

overwriting the previous contents of the register. Addresses in this system take the form of a duplet <capability register number, offset in segment>. Whenever execution crosses a protection domain boundary, all of the capability registers must be flushed out as their contents will not be valid in the new domain and the contents of the registers are preserved on a stack so that the old environment may be restored when the called domain is left.

HYDRA likewise specifies objects by their offset in a central table. Associated with each protection domain is a capability segment known as the Local Name Space (LNS) and capabilities may be moved in and out of the LNS by kernel primitives which take LNS offsets as arguments. Objects in HYDRA can contain capabilities within their representation and the addressing mechanism permits these capabilities to be addressed by a path name which specifies a route starting at the LNS through the capability parts of a series of objects leading to the target capability. Each component of the path name consists of an offset into the capability segment of the last object reached. A typical path from a LNS through several objects is shown in Figure 4.2-2. The LNS is a normal HYDRA object; the capability part describes the privileges of a protection domain and the data part holds system and accounting data such as the number of capabilities present and so on. The ability to follow a path and pluck capabilities out of objects is controlled by an complex set of access codes [Cohen and Jefferson 75]. In HYDRA, a segment of memory is addressed by indexing a capability for the segment into the LNS and then causing the kernel to evaluate the capability and configure a nominated relocation register of the underlying hardware accordingly so that memory can be addressed through it.

Explicit capability register machines are unsatisfactory for a number of reasons. The most apparent is that programmers have to concern themselves with the allocation and priming of capability registers and this activity is not confined to system programmers; it must be carried out at all levels. Register allocation can be left under the control of a high level language compiler, although in doing so it is difficult to avoid introducing machine dependent features into the language. This can be a great disadvantage if

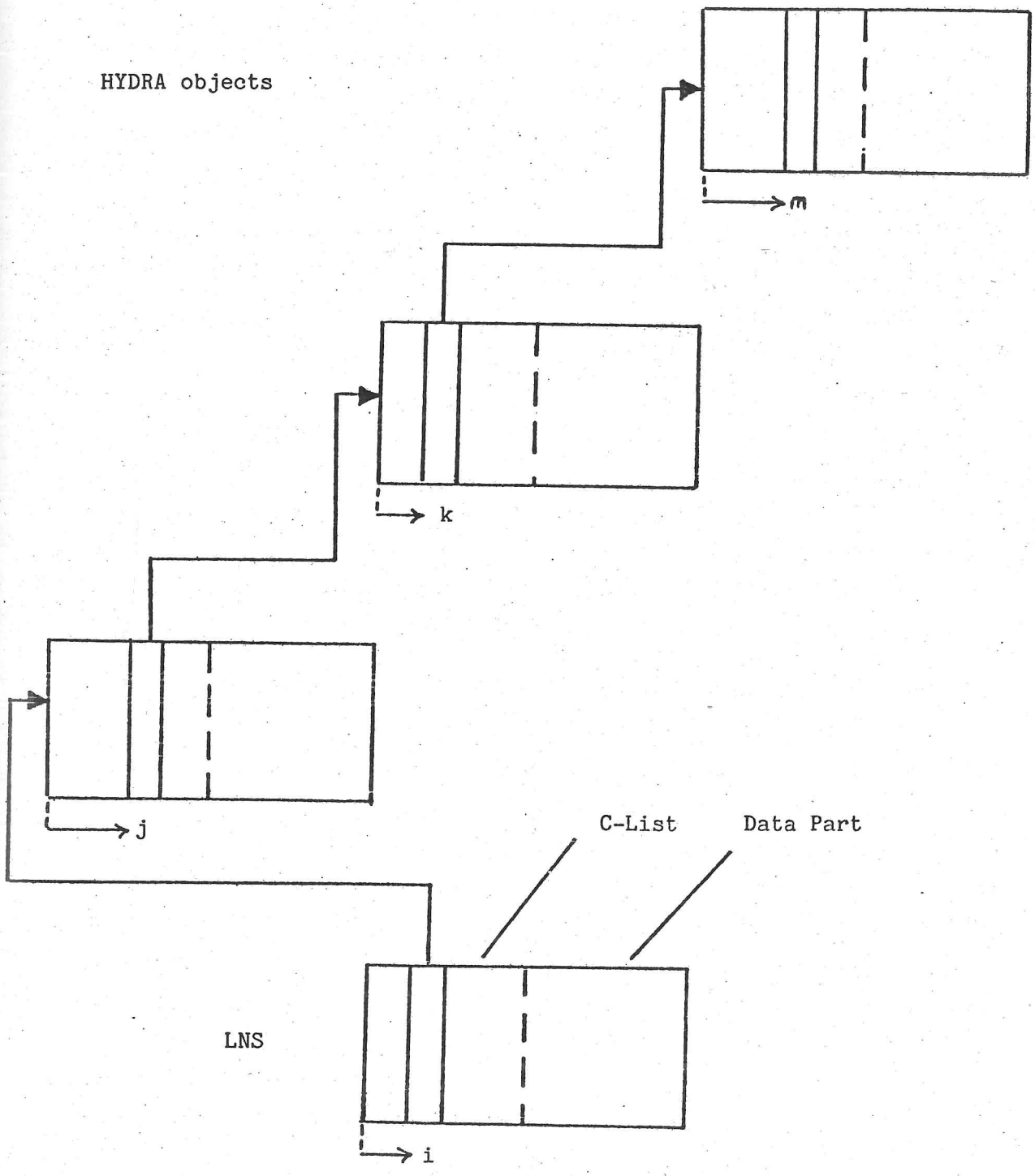


Figure 4.2-2 HYDRA Addressing Structure

the language is portable such as BCPL [Richards 69] or Algol68C [Bourne et al. 74], where all machines share the same compiler but have their own intermediate code translators. Introducing protection facilities in the translator is not easy as much of the information needed to decide the contents of capability registers is only available in the compiler.

If the working set of capabilities needed by an executing program exceeds the number of registers available, many machine cycles will be wasted in repeatedly loading and unloading capabilities. This loss can be diminished by increasing the number of registers available at the expense of more state information to preserve over a process or domain switch. Another danger is that a register may be left containing a capability when it is no longer required and subsequently, because of a programming error, the register may be exercised by accident or may be thought to refer to another object. This sort of thing can lead to very obscure program failures.

If a change is made to system tables, such as the SCT and Central Capability Tables of the Plessey system, any registers previously loaded from the tables must be flushed out and evaluated from scratch, as the data on which they rely might have changed. By the simple expedient of holding table offsets in capability registers it is possible to reduce the flush to only those capabilities dependent on the changed data. Additionally, if the change was in the global name table, it is necessary to flush out capabilities not only in the currently running process but also any preserved in process bases and other state information.

Many of these problems can be avoided by arranging for the addressing mechanism to select automatically and to evaluate a capability to determine an object's representation. In the Plessey system this would correspond to making addresses of the form <offset in SCT, offset in segment> rather than <capability register, offset in segment>. If every reference to an object, especially if it were a segment, caused a capability to be evaluated, the overheads of the mechanism would be immense but they could be avoided if evaluated capabilities are retained in

some form of capability cache. For example, in the case of the modified Plessey system, it would be possible to provide a bank of capability registers each preceded by a tag which is used to hold the central capability segment table slot of the capability from which the register is loaded as is shown in Figure 4.2-3. When an address is presented to the cache, an associative search can be made for a register whose tag matches the SCT offset of the address. If a match occurs, the selected register can be used to control access to store, otherwise a free register in the bank can be loaded and the store access re-tried. The function of the associative capability cache is similar in operation to the current page registers found in machines with paging hardware [Denning 70]. The management of the cache can be carried out by software running in a privileged state which permits changes to the contents of the registers or, as is the case in the CAP computer, by microprogram.

An associative cache is more expensive than directly addressable registers in terms of hardware, although the falling costs of integrated circuits is reducing the price of associative memory. In return for the investment, the advantages of slaving capabilities are of great benefit: no longer is it necessary for programmers to become involved in capability register loading and dumping and the protection mechanism becomes an integral part of addressing which offers simplicity for naive users of the system who are only required to understand the addressing architecture of the machine and not necessarily its protection mechanisms as well. This latter feature is also useful in the area of high level languages as the objects accessible to a program can be mapped into a language's view of the address space in which it runs, rather than forcibly having to bolt on knowledge of capability mechanisms. Whilst there is no longer the problem of leaving unwanted capabilities lying around in registers or misloading registers and accidentally permitting protection violations, the difficulty still exists at a higher level in that the proformae that an operating system uses to set up capability tables prior to running a program must agree with the addresses used by the program. This can be skirted around to a fair extent by allowing compilers to construct the proformae, as is the case with the CAP

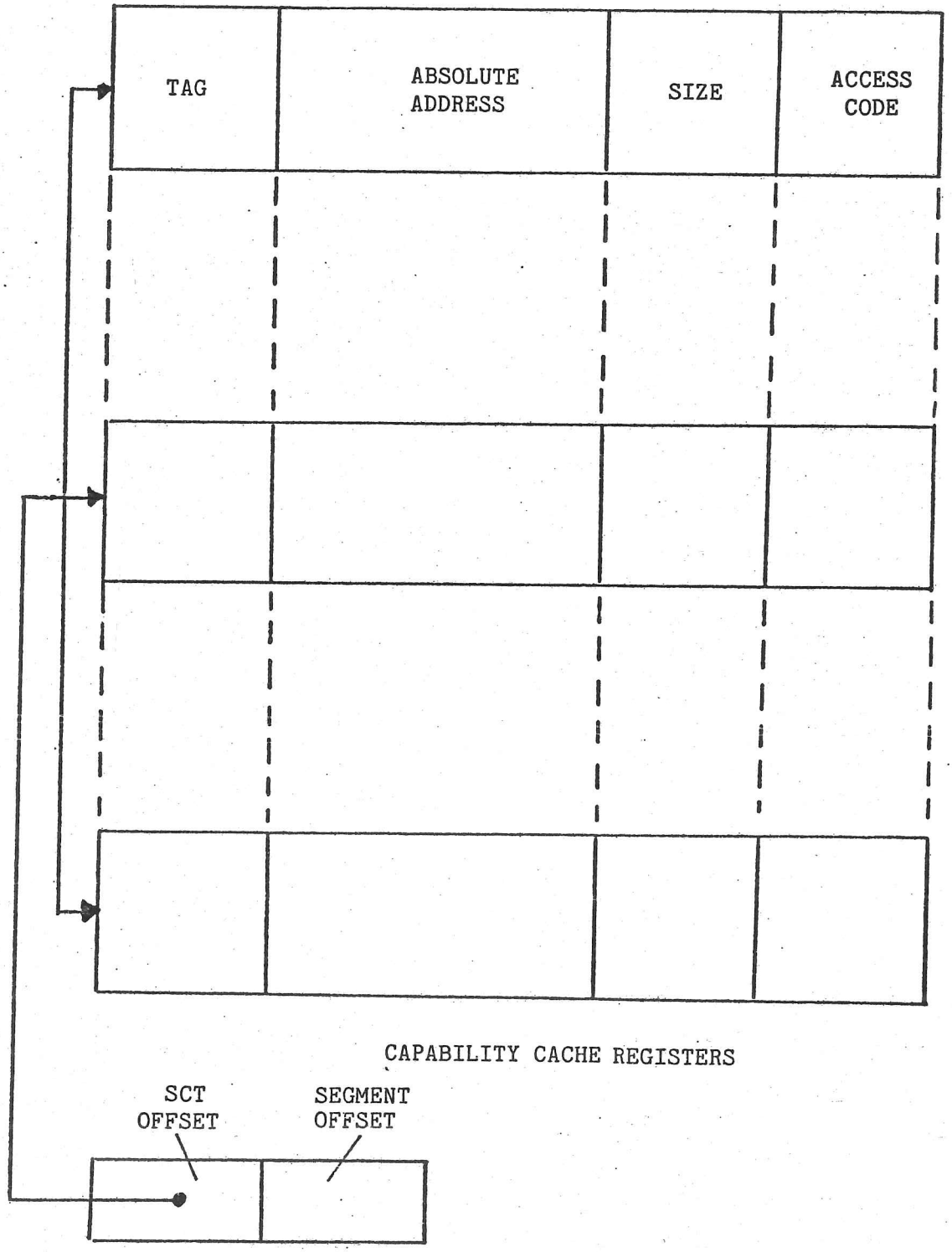


Figure 4.2-3 A Capability Cache

Algol68C compiler [Birrell 78]. It is, of course still possible for programs to become confused if they compute an address wrongly; however if the address space is vast, there is a good chance that misleading addresses will turn out to be invalid, more so than if an address is a small integer or register number.

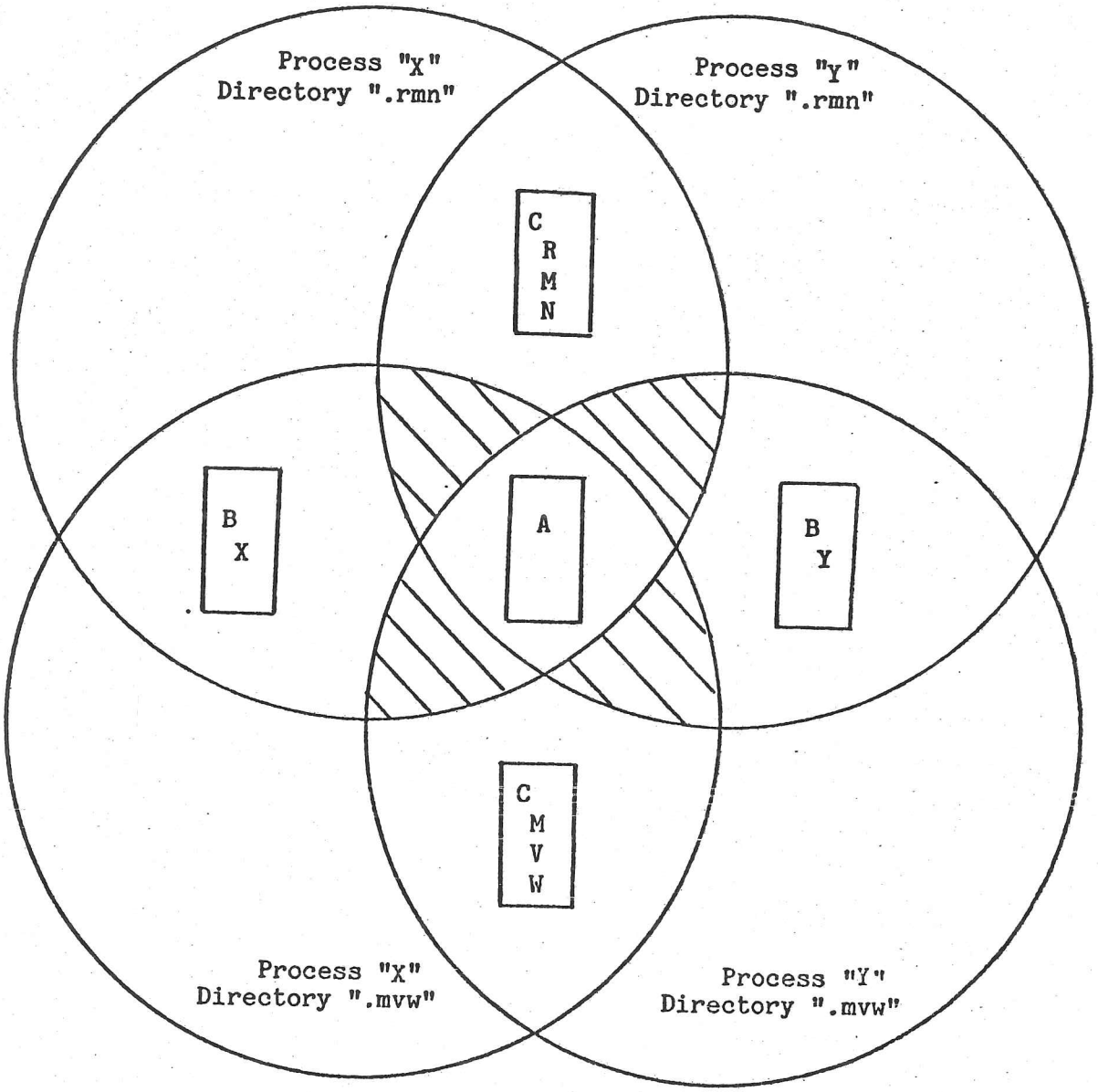
If the associative selection mechanisms of the capability cache are sufficiently powerful, it is possible to arrange that capabilities are left in the cache over a domain call so that on returning from the domain they may be made accessible again without having to construct them afresh. It is a necessary requirement that capabilities held over in this way are only accessible in the domain to which they belong. The cache of the CAP computer is driven like this and the retention of capabilities during protected procedure calls leads to a considerable saving of machine cycles [Cook 78]. Similarly with the need to flush out capabilities in a register machine, it must be possible to clear capabilities out of the cache if the data structures from which they are evaluated have been altered. As each register in the cache proposed above is keyed by the address of the capability it is derived from, any change to a capability can be accomplished by flushing out any entry whose tag matches the index of the modified capability. Changes in the global name table can cause the entire cache to be flushed or, by providing a field in each register giving the table offset of the object it protects, a selective clearing can be made.

4.3. Structured Addressing Architectures.

Having just a single capability table in a domain is not entirely satisfactory as it is not possible to share those parts of the domain that are common to other instances of it elsewhere in the system. An example of the usefulness of such a feature is provided by filing system directories in the CAP filing system: a directory consists of a segment describing the contents of the directory embedded within an instance of a directory manager protected procedure and every incarnation of the procedure can usefully share capabilities for segments of pure code, libraries and read-only data structures. The workspace of a directory manager is local to a process and can be shared between all

instances of the directory manager within a process because control will only be in one of the managers at any time. Some data structures, such as directory segments, are local to each instance of the directory manager and cannot be shared. If the capabilities of a protection domain can be divided up into these various classes and shared there is considerable scope for saving space as is illustrated in Figure 4.3-1. Furthermore, by splitting the table into a number of capability segments, it is possible to protect some of the capabilities belonging to a domain from being overwritten by placing them in a segment for which only read capability access is held. In HYDRA, where a process's capability table, the LNS, is a single table, there is a complex set of access controls provided to prevent individual capabilities within it being overwritten by accident. The ability to partition the layout of the capabilities belonging to a protection domain helps to prevent addressing the wrong capability by accident as, particularly if a domain is small, the capability address space will be sparse and arbitrary addresses produced by programs are likely to reference slots that are not in use and will cause a protection violation.

The CAP computer has an elegant addressing and capability architecture which serves to illustrate some of these points and is depicted diagrammatically in Figure 4.3-2. For the current purpose it is sufficient to assume that evaluated capabilities are efficiently cached in a large bank of capability registers. (The hardware for supporting the cache is described fully in Chapter Eight). As outlined in Section 3.2, the Process Resource List of every process contains an entry for every object available to the process and the address space of a protected procedure is defined by up to three capability tables (i.e. capability segments), the entries in which select a subset of the objects in the PRL. As control is passed between protected procedures, different sets of capability tables become enabled and so the selection of objects available changes. A distinguished entry in the PRL describes the process base which, as well as state information, also holds sixteen pointers to PRL slots that define the currently enabled capability tables of the process. A pointer may be null, in which case the corresponding capability table is disabled. The fourth,



A: Program Code and Global Privileges.
B: Local Process Workspace and Privileges.
C: Local Directory Data Structures.

Figure 4.3-1 Shared Capability Tables

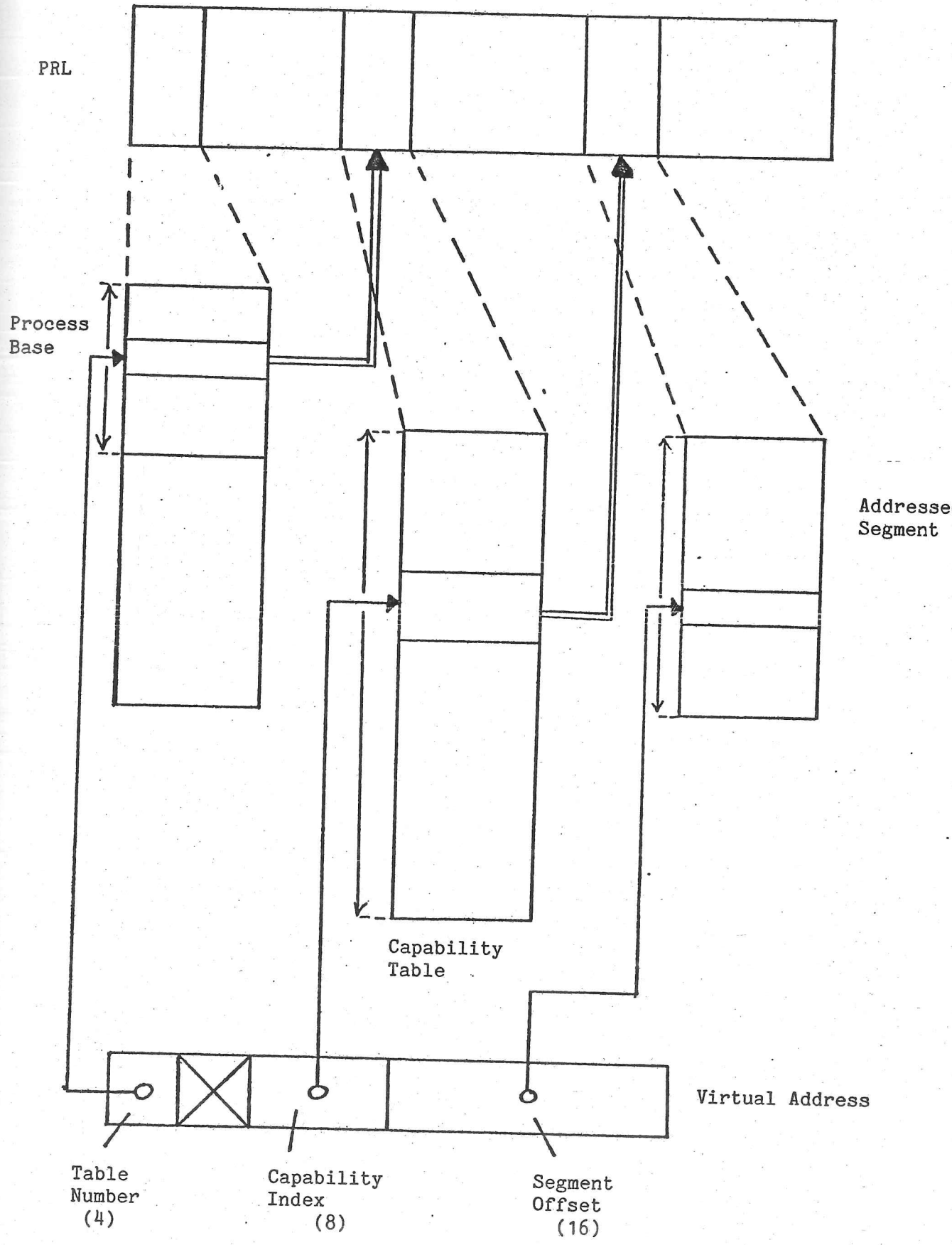


Figure 4.3-2 CAP Process Addressing Scheme

fifth and sixth capability table pointers are changed whenever a protected procedure is entered and the new values indicate the capability tables defining the address space of the called procedure. The second and third capability tables are used for argument passing between protection domains and the remainder provide a set of a tables that are globally available to all procedures executing in the process. Each of the capability tables may hold upto 256 capabilities, although in practice only a small number of capabilities are kept in a few tables and most of the address space remains unused. An address consists of three fields: a capability table number, a capability index and a segment offset. The first two fields taken together are known as a capability specifier. The capability table number nominates, via one of the pointers in the process base, the capability table containing the addressed capability and the capability index chooses the capability from within the table. The segment offset field is used to address words within a segment if the capability is store type, but ENTER capabilities and other non-store type objects are addressed by the capability specifier alone.

As was described in Section 3.2, PRLs contain the addresses of capabilities in the immediately superior address space and they are in the form of a capability table number and capability index pair.

Because of the changes of virtual address spaces arising during process and protected procedure switching in the CAP system, care has to be taken not to pass addresses between address spaces because they are not valid in any other context than the space in which they are defined. In particular it must not be possible for any part of the system to be duped into giving away privileges in an unauthorised fashion by a misleading address passed as an argument. This restriction on the propagation of addresses has not been found to be a great nuisance with the CAP system because any reference to an object during a protected procedure call is always accomplished by using a capability rather than an address and addresses are relegated to the simple task of identifying capabilities in the current procedure. The one difficulty that can arise is with multi-segment data structures

that contain inter-segment addresses because, if a capability for a particular component of the structure is moved to a different address, all of the addresses for it in the remaining segments must be edited appropriately.

4.4. Capability-Based Addressing.

In an important paper, Fabry [74] shows that the naming function of capabilities and their relationship to addressing descriptors can be used to provide a very elegant form of context-free addressing. His approach relies on capabilities being readily identifiable in both memory and processor registers. A machine operation which expects a register to contain a reference to an object will complain if the register does not contain a capability; on the other hand if it does, the machine will evaluate the capability to acquire the name within it and hence the representation of the object to be accessed. On entry to a domain, a processor register is loaded with a capability for the domain descriptor so that by addressing with this capability it is possible to access the other capabilities in the domain. It is important to note that unlike capability register machines, it is not necessary to consider in advance the allocation of capabilities to capability registers; instead corresponding to loading an address as data from memory into a register in a conventional machine, there is the action of picking up a capability. Thus the distinction between capabilities and data serves two purposes: firstly, to prevent a capability from being corrupted or forged and secondly to indicate to the addressing machinery that an item can be used as a valid address.

The scheme has the advantage that there are no problems concerned with shared addresses between domains and processes because the capabilities provide a global address space and do away with the need for virtual address translation. It is still possible to carry out the relocation of segments in virtual memory by modifying the contents of the global object table so that the scheme retains the power of a conventional virtual addressing scheme. Unfortunately, to implement this very pure scheme, it is necessary to use tagging to mark capabilities because it is unreasonable to expect data structures that contain a mixture of

capabilities and data to be partitioned into separate data and capability regions. As was pointed out earlier this can lead to a number of difficulties in an operating system.

CHAPTER FIVE.

TYPE-EXTENSION MECHANISMS.

5.1. Objects, Representations and Types.

In an object orientated system, the concept of extensibility as the introduction of further levels of abstraction corresponds to the provision of new objects beyond those provided by the hardware or kernel. These new objects must be protected by restricting the operations to which they may be subject in the same way that the kernel controls access to basic hardware objects such as processes and segments. Mechanisms are required for naming and describing abstract objects in addition to basic objects.

An important consequence of the layering methodology is that the kernel has no knowledge of its surrounding layers; indeed, if the dynamic creation of new types of objects is permitted, the kernel cannot have any built-in data about the range of objects that may exist. However, it would be unreasonable to have to implement parallel copies of the kernel protection machinery in every layer, both because of the implementation difficulty of ensuring that one layer cannot subvert another and the nuisance to users who have to cope with a multiplicity of mechanisms for manipulating objects. The functions of the kernel concerned with naming and protection, such as creating capabilities, copying them and performing access checks, can and should be available in every layer. This means that the kernel must be able to employ its capability mechanisms for objects which it is not able to interpret directly.

A particular layer in a hierarchical system builds upon the facilities afforded by lower layers and any new object that is introduced must be constructed from lower level objects which will form its representation. Objects made in this way are described as extended objects and the layers that implement them carry out operations upon them by manipulating their representations. Obviously this right must be denied to the users of extended

objects, otherwise they could undermine the layer implementing an object. It is therefore necessary for the abstraction mechanism to provide some means of concealing the representation of an object from its users, yet at the same time permitting the layer responsible for an object to get inside it. Furthermore, a layer should only be able to unpick the objects it implements and no others.

Objects can be partitioned into types; a type is an equivalence class of objects of identical structure, with the same operations defined for every member of the class. Every object has associated with it a type code which identifies the class to which it belongs. Typed objects are similar to 'classes' in Simula [Dahl and Hoare 72] or 'clusters' in CLU [Liskov 76]. The part of a system that implements a particular type is known as the type manager for that type. The term typed object is often used as a synonym for extended object and a collection of typed objects together with their manager is sometimes known as a protected sub-system. The primary motivation for an extensible system is to enable users to tailor its basic facilities to suit their requirements by the construction of protected sub-systems for additional types of objects beyond those already provided.

5.2. The Use of Protection Domains as Extended Objects.

The CAP [Needham and Walker 77] and Plessey System 250 [England 74] have a simple way of protecting the representations of extended objects that requires no additional machinery outside of memory protection, which is to embed extended objects within protection domains. The security of the representation of an object shielded by a protection domain is guaranteed if users are only given the right to call the domain so that they cannot tamper with the contents of its environment. Operations upon an extended object are carried out by calling it, with an entry code to identify the service required and the code executing within the domain uses the privileges available to it to modify the representation of the object accordingly. It is easy to see that this scheme is extensible because the protection domains describing objects whose representations are also extended objects, will contain domain capabilities for their components.

A protection domain that is used in this fashion to stand for a protected object can be viewed as an instance of a type manager with the identity of a particular object bound to it. Thus, for objects of a given type there will be distinct copies of the domain responsible for the type containing different representation capabilities. However, this does not imply that there will be multiple copies of code of the type manager and its data structures as they can be shared by virtue of the normal capability mechanisms.

The control of access to the extended object, as opposed to its representation, is carried out by building into the domain information about the operations that it is willing to carry out. If, as is the case with the Plessey System 250 and early versions of the CAP, access information is built into the data structures of a protection domain, there is a lack of uniformity with the kernel access control primitives. In particular, when users interrogate the access code of an extended object, they will only be told about privileges relating to the object as a domain and access codes for objects can only be obtained by calling the domains implementing them with an entry code which signifies "what is your access code?". To make a capability for an extended object that has weaker privileges, it is necessary to create a new copy of its domain containing a reduced access code.

More recent versions of the CAP system hold access codes for extended objects within domain capabilities and when a domain is called, the access code contained in the capability that was used to address it is loaded into a register so that it can be inspected by the program executing inside the domain. This artifice has the advantage of homogeneity with the primitive mechanisms for querying access states and making reduced privilege copies of capabilities, but it is only possible because CAP has no intrinsic access codes associated with domain capabilities.

A domain capability does not convey any information to the kernel about the type of object it protects; to the kernel type checking mechanisms, an extended object will always be simply a domain. For the benefit of users, a type code can be embedded in

the capability for an extended object, if there is room, or as an alternative, a type code can be built into the data structure of a domain in the same way that was suggested for access codes. To prevent bogus domains from masquerading as bona-fide extended objects, type codes must be well protected. If they are part of a domain capability there is obviously no protection problem, but the lack of space for holding the bits of a type code will reduce the size of the type code space and this in turn will affect the extensibility of the system. To prevent forgery, type codes kept inside a domain must be held as special type capabilities that can be inspected by the kernel. The lack of consistency of these devices with the type conventions of the kernel makes them less than perfect.

The freedom with which capabilities for extended objects can be passed between processes depends upon the willingness of the system to support independent domains that can be called by any process and implicitly includes provision for several processes to be executing concurrently in a single domain. This can be very difficult to arrange, as each process running in a domain must be given its own workspace so that it cannot interfere with any other processes that are present. Most protection systems do not permit multi-threaded protection domains (this point will be returned to in Chapter Seven), instead, every process is given a copy of a shared domain with the process's workspace bound into it.

There is a further restriction on protected procedures in the CAP system which greatly impairs their utility as extended objects. It was indicated in Section 3.2 that it is not reasonable to expect the CAP inter-process message system to construct copies of a protection domain dynamically because of the work involved and thus it is difficult to pass extended objects between processes. In the CAP system, an extended object is sent to another process in the form of its filing system name that can be used to retrieve a filed prescription which specifies how to create a copy of the domain. The overheads of this jury rig mechanism are rather high and the frequent use of it is not to be recommended.

A further point of interest arises from the observation that type managers typically retain a lot of information about the process in which they run for the purposes of collecting statistics and charging. If a domain is unattached to a particular process, it cannot make use of local memory within a process to hold this sort of information. In the CAP system and others (MULTICS, CAL-TSS) considerable use is made of this Algol own like storage for accounting and housekeeping purposes. It is not clear to what extent it is strictly necessary, as it would be perfectly possible for an independent domain to keep, within its own space, a record for each process that calls it.

A less important restriction owing to the use of domains as extended objects is that it is not possible to carry out other than monadic operations, because a domain stands for a single object. For example, services like CLOSE ALL FILES or FILE TO FILE COPY or even CREATE A FILE cannot be implemented unless they are posed as operations upon a single object, which will seem artificial to users. The CAP operating system frequently splits the functions of a type manager into two parts to circumvent the prohibition on multiple operands. For example, in the CAP I/O stream system there is a protected procedure, the I/O Controller, which creates new streams and keeps a record of which streams are attached to devices and so on. Users see a stream as a Stream Protected Procedure holding the representation of the stream, either as a message channel to a device or a segment in the filing system, with operations such as OPEN, CLOSE and TRANSMIT BUFFER that affect the contents of the stream.

5.3. Sealed Capability Type Extension.

To overcome the difficulties associated with using domains as protected objects it is necessary look for some means of constructing a capability for an extended object that is recognised as such by the kernel and can be passed around freely while, at the same time, it must be possible for a duly authorised type manager to use the extended capability to get at the representation of an object. In his thesis, Redell [74] surveys a number of proposals for describing extended objects. He shows

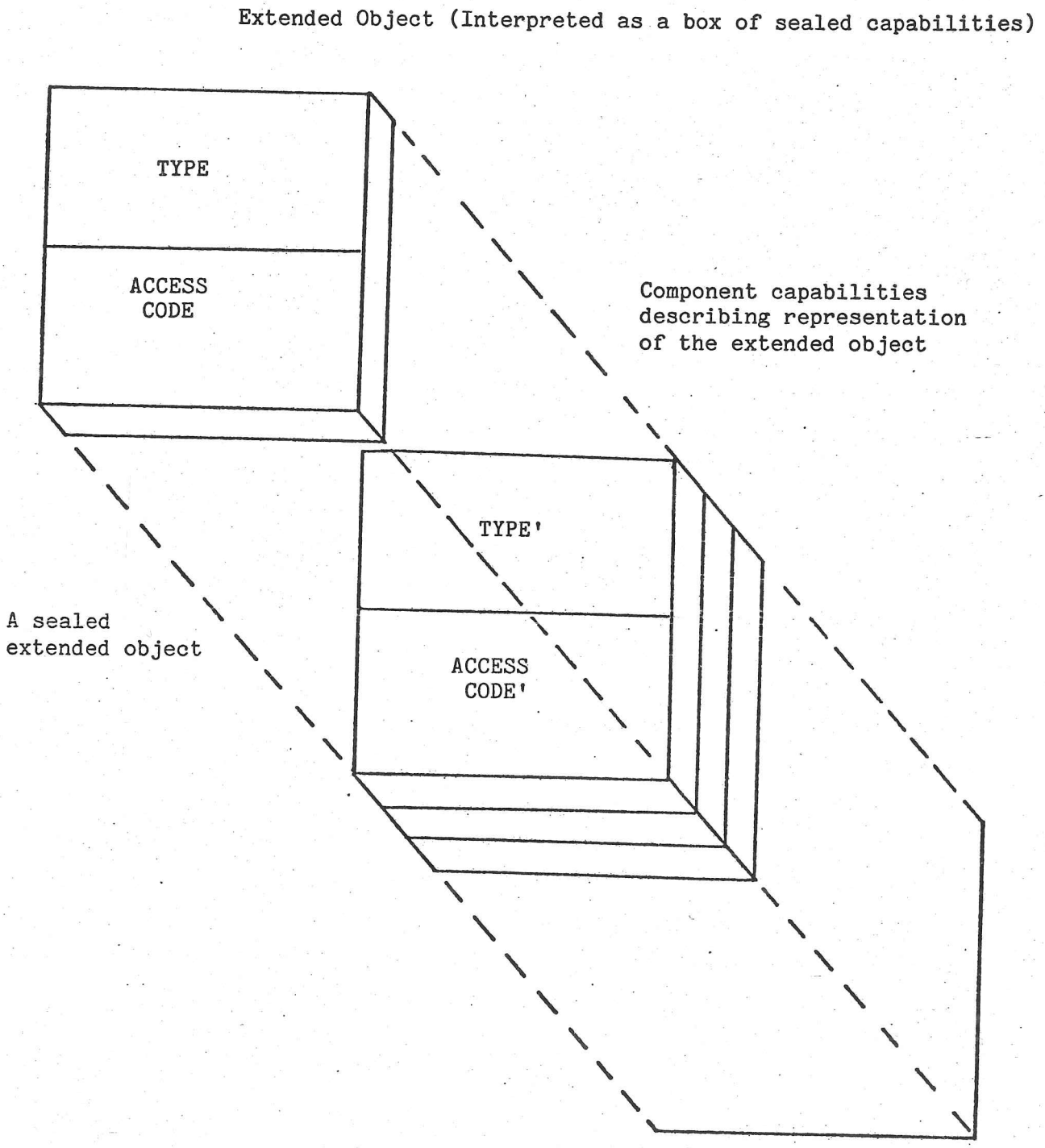


Figure 5.3-1 Capability Sealing

that the most reasonable mechanism is that based on 'sealed' capabilities, in which a capability for an extended object can be viewed as a box with capabilities for the objects from which it is made sealed inside [Lampson 69]. The front of the box is labelled with the type and access code of the extended object and its contents are concealed from users, although a type manager can be given the privilege to acquire the capabilities held within it. Type-extension is portrayed by the act of nesting boxes within boxes as illustrated in figure 5.3-1.

HYDRA is a practical example of a system that has a sealed capability type-extension mechanism and its essential principles are outlined in Jones' thesis [Jones 73]. As was described in Chapter Three, HYDRA has a global table with an entry for every object which is marked with the object's type. An entry for an extended object additionally contains a pointer to a capability segment that holds capabilities for its components or constituent rights. The environment described by these capabilities is never used for executing instructions, in contrast to the environment of a protection domain used as a protected object; instead it is purely a repository for capabilities.

When an object is created, it is not sufficient just to issue a name by allocating space in the global symbol table, in addition the environment for its components must be set up and initialised. For this purpose, there are two kernel primitives called LOAD and STORE: LOAD permits a capability to be copied out of the environment of an object and put in the current domain, provided that the object was addressed with a capability possessing load access; STORE, in conjunction with the store access code, is used for the converse operation of copying a capability from the current domain into the environment of an object. Both load and store are generic access codes that are defined for all types of extended objects, and the interpretation of other access codes in a capability depends upon the type of the object it protects.

Users of protected objects are not granted the potent load and store privileges and they see an object as a single atomic whole. Type managers may use the process of amplification to acquire their privileges for an object. Amplification is controlled by a

template in the form of a triplet <type, required code, amplified code> which is used as follows: if the type of a capability for an extended object matches 'type' and the capability possesses at least 'required code' in its access code, a new capability for the object is created holding 'amplified code' in its access code. By the use of amplification, it is only possible to increase the degree of access permitted to the extended object, usually to include LOAD and STORE, and the access codes of the capabilities sealed within the object remain unaltered from the values they had when they were last stored. Otherwise, if this was not the case, it would be possible to use amplification to acquire illegally new privileges in capabilities by the sealing mechanism. Templates are protected by storing them in capability segments and treating them as prototype capabilities.

The HYDRA scheme outlined above ideally fits into the criteria for type-extension supported by the kernel because the amplification mechanism does not rely upon any knowledge of the representation of extended objects and the kernel is only involved to the extent of matching type codes in the global symbol table entries and templates. HYDRA actually has a far more extensive set of access codes for amplification, that permit different sorts of restricted access to the components of an object than the simple load and store privileges [Cohen and Jefferson 75], although the same principles hold in their use.

5.4. Types as Objects.

The integrity of the type-extension primitives based on capability sealing rely on the authenticity of type codes, which, like names, should be unique and insubvertible. In an extensible system it must be possible to cope with a potentially large number of types. If the set of type codes is limited in size, there are likely to be severe resource control problems. The management of types can be made a kernel function by encoding type codes as names for type objects and thereby provide both protection and a suitably large type code space.

A type object is used to stand for the entire class of objects of its type and the privileges in a type object capability refer

to the class as a whole; for example, the implicit type object capability found in an amplification template controls the ability to manipulate the representation of a class of objects and it is possible to propose other access codes, such as create which permits the size of the class to be increased.

Just as objects of the same type form an equivalence class, type objects also form a class. It is possible define a 'master' type which stands both for itself and for all other type objects and is used to control the creation and proliferation of types. In this way, there is a hierarchy of objects as shown in figure 5.4-1. The 'master' type object is the property of the kernel and describes the class of types, each member of which denotes in turn a distinct class of objects.

There is another hierarchy in a layered system corresponding to the partial ordering of types imposed by the increasing levels of abstraction. The hierarchy is not a tree like that of objects and types, instead it is a directed graph containing no cycles. Figure 5.4-2. (based upon Redell [74]) shows a selection of basic and extended objects and the relationships between them. The extended types 'text file', 'sorted file' and 'linked list' are represented as 'segments' and a 'document' can be any of these types. The arc joining 'documents' to 'segments' reflects a possible implementation of a long document as a segment of capabilities for smaller documents.

5.5. A Simplified Scheme.

Redell proposes a simpler variant of the HYDRA mechanism which retains the same power over the control of objects and types. In his scheme, an object may only be represented by a single capability - objects with many components can be represented by a segment of capabilities for their constituents - and this capability is held in the global object table entry for the extended object so that the kernel does not have to administer a pool of storage for variable length representation capability lists.

As in HYDRA, types are represented by type objects for which there are two recognised access codes: seal and unseal. Extended

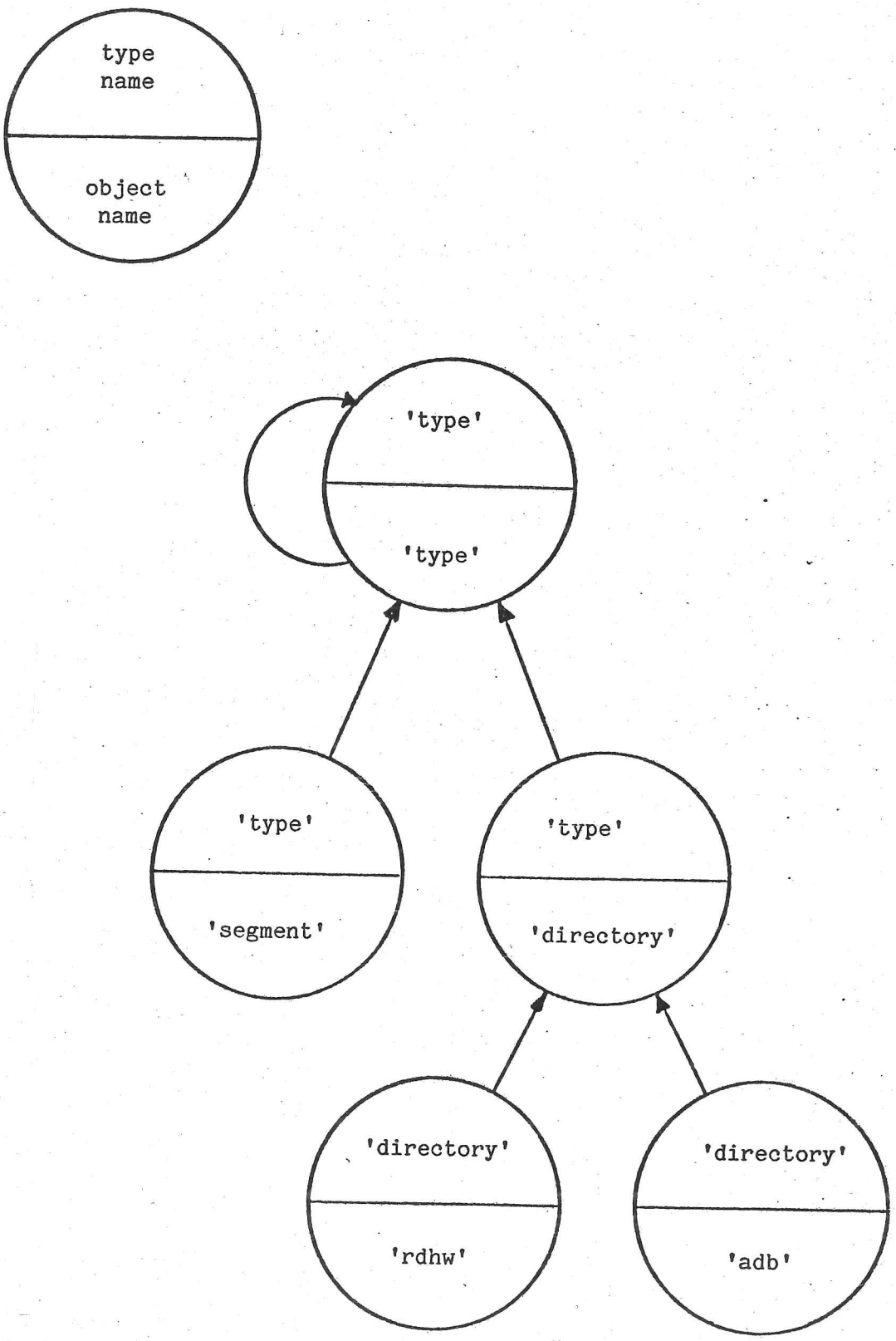


Figure 5.4-1 The Hierarchy Of Object and Type Names

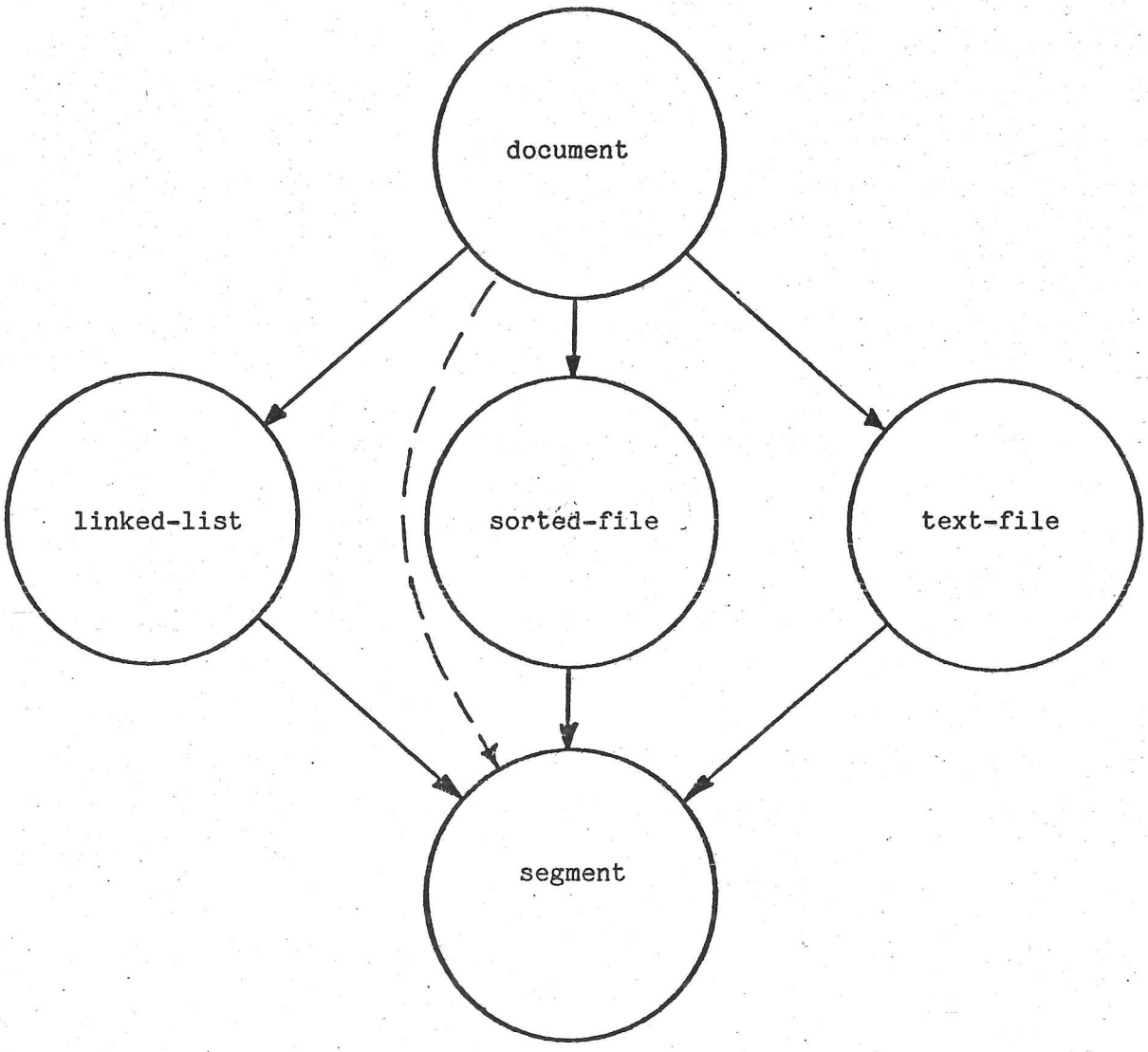


Figure 5.4-2 A Hierarchy Of Abstraction

objects are created by the SEAL kernel primitive which takes two arguments, a capability for a type object with seal access and a capability for the representation of the object. The kernel acquires a global object table slot for the new object and sets the type field to be the name of the type object and its contents to be a copy of the representation capability. The result of the entire operation is a capability for the newly constructed extended object as shown in Figure 5.5-1. The sealed capability is not visible to the user, who sees the extended object as a wholesome entity. The owner of a capability with unseal access for a type object can interrogate the representation of any member of the class using the kernel UNSEAL primitive which delivers a copy of the representation capability sealed within an object as its result.

This mechanism has the advantages of simplicity, flexibility and, as will be seen in the next chapter, considerable unity with a powerful revocation scheme. What the scheme lacks is any facilities for creating basic hardware objects that have a data rather than a capability representation. For these objects there is a common requirement to modify their representations; for example, to relocate a segment in store by altering an absolute address in its map entry. Thus for these reasons it is necessary to augment the basic set of orders provided in Redell's scheme to arrive at a full suite of operations for both basic and extended objects.

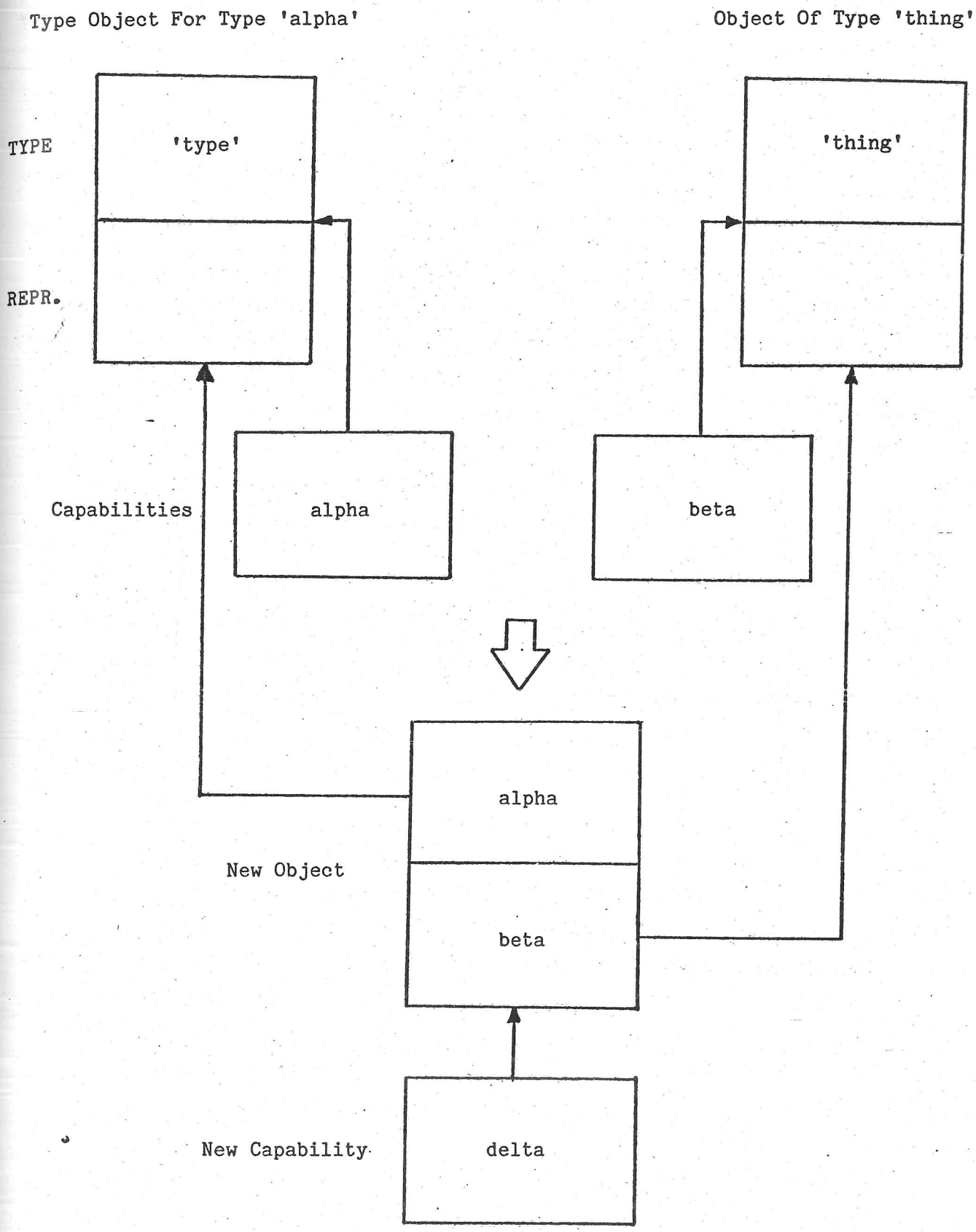


Figure 5.5-1 Redell's SEAL Operation

CHAPTER SIX.

REVOCATION MECHANISMS.

6.1. A Review.

In a capability-based system, one user can allow another to have access to an object by passing him a copy of a capability for it and in the interests of his own security, the original owner might well pass a capability with weaker privileges than his own. If the act of sharing corresponds to renting use of the object temporarily rather than to mutual cooperation, the level of trust the owner of the object has for the client is likely to be reviewed, especially if the latter neglects to pay a rental fee or some such thing. In these circumstances, the owner will wish to recall the privileges that he gave the client earlier, although perhaps only until the client redeems himself by making a suitable payment. To handle this sort of situation, there is a the need for revocation of access that takes immediate effect. Note that revocation may only be temporary and need not necessarily cause the loss of all privileges for an object, but only a subset of them.

The situations outlined above demonstrate that revocation is closely bound up with the notion of ownership. It is generally understood that the owner of an object is the user or funding agency which the system charges for storing and manipulating it. Some systems only permit an object to be owned by a single user or principal, although in real life there are much more complex patterns of ownership; for example, a database jointly owned by a group of cooperating users. Ownership need not be static and can itself be considered as a privilege that can be shared. Some everyday analogues of passing on the right of ownership are sub-letting and bill collection. In the former, the user of an object passes on a copy of his capability to another user, who may use revocation to restrict the privileges available to a set of sub-users independently of the original owner, who in turn can restrict the activities of all the users. The second example, bill collection, corresponds to a user passing the right of

control over an object to an agency that revokes access for users who default on their debts and, naturally, the original owner of the object will wish to reserve the right to withdraw the agency's revocation privileges when he has done with their services.

The notion of ownership provides a means for describing the use of revocation, but it does not provide a mechanism for carrying it out. An important property of capability systems is that a privilege can be passed between domains or processes quite freely and when the owner of an object revokes access to it, the kernel is obliged to stop all programs executing until the access code in every capability for the revoked object matches the object's changed status. Obviously it would be madness to scan exhaustively every capability in the system to be sure of finding those to check; nor is it reasonable to consider implementing a scheme of pointers from parent capabilities to their descendants because, as a capability can be copied many times, vast amounts of memory would be needed to hold all the pointers. The need to locate and modify many distributed copies of a capability is the fundamental problem of revocation and, the solution lies in the formulation of revocation as an operation involving the mapping between capabilities and objects.

For the time being, it is instructive to turn to some of the systems implications of immediate revocation. The first observation is that any service, and especially the operating system, must be prepared to find that a capability it has been passed as an argument might suddenly become impotent because an asynchronous process has carried out a revocation operation. If the service was in the middle of a critical section, there is a high probability that the revoked object would be left in an inconsistent state. The standard remedy against this effect is to ensure that a program accesses its arguments once only to take a copy of them for use as data in its computations and any update operation on a data structure must be done atomically so that revocation cannot prevent it from completing.

Revocation poses some particular problems in systems that support protected objects. Uniformity requires that it should be possible to revoke capabilities for extended as well as basic

objects. Furthermore, it should be possible to incorporate revocable capabilities into the representations of objects, which means that the revocation machinery must be able to monitor these capabilities as well as those in protection domains, otherwise a user can shield a capability from revocation by disguising it as an extended object.

There are some circumstances in which it may be desirable to delay the effects of revocation; for example, consider a capability for a protection domain: if revoking the capability is defined to withdraw immediately the right of execution within it, the system is obliged to identify all of the processes running in the domain and force them to exit straight away. Furthermore, the domain has no opportunity to tidy up or recover from the interruption of its duties and may well be left in an irregular condition. This point can be addressed by defining the sequence of instructions executed between a domain call and exit to be atomic with respect to the domain making the call, which will mean that revoking a domain capability will prevent any further call to it being made, but any call that is in progress is allowed to finish.

Within the context of an operating system there are two main uses of revocation: the first is the reflection of revocation operations in a filing system by the immediate revocation of access to versions of filing system objects that are active in the machine, and the second is to prevent malicious domains from retaining copies of capabilities that they were passed as arguments and from interfering later on with the objects that the capabilities protect. The rationale of immediate revocation in response to filing system operations stems from systems like MULTICS [Organick 72] that have one level filing systems in which segments are swapped between main memory and the filing system, whereas two level systems copy segments out of the file system into an autonomous virtual memory swapping regime. In a one level system, revoking access in the file system automatically includes revocation of active objects because of the intimacy of the swapping and filing systems, whereas in a two level system, there is a considerable amount of work to be done to find the active

copies of a filing system object. This can be avoided by declaring that revocation only affects subsequent filing system accesses to objects and leaves any currently active versions of a revoked object alone. This latter approach to revocation has the advantage that it need not involve the kernel and can be carried out entirely at the level of the filing system.

Unfortunately the second use of revocation, the control of capabilities passed as parameters to a domain, does require the intervention of the kernel because of its association with the domain call mechanism. The problem to be solved is one of confinement [Lampson 73], in that it is required to control the proliferation of any capabilities that are passed to a domain, in particular to ensure that it neither retains an argument capability in its own storage nor covertly hides it away in some other domain. It is not reasonable to rely on the use of generic access codes that prevent a capability from being copied as this hinders a domain that legally passes its arguments on to other domains in the course of its actions.

Domain call parameter revocation is less disastrous from the systems point of view than its immediate filing system counterpart because it only takes place on domain exit, and does not interfere with the program running within the domain.

Even with the use of revocation, it is not generally possible to prevent a domain from remembering information that it is passed and leaking it elsewhere. Revocation can only be used to confine the proliferation of privileges; it is no use at all as a mechanism for preventing the flow of information and data. It is necessary to look towards the analysis of information flow [Denning et al 74, Fenton 74] to evaluate the possibility of a data leakage from a protection domain.

6.2. Revocation in Capability-Based Systems.

It might seem that the simplest way to allow one domain to revoke privileges that it has passed to another domain is for the former to have complete control over the capabilities of the latter. This assumes that the domain which is given a revocable capability has complete faith that its controlling domain will not

take advantages of its privileges and interfere with any other capabilities, apart from those which are to be revoked. Coordinators in the CAP system have a relationship of this sort with respect to their sub-processes and they can revoke a capability in a junior process by modifying the capability for the revocable object in the Coordinator's address space. Notice that, because of the nested address spaces in CAP, this mechanism revokes access in any copies of the capability in all the sub-processes of the Coordinator, so that a sub-process cannot cheat by hiding a revocable capability in any of its brother processes. If CAP had a global naming scheme instead, the Coordinator could still use its position in the process hierarchy to get inside the processes it controls to carry out revocation, but the task would then be much harder as it would need to scan every capability within the controlled process, which might have taken many copies of the target capability. Furthermore, the controlled process would have to be confined so that it may not transmit a revocable capability to a process over which the Coordinator had no powers.

An alternative mechanism, is to encapsulate a revocable capability in a domain that monitors all access to it. Mutual suspicion is now handled successfully because the shielding domain has no control over the domain which calls it, but the scheme is faced by a number of difficulties not very dissimilar to those encountered with the use of protection domains for typed objects. There is the problem of type recognition; to the base level, a revocable capability for any sort of object always appears to be of the type 'domain', although the confusion can be avoided by using the auxiliary type marks that were suggested in Section 5.2. More important, there is a loss of efficiency caused by the time taken by the domain to check and interpret every service it is asked to do, and in the case of kernel-defined objects such as segments the losses are immense.

If copies of a revocable capability are given out to a number of different users, it is necessary to consider how a caretaker domain distinguishes between them. A simple approach is to pass to each user a separate copy of the domain with his access status

bound into it, otherwise the domain must recognise each of its callers and check their corresponding access codes, which is a direct copy of the kernel protection facilities and a wasteful duplication of machinery.

The two schemes outlined above are the only two revocation strategies available for users of the CAP system, the designers of which felt that revocation was undesirable as a matter of policy and were not willing to add any additional machinery to handle it. Redell's thesis [74] investigates many of the possible paths that may be followed if further mechanisms are introduced into the kernel specifically to handle revocation. The systems he describes fall into two classes: revoker capabilities and dependent capabilities.

Revoker capabilities [Neumann et al 74] are capabilities for the mapping between a capability and the object it names. A revoker capability can be used to alter the mapping and vary the accesses conveyed by capabilities that map onto objects through it as illustrated in Figure 6.2-1. In effect, the mapping between a capability and an object is itself treated as an object which suggests as an implementation that a revoker capability will map onto a revoker object whose representation describes the mapping for another object. The main difference between this and the gate-keeper domain scheme is that the revoker capability does not describe an active object which guards all access to the revocable object, instead it is a contrivance for interfering with the mapping between names in capabilities and entries in object tables. A corollary of this is that the privilege of revocation can itself be made revocable by controlling the mapping between revoker capabilities and revokers.

The dependent capability scheme is rather different; there are no special revoker capabilities, but instead it is arranged that all copies of a capability are dependent on the original so that when the holder of a capability revokes access, all of the copies dependent upon it are similarly affected, which is to say that capabilities somehow depend on the source from which they were derived. In this approach there is a distinction between transmitting a plain copy of a capability and a revocable one.

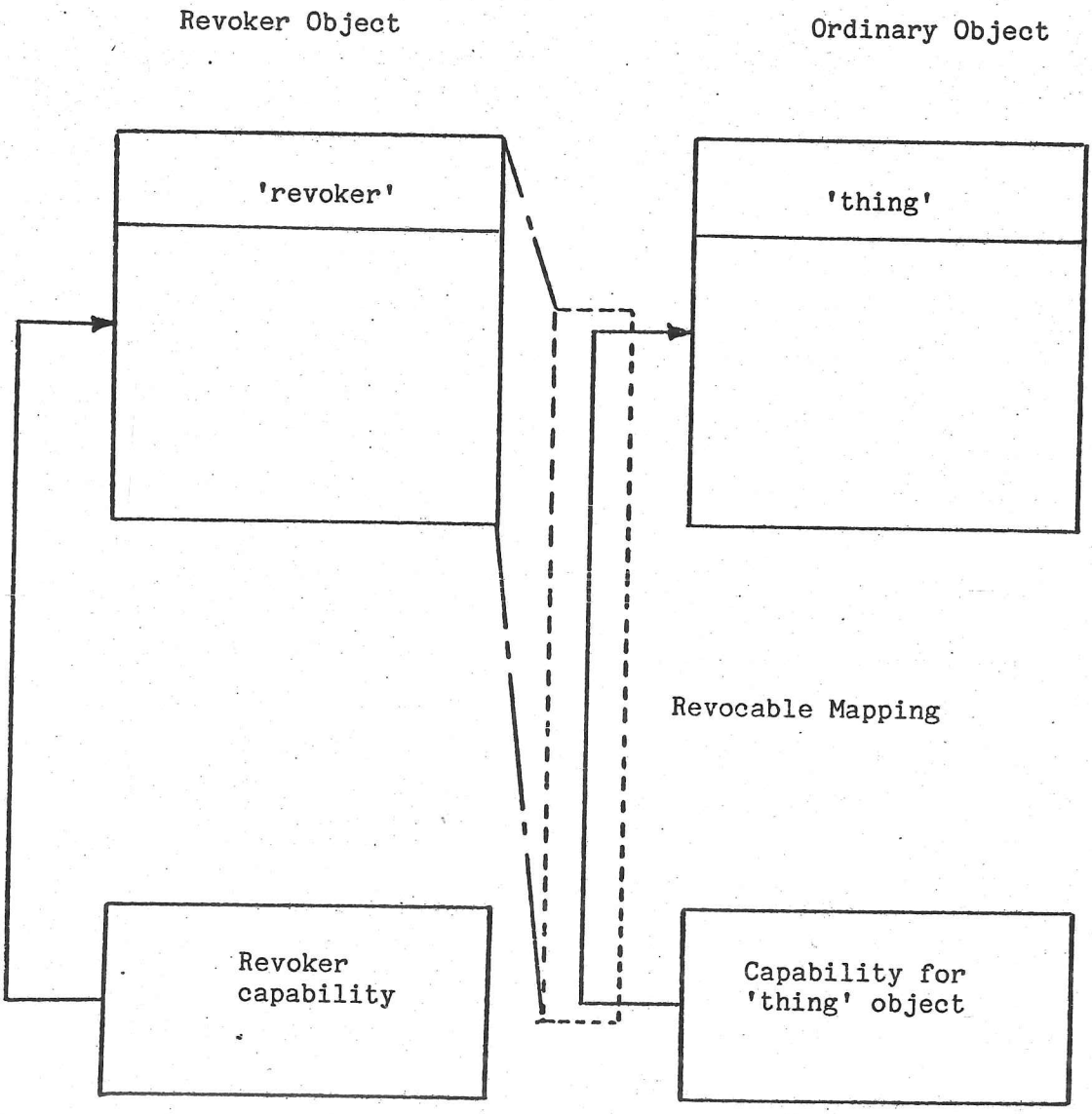


Figure 6.2-1 Revoker Capabilities

This distinction establishes a tree of dependencies between copies of a capability which is structured as follows:

a) The initial capability occupies the root node.

b) A non-revocable copy of a capability occupies the same node as the capability from which it was derived.

c) A revocable copy of a capability occupies a new successor node descended from the node of the original capability.

A diagrammatic representation of a typical tree is shown in Figure 6.2-2. All of the capabilities in an individual node of the tree always contain the same privileges, since any change to one of them affects all of its companions equally because they are all dependent on the capability from which they are descended. If a revocation alters a privilege at some level in the tree then privileges are affected in the levels descending from it. The main point to notice about the tree is that it demonstrates that with the two different copying primitives, dependent capabilities pose no constraints on the use of revocation because the tree describes a general hierarchy of control.

Dependent capabilities have a great deal to recommend them. They avoid the need for special capabilities authorising revocation and also escape from treating the capability to object mapping as an object which is not straightforward to implement, although it does not mean that revocation itself cannot be made revocable. The main complaint against dependent capabilities is that an early decision is required to determine whether or not a capability should be revocable because, once a non-revocable copy is given away, all control over it is lost forever. However, it is reasonable to suggest that any level of trust apart from absolute confidence is liable to change and should be mirrored by the use of revocable capabilities at all times.

6.3. A Sealed Capability Implementation.

In the last chapter, the use of sealing to conceal the representation of an object was described; Redell calls this opaque sealing and it is also possible to consider transparent sealing in which a sealed object can be read, but cannot be modified. Redell describes a mechanism based on a mixture of these two types of sealing to implement a dependent capability

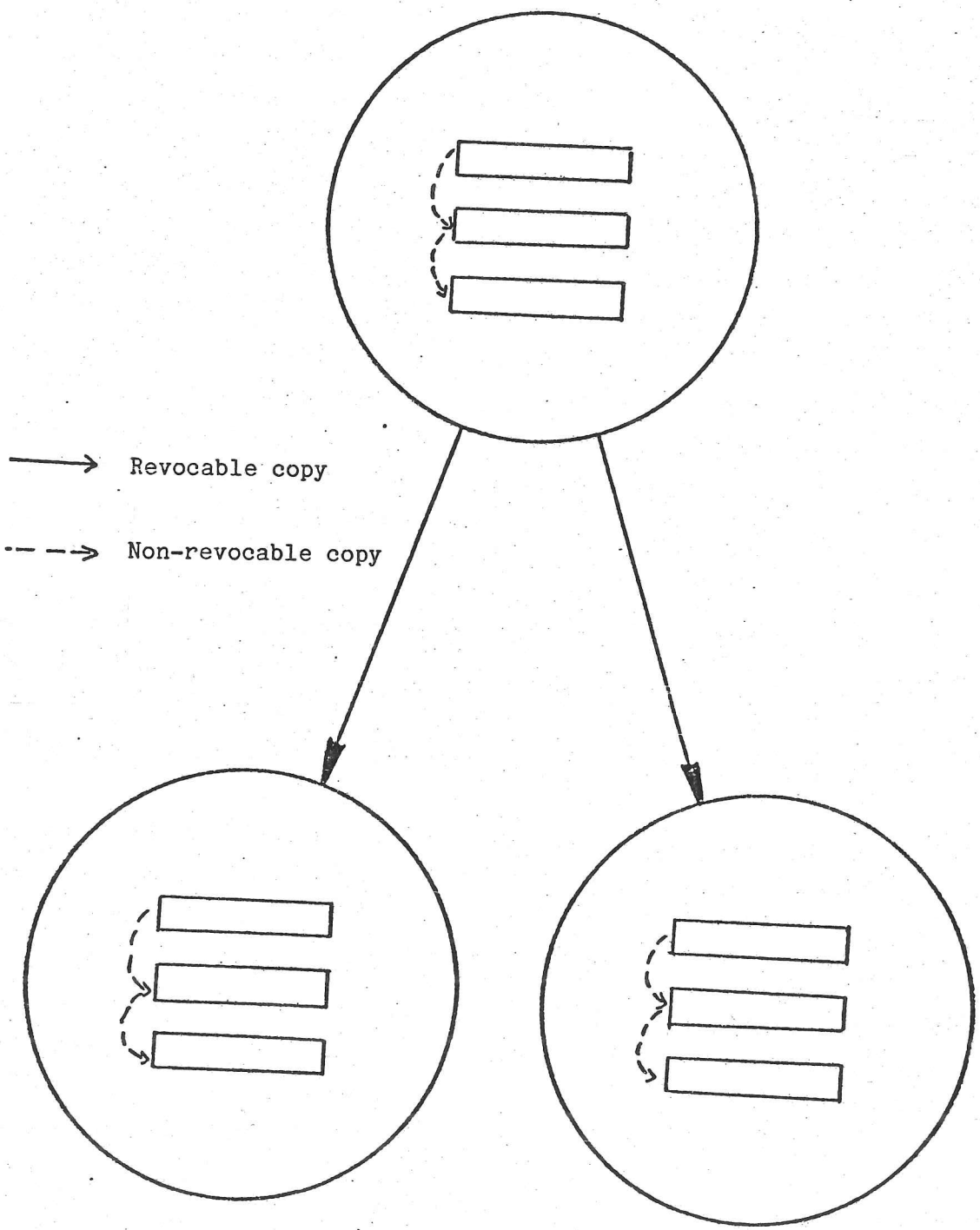


Figure 6.2-2 A Hierarchy Of Dependent Capabilities

revocation scheme. He introduces a new sort of slot, called a revoker, in the global object table which resembles an extended object, except that its type is recognised by the kernel. Capabilities for revocable objects are illustrated in Figure 6.3-1. A revoker contains a capability together with an access mask and every field in a revoker, with the exception of the bits of the access mask that are off, is transparent. If, in the course of the evaluation of a capability, a revoker is encountered, the transparent capability sealed within it leads on to another table global object table entry which might also be a revoker until eventually either a basic or an extended object is found, which is taken to be the object the original capability denotes. The opaque parts of the access masks, that is the bits that are off, cut out accesses that are not to be permitted and this selective filtering action is used to capture the action of revocation.

Redell introduces a kernel primitive, REVOKE, which takes two arguments: a capability and an access mask. If the name in the capability points immediately at a revoker, the kernel modifies the access mask of the revoker to be the intersection of the access mask sealed in the revoker and the access mask argument, otherwise it signals a fault. Whenever any capability pointing at the revoker is subsequently evaluated, the privileges it conveys will be tempered by the new access mask so that if, for example, every bit in the mask was off, the effect would be one of total revocation of privilege. Thus, the main difference between dependent capabilities and revoker capabilities is that revocable dependent capabilities may be used to access the revocable object, but revoker capabilities may not.

It may be noted that Redell's scheme in this form only allows access to be reduced; there is no mechanism for temporary revocation and it will be shown in the design of the CAP kernel (Section 10.4) that it is only necessary to make a few changes in order to remove this restriction.

So far, the mechanism developed permits capabilities that can be revoked to be established by sealing in the presence of a revoker type object; some additional mechanism is required so

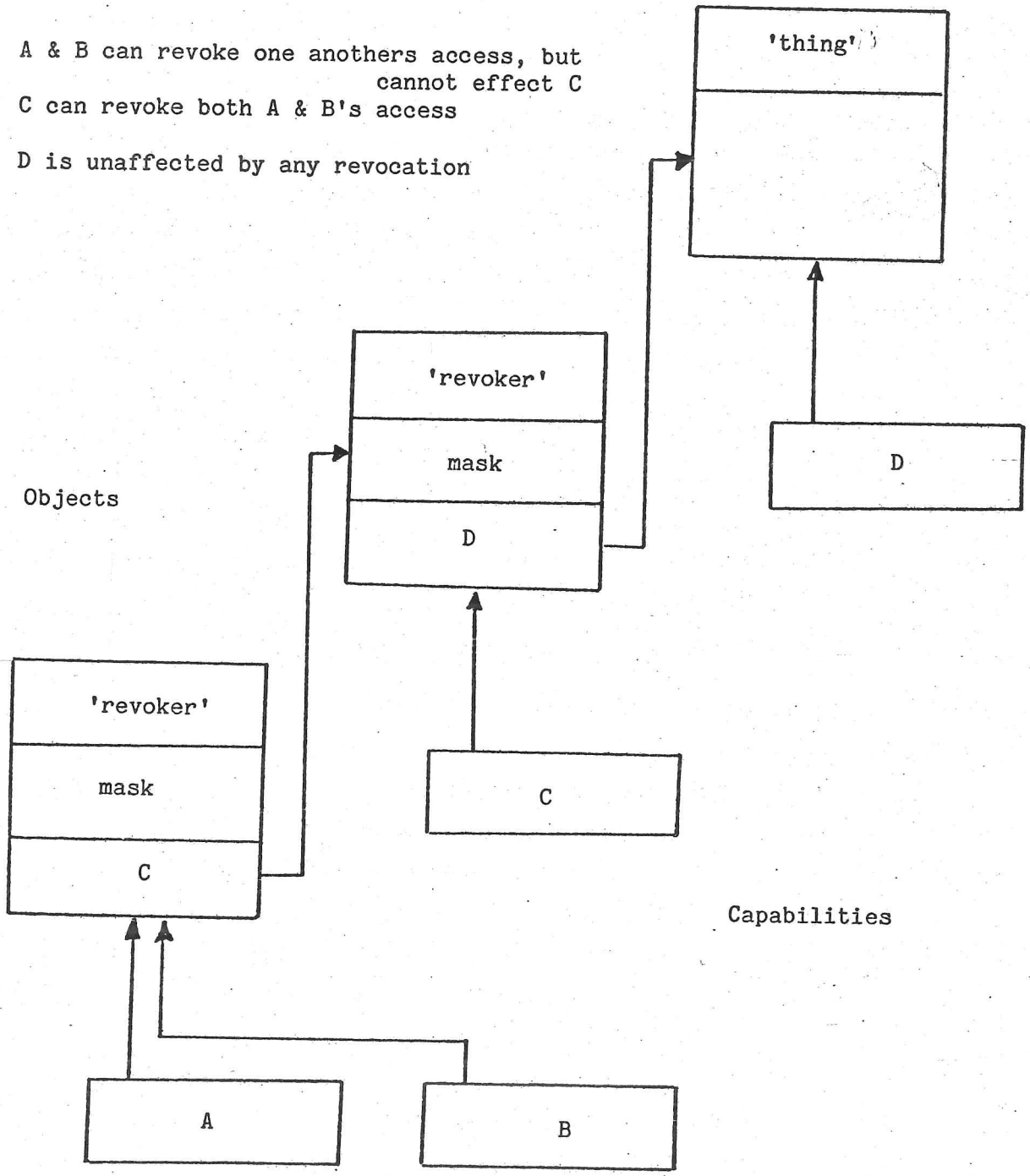


Figure 6.3-1 Redell's Revocation Scheme

that it is possible to copy revocable capabilities, leaving the power of revocation with the owner of the revocable object, without allowing the privilege to others. For example, it might be required to pass copies of a revocable capability to several people without wishing to allow any one of sub-users to affect the powers of the others. On the other hand the owner of the object will wish to be able to deny access to the object to all of the sub-users. This is accomplished by an additional type of extended object called a locker which is totally transparent to the capability evaluation process. The only purpose of a locker is to prevent the REVOKE operation from being able to succeed, because the type of first table entry leading from a locked capability will be a locker and not a revoker. Thus, only the holder of a revocable capability can exercise REVOKE, although the its effect will be noticed by every capability that denotes a chain passing through the revoker controlled by the revoked capability. An example of this sort of sealing is shown in figure 6.3-2. The UNSEAL operation is not allowed for revokers and lockers because it is not a acceptable function.

In Redell's design, REVOKE is the only operation available for reducing access to objects because his capabilities do not contain access codes and therefore it is not possible to carry out any form of access code refinement as is carried out in the CAP system. A capability on its own denotes full privileges for an object and the access masks in any revokers intervening in the path between a capability and its root object table entry are the only means available for reducing access.

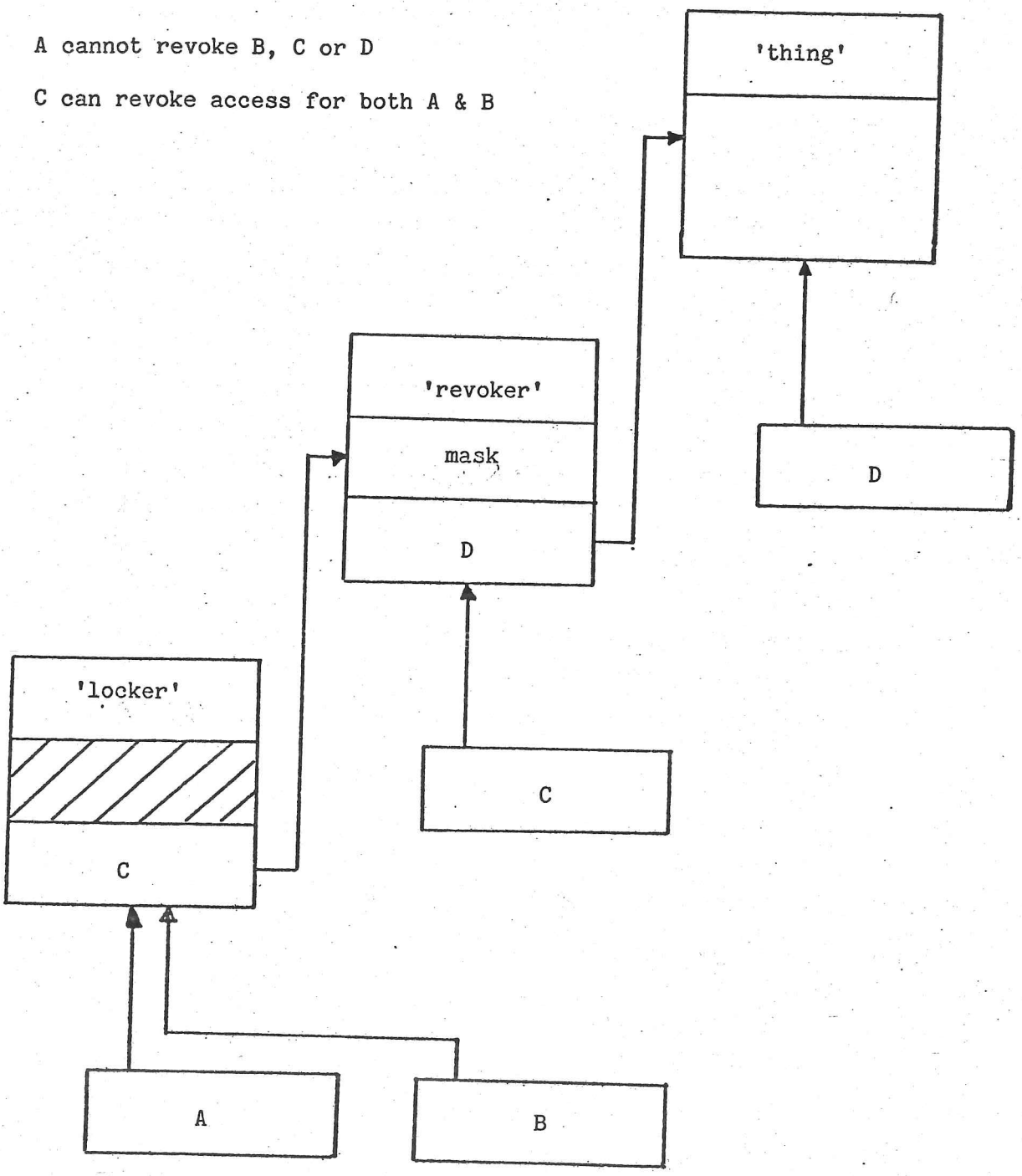


Figure 6.3-2 The Use Of Lockers

CHAPTER SEVEN.

PROCESSES AND PROTECTION DOMAINS.

7.1. Protection Domains.

A capability for a protection domain leads to an object whose representation holds information about the state of the domain and its capabilities in the form of a segment known as the domain descriptor. Because nothing is ever done by a domain in isolation, but always by a process executing within it, there is a relationship between domains and processes, which is called 'environment-binding' by Jones [73]. In the most general terms, domains and processes can be considered separately and processes allowed potentially complete freedom in their association with protection domains. This means that a process can move through many domains in the course of its execution and, on the other hand, allows many processes to execute concurrently in a single domain. Most systems impose restrictions on this total flexibility to reduce the amount of machinery needed for inter-domain and inter-process communication.

One simplification is to make a process into a single protection domain so that the inter-process communication facilities can also be used for inter-domain calls; this means that a task, which in the general scheme would have been a process with several protection domains, has to be implemented as a multiple set of processes, only one of which will be active at any one time. This is a clumsy use of parallelism and can be rather inefficient if inter-process communication is slow. In a traditional computer architecture, this is the only domain structure which exists and process switching is a slow and lumbersome task carried out by software. This discourages the use of small domains for reasons of inefficiency and leads on to contraventions of the principle of minimum privilege because processes (i.e. domains) will typically be large and encompass many activities. It is possible to circumvent these problems by making the cost of a process change small and by building a simple inter-process communication system which has the parameter passing

capabilities and the speed that is expected of a domain call mechanism.

The main reason for wanting no more than one process executing in a domain at any one time is because of the likelihood of addressing conflicts between parallel invocations of a domain [Lampson 69]. There are a number of ways in which multiple execution in a domain can be prevented, of which the simplest is to make an entire domain into a critical section, not unlike a Hoare monitor [Hoare 74]. However, it may well be that this introduces unnecessary serialisation for domains that are not critical regions. There is also a further problem concerned with the degree of parallelism associated with monitors: in general there are two forms of monitor call [Lauer and Needham 78]: one that diverts a process into a monitor directly and another that divides a process into two parallel forks, dynamically creating new workspace for both forks, only one of which enters the monitor. In a capability based protection system, a process has to carry a considerable amount of state information around describing its current set of capabilities and so on, which would make the cost of a forking monitor call prohibitive because of the expense involved in duplicating the protection structure of a process as well as its workspace.

Addressing conflicts in a domain that admits several processes at once can be avoided by one of two techniques, the first of which is to provide a stack-like implementation of dynamic workspace in each process, so that on calling a shared domain, a process can acquire its own local workspace and be free from interference from other processes executing in the same domain. The second technique is really a modification of this in that, instead of setting up the workspace dynamically, each process is given its own private copy of the domain with the process's local workspace built into it and relies on the normal capability sharing mechanisms to avoid the wastefulness of duplicate copies of pure code and data. Protected procedures in CAP are shared between processes in this way.

7.2. Inter-Domain Communication.

When a process moves from one domain to another, it needs to be able to pass capabilities as well as data parameters. It would not be reasonable for domains to pass arguments by putting them in shared segments because of the amount of memory that would be wasted setting up a buffer for every pair of communicating domains. Domain call mechanisms normally transmit privileges between domains by copying capabilities out of one domain into another: for example, in the CAP system [Needham and Walker 77], a particular capability table, number three, known as the N-capability table, becomes the number two, or A-capability, table in the called protected procedure after executing an ENTER instruction and capability parameters can be passed to the called procedure by copying them to the N-capability table prior to the call. This mechanism is illustrated in figure 7.2-1. When control returns to the calling protected procedure, the A-capability table of the called procedure reverts to its previous state as the N-capability table of the caller and result capabilities can be taken out of it. The switching of the capability tables is carried out by manipulating those pointers in the process base that describe the current set of capability tables and effectively amounts to copying the capabilities for the capability tables in and out of the domain descriptors of the protected procedures. In advance of calling a procedure, an N-capability table can be allocated dynamically from a stack by the MAKEIND instruction and then capability arguments can be copied into it. The stack, called the C-stack, is controlled by the microprogram and also holds linkage information for use by the RETURN instruction which will cause control to resume in the calling domain immediately after the ENTER instruction that invoked the domain call.

The HYDRA kernel has a more complex domain call mechanism [Cohen and Jefferson 75] although the principles are essentially similar to CAP. The main additional feature is the use of parameter templates, similar in form to the type-extension templates described in Section 5.3, to carry out argument checking. The domain call primitive compares each argument passed

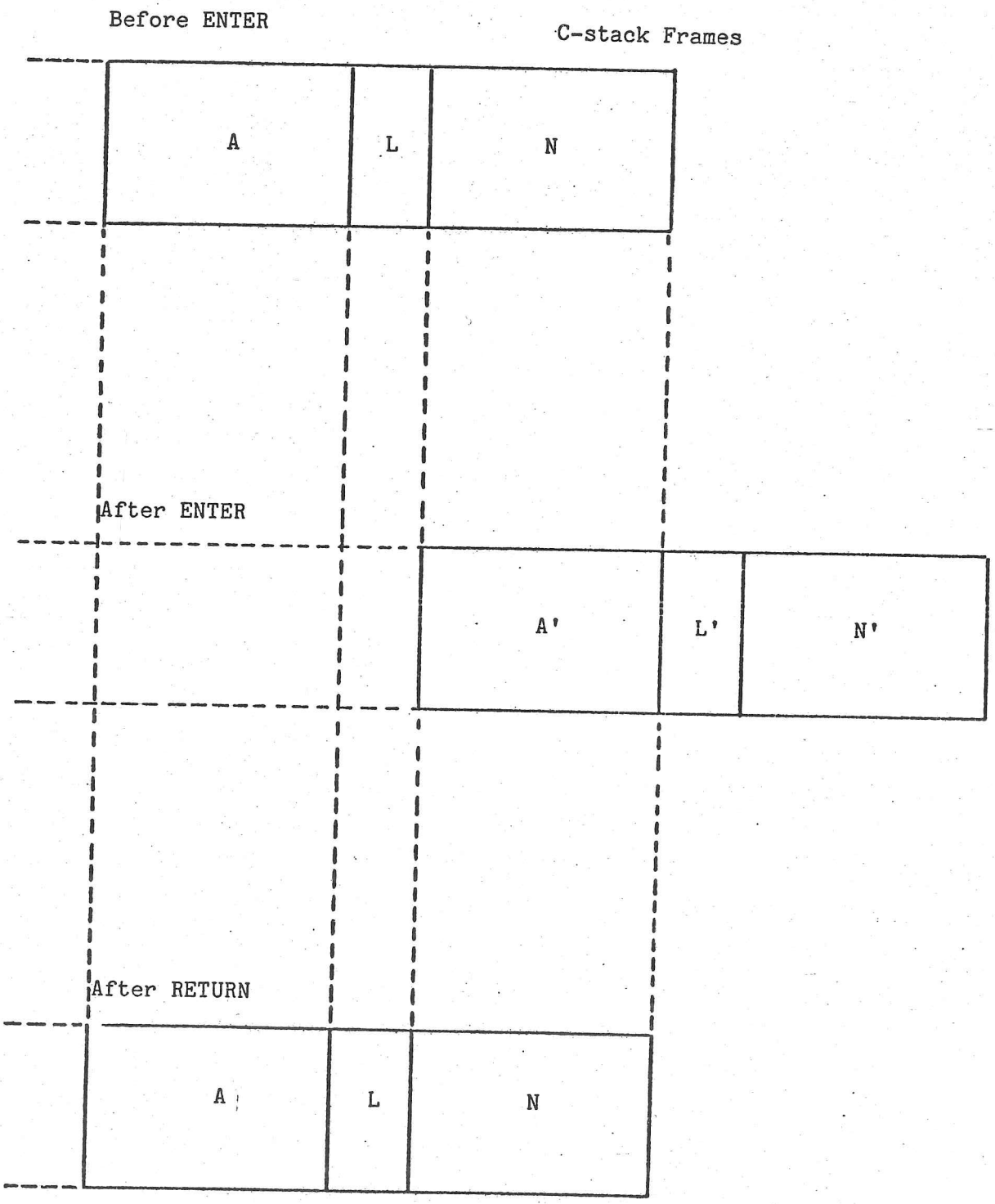


Figure 7.2-1 Action Of ENTER and RETURN

to a domain against a set of parameter templates in the domain object and signals a fault if an argument capability does not match the type of the corresponding template, or if its access code is weaker than the 'required access' field in the template. Provided that an argument capability is consistent with the matching template during a domain call, the domain is given a copy of the argument with the 'amplified access' field of the template included in its access code. The last facility is used extensively by type managers to acquire load and store privileges for extended objects.

7.3. The Use of Protected Domains.

There are four principal applications of protected procedures in the CAP system [Needham and Walker 77]: gate-keeping, protected objects, trivial services and operating system intervention. It is useful to look briefly at this spectrum with the aim of indicating how much the efficient domain machinery of the CAP system contributes to the success of the CAP operating system.

The first application includes domains guarding the use of other system facilities, such as the Enter Coordinator Procedure (ECPROC) which provides an interface to the Master Coordinator and performs validation checks on Coordinator calls. ECPROC is in a much better position to look at capabilities during a Coordinator call than the Coordinator itself, because it runs in the same name space as the process making the call, whereas the Coordinator would have to interpret sub-process addresses and duplicate the naming mechanisms of the microprogram in software. There is also a gain in efficiency because arguments are verified within a process and this reduces the amount of time spent with interrupts disabled and cuts down on the number of context switches between processes and the Coordinator.

The CAP system embeds system data structures into protected procedures that implement all of the operations allowed to be carried out upon the data which, as was stated in section 2.1, helps to achieve both minimum privilege and accountability. For example, the data structures of the inter-process message system

are guarded by ECPROC because the interactions between the message system and multi-programming require access to Coordinator data structures. This data must be protected from being corrupted by ordinary programs and also it is necessary to prevent users from tampering with the contents of messages.

In Section 5.2 the use of protection domains as protected objects was investigated and it was noted that, for CAP, this was the only way of implementing extended types. Even if the CAP had a more powerful protected object mechanism, protection domains will still be needed for use as type managers. This use of domains is rather like the protection of data structures described above, because a type manager encapsulates the privileges for getting into the structure of protected objects and performing operations upon them.

A somewhat surprising use of protected procedures peculiar to the CAP system, and directly attributable to the cheapness of the ENTER and RETURN orders, is the implementation of trivial services as protected procedures. CAP has a general purpose program called PARMS which takes a character string representation of a command line and will decode and command parameter strings from it. All protected procedures invoked by the CAP Command Program are given capabilities for PARMS, together with the command line that caused the procedure to be loaded and the procedure can call PARMS to decode its command parameters. PARMS is a protected procedure simply so that the interface to it is well-defined and straightforward in terms of the ENTER/RETURN and capability passing primitives.

Casting services of this sort as procedures is very useful in systems that support a multiplicity of languages because it avoids the need for one language system to have to know how to make subroutine calls in another, which would be the case if say PARMS, written in Algol68C, was called by a BCPL or a FORTRAN program as a subroutine. Instead, it is only necessary for each language to provide a mechanism for calling protected procedures and passing arguments, to make it possible to use service utilities written in any other language. From the point of view of both documentation and implementation, it is useful to have the common base level of

the hardware primitives for describing interfaces in terms of domain calls with simple numeric and capability parameters, independently of language considerations.

After a fault or trap, a process has often to be involuntarily forced into the operating system so that the event can be processed. In order to preserve the principle that the operating system should have as little access to user capabilities as possible, CAP makes the entry to the operating system take the form of simulating the effect of an ENTER at the point of the fault into a special protected procedure that inspects the trap and decides what is to be done. The procedure, called FAULTPROC, can then call other protected procedures to recover from the fault. For example, after a virtual memory trap ECPROC will call the store management system to load a segment into store and RETURN to the procedure from which it was forcibly called so that normal execution can resume.

In an evaluation of the CAP system, Needham [77] shows how the exploitation of protected procedures by the CAP operating system falls in line with the desiderata appearing in Chapter Two, and the conclusion that can be drawn is that the effectiveness of the CAP operating system is founded on the use of small independent protection domains. These domains exploit a very efficient domain call mechanism in which a domain call takes a time comparable to about one hundred ordinary instructions [Cook 78]. Software kernels like HYDRA, in which the time taken to switch between domains is measured in the equivalent of thousands of basic instructions, cannot match this performance and the operating systems built around them suffer accordingly.

The cost of domain calls can be cut down by making the parameter passing mechanisms as straightforward as possible. Much of the cost of a domain call in HYDRA comes from the parameter template machinery for checking arguments because it has to be sufficiently general to match most user requirements and naturally the price of this complexity is a high overhead. Simple transactions normally only involve a few trivial arguments and it is likely that user-written code within a domain, using knowledge of the nature of expected arguments can do a more efficient job of

parameter verification.

A lesson to be learned from the implementation of the CAP ENTER/RETURN orders is that the domain call operation itself should carry the smallest overhead possible when establishing a new protection domain and should leave tasks like evaluating the capabilities in the new domain and setting up its capability tables undone, until they are referenced by the code running in the domain. Furthermore, efficiency will be increased if it is possible to preserve as much as possible of the state of the calling domain, so that on return to it there is no need to re-evaluate the capabilities that were current at the time of the call.

7.4. Unified Communication Systems.

Inter-domain communication is based on a procedure call model. Inter-process communication, on the other hand, is more complex because it is bound up with the synchronisation of parallel processes. For the purposes of discussion, a simple system with processes communicating by messages using the primitive operations SEND, RECEIVE, REPLY and WAIT and domains using CALL and RETURN will be considered. Users of the system need to know in advance whether or not a particular module is either a parallel process or a domain in the current process so that they can use the appropriate communication functions. This can lead to great inconvenience if at some later stage it is decided to convert a module from a process to a domain or vice-versa to suit a change in hardware or software configuration. In the CAP system the general structure of system modules is very simple as shown below:

```
initialisation;
DO # for every call#
     CASE get arguments; entry code
     IN
          service 1,
          service 2,
          ...
          service n
     ESAC;
     return results
OD
```

If the service is provided by a protected procedure, the arguments are passed in the course of the CALL operation and the answer is delivered by RETURN; whereas, if a message interface is used, the

module will execute WAIT and hold up until a message despatched by SEND in another process arrives, so that the arguments in it can be picked up by RECEIVE and processed before the results are returned by using REPLY.

CAP disguises the implementation of modules in the operating system from users by concealing them in gate-keeping protected procedures that check arguments and then communicate with system modules either by domain calls or messages, depending on the type of the module. The use of a gate-keeper reduces the efficiency of the concealed module by adding to the overheads of transactions with it, but on the other hand, if a system module is to be reconfigured, it is only necessary to edit and recompile the gate-keeper and users do not have to alter their programs.

If there was a single set of communication primitives that could be used for both varieties of modules, there would be no need to recompile anything at all, instead it would be sufficient just to switch the type of the module appropriately between 'process' and 'domain'. There would be an increase in efficiency as gate-keeper domains could be disposed of, and in addition, users would only have to know about a single communication mechanism rather than two.

In a simulator for investigating the effects of hardware and software configuration on system performance, Stroustrup [77] supports three types of modules: processes, procedures and monitors with two communication primitives, ACTIVATE and GET ARGUMENTS. ACTIVATE takes two arguments, the identity of a module to run and the name of an argument block which is used for passing arguments and results. The operand of GET ARGUMENTS is a notional communication channel which can take the two values 'request' and 'reply'. These are a set of minimal facilities that can be expanded to allow for more ambitious communication protocols. As an example, the following procedural call can be implemented

```
results := CALL (module, arguments)
```

as the sequence

```
ACTIVATE (module, arguments);
result := GET ARGUMENTS (reply)
```

If the module woken up by ACTIVATE is of the type 'process' or

'monitor' as opposed to 'procedure', it can be run in parallel to the calling module and GET ARGUMENTS functions as a synchronisation primitive to ensure the desired ordering of events.

It is clear that in this unified scheme a module can be written without knowledge of how it, or the modules it calls, are configured, and changing the type of a module does not require its recompilation which provides a potent degree of flexibility. However, despite the utility of the mechanism for experimenting with the effects of reconfiguring systems, it is rather too fundamental to be included in a practical implementation and attention must be directed to see how similar freedom can be included into a mechanism that is reasonable for incorporation into an operating system kernel.

Earlier it was stated that it is not reasonable to implement protection domains as monitors and thus it is necessary to turn to message-based communication systems. Watson [78], in his alternative protection system for CAP, has a non-hierarchical module structure which allows a module to be either a domain or a process. The unit of communication is a fixed size argument block issued from a central resident table. The ENTER instruction takes a capability for a module and an argument block as its operands and attaches the argument block to a queue of incoming messages for the called module. This module will be marked as either a process or a domain: in the first case, the calling module is allowed to continue execution after the ENTER instruction, while in the second case, the calling module is held up until its argument block is processed by the called module. Upon receipt of an argument block, a module is activated and the head message is taken off its incoming queue and made available so that arguments can be extracted from it. When an activation has been processed, a module can execute the RETURN instruction which will return the current argument block to to a queue of returned messages in its originator where it can be picked up by the RESULTS order. A module is deactivated if it tries to execute RESULTS when its returned messages queue is empty and will be awoken when a results message arrives. After the use of RETURN, the incoming message

queue of a module is inspected and if there is more work to do, the module will stay active, otherwise it will be held up until a new message turns up.

The microprogram implementing Watson's scheme carries out simple scheduling operations between modules as they execute the various message primitives, and it will select which processes to run on the basis of priorities held in process bases after process-type calls. More complex scheduling decisions are left to a software coordinator.

Argument blocks are concealed from users; they are the private property of the microprogram. The capabilities they contain become available as a current capability table after use of the RESULTS operation, and the data arguments are loaded into registers. The remaining information in an argument block such as return links and status bits is private to the microprogram.

Watson's scheme achieves the objective of unifying inter-process and inter-domain communication but it suffers from a number of drawbacks. Firstly, a module is only allowed to be in receipt of a single message at a time so that it is not possible for a module to multiplex calls as might be required by a disc driver that schedules disc accesses to minimise head movements.

Modules communicate directly with other modules and there is no notion of a message channel which would permit a utility to be served by several parallel modules, or for message paths to be dynamically switched between processes. Furthermore, since messages are routed to the same queue, it is not possible for a module to associate priorities to different sources of calls; for example, the CAP real store manager has a high priority channel for virtual memory fault handling and a low priority channel for user services, such as demands to modify the length of a segment.

A fundamental difficulty is created by having a single resident table of argument blocks because of the possibility that the limited stock of blocks may be overdrawn with disastrous consequences for the operating system.

Despite these objections, Watson's scheme has the advantage of

efficiency and simplicity because of its microprogrammed implementation and it demonstrates that it is possible to provide a basic set of primitives that encompass many of the properties of the fundamental scheme described by Stroustrup. In terms of microinstructions, the part of Watson's microprogram concerned with inter-module communication consumes a similar amount of space to the protected procedure and hierarchical process call facilities of the original CAP system. The cost of an ENTER in Watson's scheme is much less than a message transaction in the CAP operating system and is comparable to the cost of an ENTER in the earlier CAP system in terms of machine cycles taken.

CHAPTER EIGHT.

THE CAP COMPUTER.

8.1. The Hardware.

The hardware of the CAP computer was designed and built in the Computer Laboratory at Cambridge. For several years prior to commissioning the machine, there had been a project to design and implement a capability-based memory protection system and by 1973 the project had arrived at an architecture [Walker 73] that was considered worthy of turning into a machine so that the design could be evaluated in terms of real-life computing. To facilitate experimentation and possible changes in design, it was decided to to make the machine microprogrammable and to equip it with a substantial microprogram memory. Many of the features of the machine reflect the original architecture, though, fortunately for the work described in this thesis, the hardware is sufficiently general to permit investigation of alternative protection systems.

The configuration of the CAP machine and its related hardware is shown in figure 8.1-1. The two intimately connected peripherals are under direct microprogram control; the tape reader is used to bootstrap new microprograms from paper tape into microstore and the teletype is used purely for fault reporting and diagnostic purposes. All other peripherals are connected to a CTL Modular One Computer which acts as a front-end for CAP and is connected to it by a fast link. Either machine may send interrupts along the link and CAP has the ability to transfer data in and out of the Modular One's local memory. There is a permanently resident executive and link program in the Modular One which provides CAP with access to its peripherals and it is left to the CAP microprogram to map this interface onto the I/O architecture that is to be presented at the user level. The Modular One can function independently to carry out peripheral tests and its own housekeeping; similarly, CAP is free to run in the absence of the Modular One although it may only use the intimate devices for I/O.

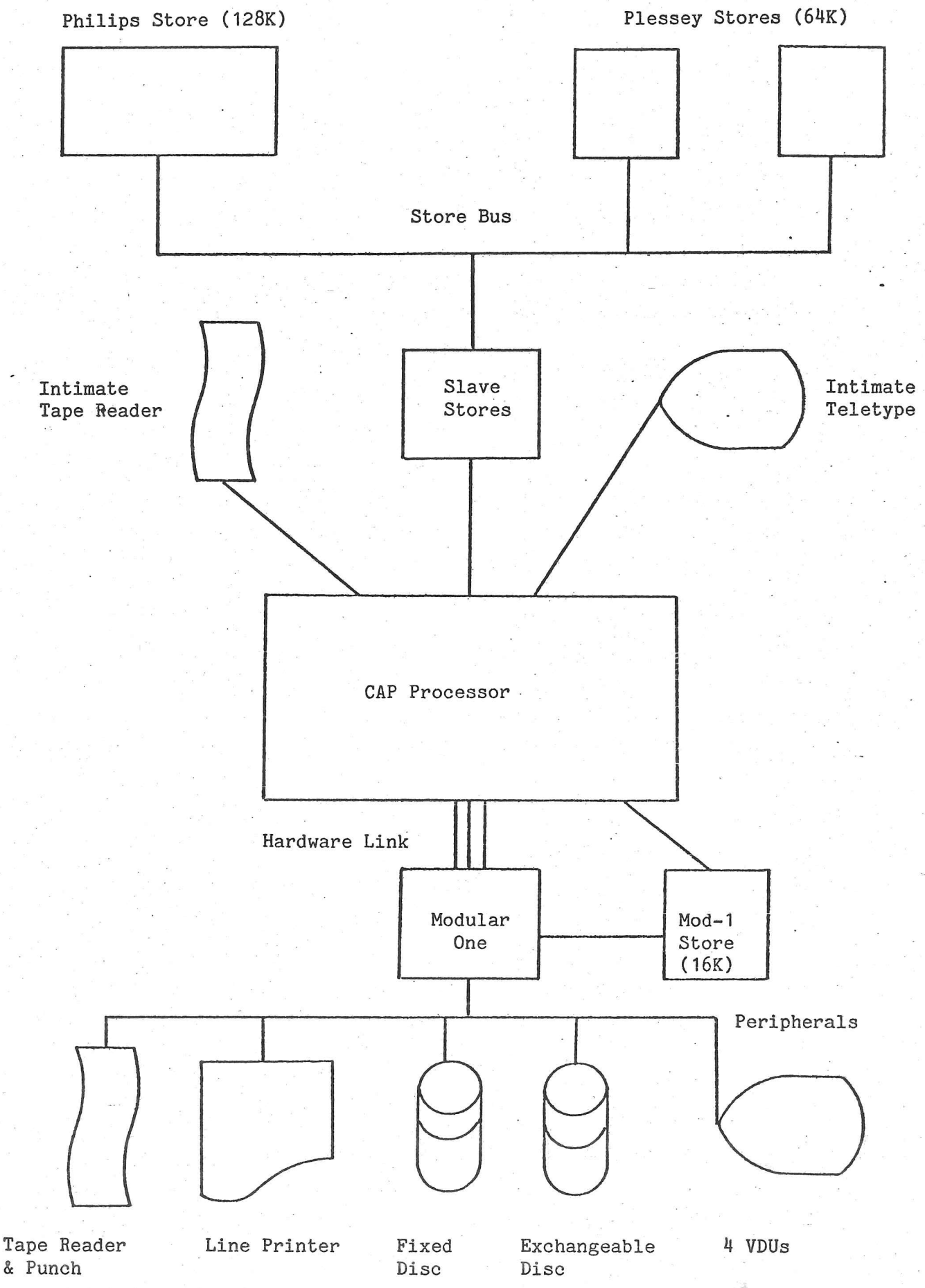


Figure 8.1-1 CAP Hardware Configuration

The main memory of CAP is provided by two interleaved 32K Plessey two microsecond, thirty-two bit core stores and 512K bytes of Philips core which also has an cycle time of two microseconds. The CAP has a thirty-two bit word length and the time taken to access and merge four bytes serially from the Philips store to make a word is ten microseconds. The rather dismal speed of the stores is compensated for by three slave (or cache) memories that have been observed to be very effective in operation [Cook 78].

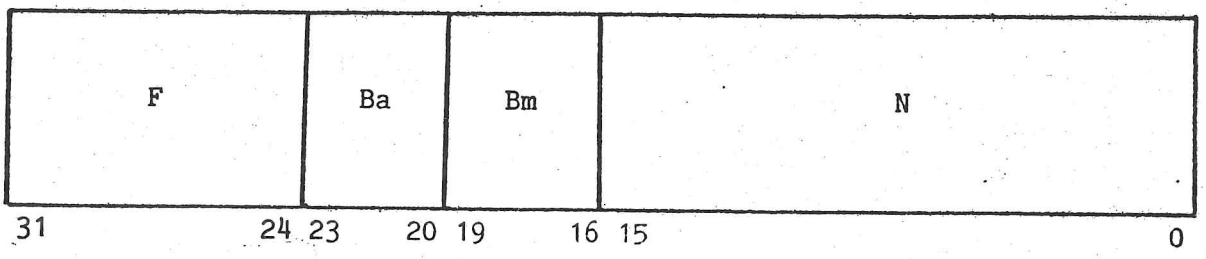
Within the CAP processor there is an autonomous floating-point arithmetic unit which has a sixty-four bit mantissa and an eight bit exponent working register. The unit has its own internal microprogram and processor to carry out addition, subtraction, multiplication, division and type conversion operations for fixed-point and floating-point numbers. The CAP microprogram can transmit arguments and pick up results from the unit.

The CAP supports a fixed format for ordinary instructions and has hardware to assist in function decoding. The instruction layouts are shown in figure 8.1-2. F is an eight bit function code, Ba, Bm and Bn are all four bit fields that select one of sixteen general registers (B0 to B15). Register B0 always reads zero and B15 is the program counter. N is a sixteen bit offset. In Type I instructions, Ba is an operand register. The contents of the register selected by the Bm field and the value of the N field (sign extended to thirty-two bits) are added to generate either a thirty-two bit address or literal data depending on the specification of the particular instruction. Type II orders are used to present three operands held in the registers nominated by Ba, Bm and Bn.

8.2. The Microprogrammer's Machine.

The structure of the microprogram processor is shown in figure 8.2-1. The microprogram memory holds 4K sixteen bit words, the top sixty-four words of which contain a hard-wired bootstrap routine, while the remainder may be dynamically loaded with microcode and data. The V-store provides the microprogram with access to the registers and control signals of other parts of the processor, such as the store logic and the floating point unit.

TYPE-I



TYPE-II

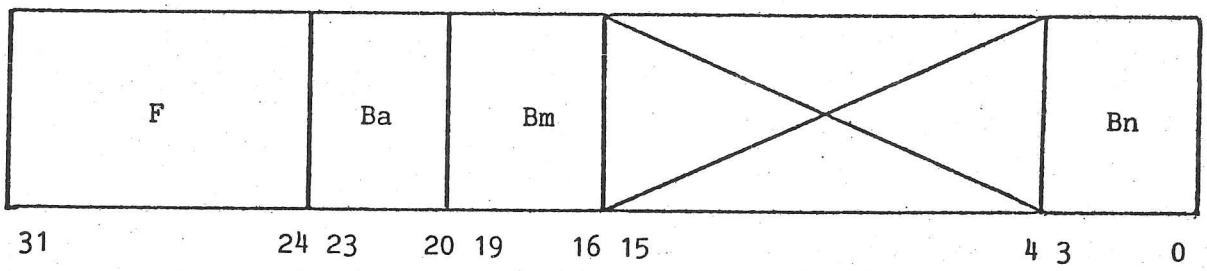


Figure 8.1-2 CAP Instruction Formats

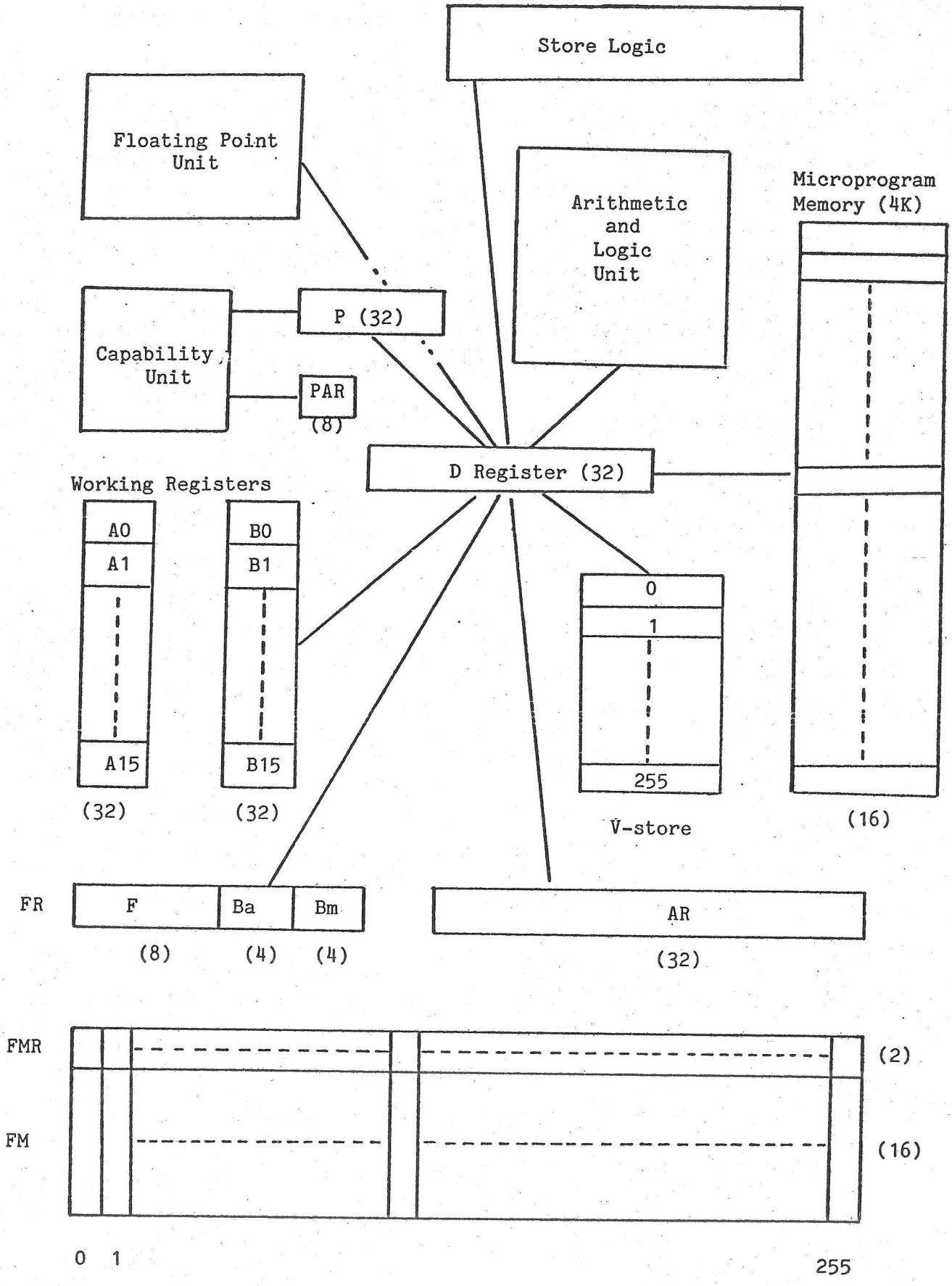


Figure 8.2-1 CAP Microprogrammer's Machine

It appears to the programmer as a bank of 256 registers. Some V-stores representing control signals have the property that the microprogram processor is held up until the function associated with the V-store is carried out. For example, writing to V17 prints the character in bottom eight bits of the D register on the intimate teletype and reading V1 advances the intimate tape reader so that a character can be latched into the D register. Other V-stores trigger events that are to take place asynchronously with the execution of microprogram instructions.

The sixteen general B registers available at the user level are also addressable from the microprogram and in addition there are a further sixteen registers (A0-A15) that are private to the microprogram. The D register is the central data highway of the machine; the Arithmetic and Logic Unit (ALU) deposits the result of a computation in D before routing it elsewhere. There is an interface between the ALU and the floating point unit and data sent between the units passes along this route. The microprogram controls the operation of the floating point unit by depositing data into the accumulator and, to a lesser extent, through the V-store.

The ALU is driven by a 50ns clock. A microinstruction takes between three and seven clock beats to complete, provided that there are no waits for external events. The microinstruction set resembles the order code of a simple old-fashioned machine; microinstructions are short (sixteen bits) and heavily encoded. This style of microprogram instruction set is often referred to as being 'vertical' in contrast to a 'horizontal' instruction set, where instructions are much longer and each bit of an instruction controls an individual function. The CAP microprogram instruction set is better illustrated by an example rather than by enumeration of the complete set of micro orders. The notation used is that of the standard microprogram assembler and the fragment of code is the part of the microprogram that decodes instructions, known as stage one.

```
(1)     B15+1->I.FETCH      // start fetch, increment B15
(2)     STORE->FR,AR        // instruction to FR,AR
(3)     BM+AR->P.FETCH      // modification - start fetch
(4)     STORE->D,A0         // read data (only for R or RW FMR)
(5)                         // instruction from FM intervenes
(6)     AD->STORE:RESTART   // return from FM for W type
(7)     AD->STORE:RESTART   // return from FM for RW type
```

The first instruction causes register B15 to be incremented after sending its contents to P, the store address register, then the store access register, PAR, is set to be execute access and the store logic is started. The next instruction (2) completes the store cycle and routes the user instruction fetched from store, via the D register, to registers FR and AR. (AR is set to be the least significant sixteen bits of the instruction, sign extended to thirty-two bits). Instruction (3) carries out the standard address modification: the contents of the B register selected by the Bm field of register FR are added to AR and sent to register P via D. The function code field of FR is used to index a bank of 256 registers (FMR) that hold access requests which are routed to PAR during this microinstruction. The access requests held in PAR may be 'read', 'write', 'read and write', or 'none', depending on whether or not data is fetched from or updated in main memory by the user instruction. The none code indicates that the modified address is being used literally and no store access is required. Instruction (4) is only obeyed if the FMR value is 'read' or 'read and write' and causes data to be loaded from store into registers D and A0. The next instruction (5) is rather special; it is held in the function memory (FM), a 256 word microprogram memory, which is indexed by the function field of FR. The order executed from FM will either complete the user instruction or else it will jump to the microstore address of further micro orders for more complex user instructions. For example, the order BBPS (B register incremented by contents of store) can be completed by:

```
BA+AD->B:RESTART    // ba:=ba+word from store
```

To implement JNEQ (jump if B register=0) a jump is placed in the function memory:

```
JMP JN.EQ           // b15: = n,if ba=0
```

which transfers control to the following orders:

```
JN.EQ, BA OR NIL->D:CSKIP // skip next micro instruction if Ba=0
       :RESTART           // start next user instruction
       BM+AR->D           // reconstruct literal address
       B15=D:RESTART      // set program counter to jump address
```

The RESTART option indicates that the instruction has been

finished off successfully and stage one decoding may begin for the next user instruction. If the intervening FM instruction in stage one does not jump out or restart either instruction (6) or (7) is executed depending on whether the currently selected FMR register has the value 'write' or 'read and write'. These final instructions enable orders that update store to be completed. For example, the order SSPB (add B register to store) has in its FM slot the instruction:

```
BA+AD->D            // D:=store + Ba
```

which computes the new value in register D and returns to instruction (6) to write the result to the store location whose address was computed by instruction (3).

There are several other operations and instruction types apart from those illustrated above, which include various shifts, byte masking, reading and writing microstore, accessing the V-store, logical operations, subroutine and unconditional jumps.

The essential difference between microprogramming and assembly code programming is that in microcode, it is left to the programmer to ensure that he gets hardware interlocks correct. For instance, in the CAP it is possible to halt the processor by reading the store data lines if the store address lines have not been set previously.

There are a number of faults primarily associated with addressing violations and arithmetic overflow in the floating point unit, which are trapped by the hardware and if one of these exceptions occurs, microprogram control is immediately switched to a specific location of microstore. It is left to the microprogrammer to provide code starting at that location to investigate the nature of the fault and to take appropriate action. External interrupts, for example those from the Modular One, are only noticed whenever a return is made to stage one. If an interrupt is signalled at stage one, control is diverted to an interrupt routine starting at a fixed address in microstore and the interrupt routine can determine the type of interrupt by reading registers in the V-store.

The microprogram instruction set has a number of weaknesses

that are a nuisance to the microprogrammer and waste precious instructions in the cramped microprogram store. By far the greatest difficulty is caused by the subroutine jump and return mechanism which has only two registers available for holding return links, because in general, two levels of subroutine are insufficient in a complex microprogram and experience suggests that four link registers would probably be better.

It is not possible to perform every operation on all registers, nor is it possible to route results back to all registers because many instructions are defined only to utilise a subset of the available registers. In general, the D register is the primary working register; the A registers are less useful in terms of the operations that can affect them and the ease with which they are accessed. As a result, the microprogrammer frequently has to waste instructions loading values into D to perform a calculation upon them and then copying the result to another register. This fault is directly attributable to the compactness of the microinstruction format which does not have enough space to encode all of the possible register combinations. A further consequence of this arises in connection with an option which permits the next instruction to be skipped over if register D has a certain value after the current order. Unfortunately, each microinstruction supporting the option only generates a single condition and it is frequently necessary to write a further instruction to test a condition different from that available after a computation. A better mechanism would be to allow conditional skipping on a condition that may be specified within an instruction, for example, from the set {=0,>0,<0}.

The difference between the thirty-two bit word length of the machine and the sixteen bit word length of the microprogram memory causes difficulty if copies of thirty-two bit words are to be kept in microstore. LOAD DOUBLE WORD and STORE DOUBLE WORD microinstructions would be very useful. If there were more registers there would be less need to use microstore as a repository for data.

There are, however, several benefits to be gained by having a simple and compact microprogram instruction set. It is much

easier to write code than would be the case in a highly parallel horizontally microprogrammed machine and this is helpful from the point of view of debugging microcode, making modifications to it and also for program verification. The last point is crucial: the state of the art of program verification is such that, for microcode at least, it is necessary to rely on visual checking alone. Therefore it is an advantage to have a microprogram which is easy to read and follow. In return, of course, there is not the same scope for carrying out several operations in parallel in a vertical microcode and efficiency will be lost because of unnecessary serialisation.

8.3. Accessing Main Memory.

In this section the hardware in the CAP system that is responsible for virtual address translation and memory protection will be outlined. The store logic operates in a number of different modes which may be selected by setting flip-flops in the V-store. In all cases an address is taken from the P register together with an access code from register PAR.

The simplest addressing mode is called absolute mode. The least significant twenty bits of P are treated as an absolute address and are passed directly to the store address lines. The access code in PAR is used to determine whether a reading or a writing store cycle is to be generated. After setting register P, the microprogram will read or write to the store data lines to complete the store transfer. The slave store mechanisms are interposed between the addressing logic and the physical memory so that, at the level of addressing, the slave is transparent to the microprogram.

The other addressing modes are used to carry out address translations and access checks and for this purpose there is a capability unit, the organisation of which is shown in figure 8.3-1. The unit divides into two parts: a bank of sixteen registers known as the tag memory (TGM) and sixty-four capability registers. The latter divide into six sub-registers known as the tag, base, limit, access, count and spare registers. The TGM and tag registers are concerned with address translation and will be

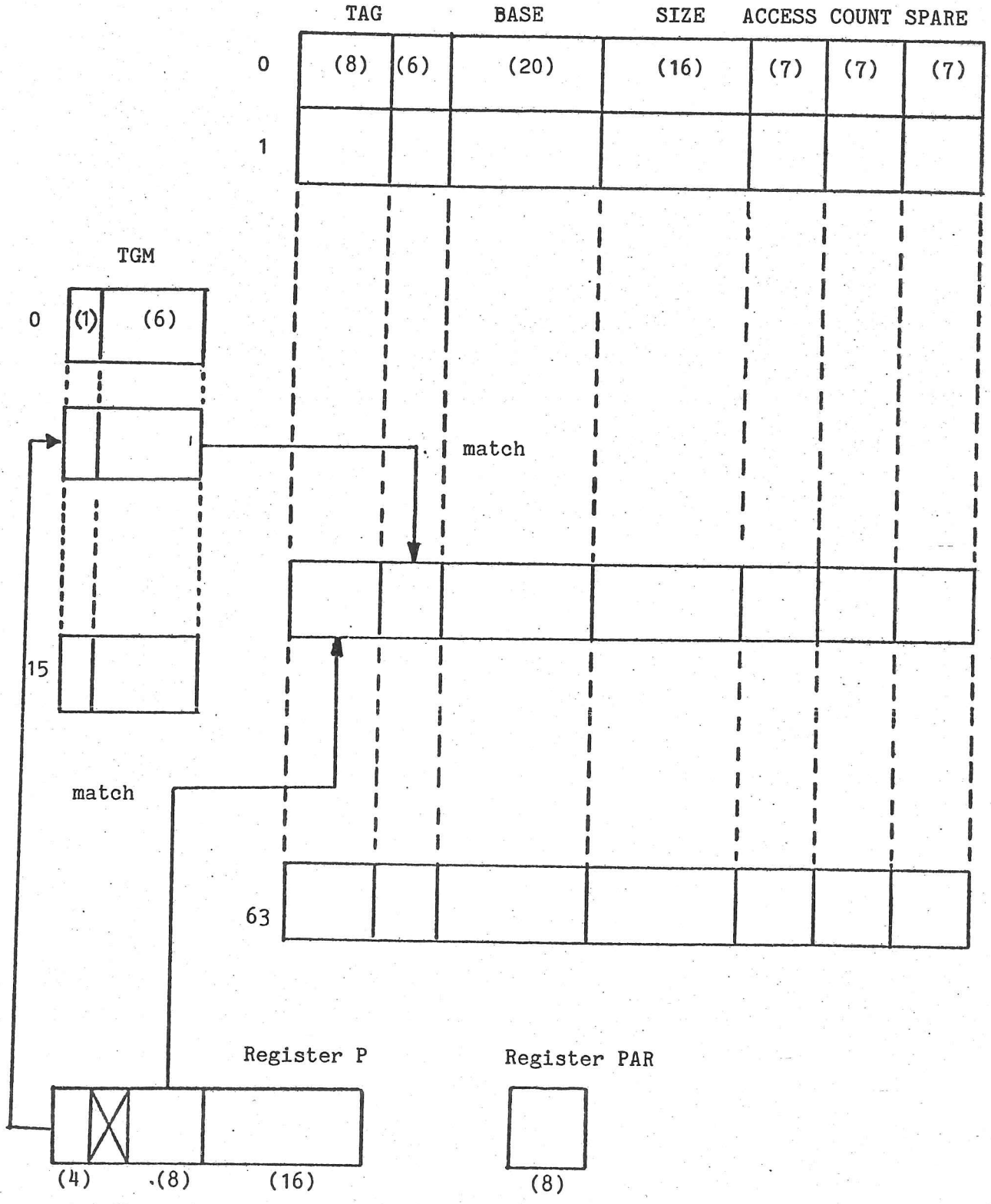


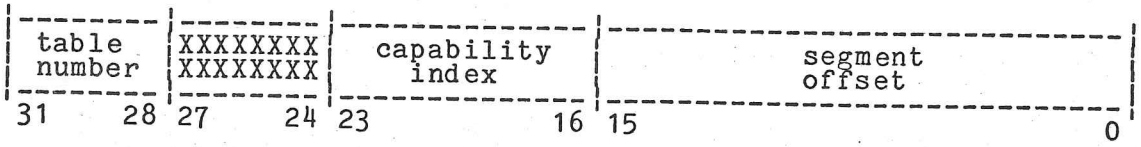
Figure 8.3-1 Capability Unit Organisation

described later. The count and spare registers take no part in addressing or access control and are used by the microprogrammer for housekeeping purposes that will be the subject of a later chapter. The remaining registers function similarly to segment descriptor registers that are found in other machines.

In last mode, a particular capability register may be selected by writing to a V-store location. When an address is written to P, only the least significant sixteen bits are used. They are compared to the limit field of the selected capability register and if the address exceeds the limit field, an error is trapped and control is diverted to location eighteen in the microstore. Similarly, a check is made to see that all of the bits set in PAR (the access request) are also set in the access field of the capability register. An access violation also causes a trap to location eighteen. Provided that the access and limit checks are successful, the sixteen bit address in P is added to the twenty bit absolute base address in the base field of the selected capability register to calculate an absolute address which is sent to the store address lines. Last mode is used extensively to address data structures whose entries in the unit have been loaded by one of the two remaining addressing modes so that address relocation can be carried out automatically and also to prevent the microprogram from carrying out an illegal access to the data.

Direct mode enables capability registers to be selected by a field in addresses. As with last mode, the least significant sixteen bits of register P form a segment offset which, together with the access request in PAR, are compared with the contents of the selected capability register. The selection is determined by the six bits preceding the segment offset in P. Using direct mode, it is possible to build a capability register system that resembles the Plessey System 250.

The final addressing mode, normal mode, is the most involved as it carries out virtual address translation in addition to access checking. A CAP virtual address is thirty-two bits long and laid out thus:



As before, a store cycle is started by writing an address to P and an access request to PAR. The four bit capability table number indexes the TGM to yield a six bit tag. An associative search is then made through the capability unit looking for a register which has a fourteen bit tag field matching the concatenation of the six bit tag from the TGM and the capability index part of the address. If a register is selected, the remainder of the address is interpreted similarly to the last and direct modes of addressing. Otherwise a trap to microstore location seventeen is generated to indicate that a match was not made.

The capability registers are divided into banks of four registers. The searching algorithm proceeds by selecting a bank and then performs an associative match on the registers within the bank. If there is no match, the search moves on to the next bank cyclically, until all of the registers in the unit have been looked at. Thus there is a high premium on ensuring that the capabilities are loaded near the point at which the unit will start searching.

The intricate structure of the unit allows capabilities to remain within the unit even when they are no longer addressable because of a protection domain or process change. This is useful if domains are small and are called frequently, as it avoids the overhead of flushing out and reloading the unit on every domain entry and exit. The details of the organisation of capabilities within the unit and the way that the TGM and tag registers are used is discussed in more detail in Chapter Twelve. For the time being, it is sufficient to say that for each capability segment in a protection domain, the TGM holds a key which is in the tag register of every capability loaded from the segment into the capability unit. If a protection domain change occurs, new keys are put into the TGM, so that previously accessible capabilities will not match until the original keys are restored after returning from the called domain.

It is up to the microprogram to handle faults and exceptions reported by the capability unit. Furthermore, it is also the responsibility of the microprogram to allocate slots for newly evaluated capabilities and to arrange that any operations carried out on capabilities in store are reflected by changes to the contents of the capability unit. The microprogram is able to interrogate and modify the contents and state of the unit by depositing information in the accumulator and to a lesser degree by accessing parts of the V-store.

It is the use of the capability unit as a cache for store capabilities that contributes most to the effectiveness of the CAP memory protection system: once a capability has been evaluated, address translation and access checks are carried out with only a minimal overhead and the unit is sufficiently large that capabilities are re-evaluated infrequently. The cost of a large cache has been traded against the time that would be wasted loading and unloading the unit if it held fewer capabilities and suffered from 'capability thrashing'.

8.4. Microprogramming Aids.

There is a standard microprogram assembler for the CAP machine. The assembler is not very rich in facilities and has a number of idiosyncracies. Despite this, the CAP kernel was written for this assembler so that code for emulating user instructions, performing I/O and so on, could be borrowed from the existing microprogram. The assembler was originally written for an IBM System/370 but at the time that the kernel was being developed, the CAP operating system became available and the assembler was moved across to it. At this stage, I modified the assembler so that it put microprograms onto disc in a format that can be loaded into CAP microstore by a simple bootstrap program. This step greatly increased the rate at which new versions of the kernel could be assembled and tested, as in the past it was necessary to conduct an assembly on the IBM machine and then to punch out binaries on paper tape for loading via the CAP intimate tape reader.

The kernel was debugged using just the raw hardware. CAP is well equipped with LED displays of registers and control signals, and there is a well-endowed control panel with facilities for obeying single shot instructions, setting a break point and obeying instructions set up on the hand keys. There is also a postmortem program which will tabulate the values of all of the microprogram registers (including the V-store) on the line printer.

Working in this way it is surprisingly easy to test large tracts of microprogram in a short time. The main difficulty is in persuading other users of the machine to desist so that hands-on access could be gained. Fortunately, during the period in which the kernel was written, this was not too great a problem.

CHAPTER NINE.

A KERNEL FOR THE CAP COMPUTER.

9.1. Preliminaries and Notation.

In this and the following chapters the design of a kernel for the CAP computer and its implementation will be documented. The discussion will distinguish between the earlier memory protection system and the kernel by referring to them as CAP-I and CAP-III respectively. The rationale for the decisions leading to the architecture about to be described can be found in the first section of the thesis.

The instruction format, addressing conventions and word length of the new system is identical to that of CAP-I for two reasons: firstly, so that utility programs like compilers can run unchanged on either system and secondly because CAP-III uses the same hard wired logic for instruction decoding and virtual address translation as CAP-I for reasons of efficiency. The microprogram for the basic instructions and organising I/O across the link to the Modular One computer within CAP-III is more or less an exact copy of its counterpart in CAP-I. The remainder of the microprogram is concerned with protection, which is very different in the two systems, although many of the kernel instructions have direct analogues in the memory protection system. The implementation of this part of the kernel was carried out from scratch through a number of iterations to the current specification.

The standard microprogrammed instruction set for the CAP machine is both conventional and extensive; it includes integer and logical operations between B-registers and store, conditional jumps, subroutine entry and exit, byte addressing, byte packing and unpacking, multiple register to store dump and reload, fixed and floating point arithmetic, block move and clear, Algol 68 CASE, modification of next instruction, integer and floating point conversions, test-and-count and exchange register with store functions. The full set is documented in Herbert [78]. In the

following chapters, a standard notation is used for describing instructions thus:

dxx digit xx of a binary number (d0 is the least significant digit)

F function code identified by d31-24 of the current order.

Ba B register identified by d23-20 of the current order.

ba The contents of Ba.

Bm B register identified by d19-16 of the current order.

bm The contents of Bm.

Bn B register identified by d3-0 of the current order.

bn the contents of Bn.

N The signed integer formed by d15-0 of the current order.

n The value bm+N.

[x] The contents of the store location whose virtual address is x.

s The value [n].

Used to introduce a hexadecimal number.

It is always assumed in the description of an instruction that any reference to reading store implies that a protection check is made every time store is accessed. Thus a protection violation is signalled if insufficient access rights are held or if an address is invalid or beyond the end of a segment or capability table. The kernel indicates these exceptions, along with other faults, as an interrupt to the software which contains a code indicating the nature of the fault. The program counter (register B15) of the current process at the time of the fault is set back so that if execution in the process is resumed, the failing order will be retried. The kernel always arranges that a process and all of the protection apparatus is left in a consistent state after a fault so that the integrity of the system will remain guaranteed. Furthermore, the kernel places no reliance on any data structures kept in main memory so that if a program, either by accident or malice, interferes with an intimate part of the protection apparatus, the kernel cannot be induced to give away privileges or behave in an unreasonable manner.

9.2. Naming.

It was decided to employ a global naming system for the kernel because global names are conceptually easier to understand and they are more suited to extended object manipulation. Another aim was to gain experience in the use of global names and to compare global naming schemes with the nested naming schemes. A forever-unique global naming scheme was obviously not suitable for microprogram implementation because of the large amount of code required to administer an object table that is kept partially on disc. Instead, a scheme that uses short names and a modestly sized resident table for just the set of currently-active objects is employed. As a consequence, active capabilities cannot be kept in the filing system and it is necessary to have operating system support for translating capabilities from the filing system into the run-time capabilities manipulated by the kernel. Experience with the CAP-I operating system System Internal Name mechanism (Chapter Three) suggests that the necessary translations can be performed at a cost that is at worst comparable to, but probably less than, the expense of organising passive and active object tables in a forever-unique scheme.

In CAP-III the descriptions of all currently active objects are found in a resident table consisting of four word slots known as the map. The maximum size of the map is 65535 words (i.e. 16383 slots) although in practice, its size is expected to be in the order of four to eight thousand slots. An attempt to access a slot that is out of the bounds of a map will result in a fault. In general, the kernel will only take names from capabilities or entries in the map.

An object is said to be 'active' if there is at least one capability for it in a capability segment. To help detect objects that are no longer referenced, the kernel keeps a reference count for each entry in the map, and it is incremented whenever the kernel sets up a capability or map slot pointing to it and is decremented whenever the kernel deletes such a pointer. As will be shown in Section 9.8, reference counts alone cannot detect all free slots in the map and it is necessary for the software to include a garbage collector which can function asynchronously

without holding up the remainder of the system.

The kernel maintains a pool of map slots that are currently free and takes a slot out of the pool whenever a new object is set up. The pool is organised as a list and information about it can be obtained by the FREEQ instruction:

FREEQ no arguments.

ba(d31-16) := head slot of free list.
ba(d15-0) := length of free list.

If the pool is empty when a new slot is required, a fault is generated and the failing instruction can be retried once some space has been recovered in the map.

The layout of a capability is shown diagrammatically in Figure 9.2-1. A capability is two words long and consists of four sixteen bit fields. The name field will nominate the map slot containing the description of the object protected by the capabilities. In fact only fourteen bits are required to address all of the slots in a map of maximum size and a sixteen bit field is only used for convenience in the kernel microprogram. The interpretation of the access code field depends upon the type of the object named in the capability, with the exception of bit fifteen, the revoke bit which is a generic code associated with revocation (Section 10.4). The use of the base and size refinement fields will be dealt with in Section 9.3.

There is one name that is treated specially; it has the value 65535 and implies that the capability is null, that is to say, the capability is not bound to an object. This capability is useful for overwriting capabilities that are no longer required to prevent them from being used any further. The kernel will signal a fault if any attempt is made to evaluate a null capability but it is perfectly permissable to move such capabilities around with the capability transfer instructions.

The format of a map slot is shown in figure 9.2-2. The type field (sixteen bits) identifies the class of objects to which the particular object belongs. Some types (segment, process, message channel, message and type-object) are recognised and supported by the kernel; other types are defined by software and are the

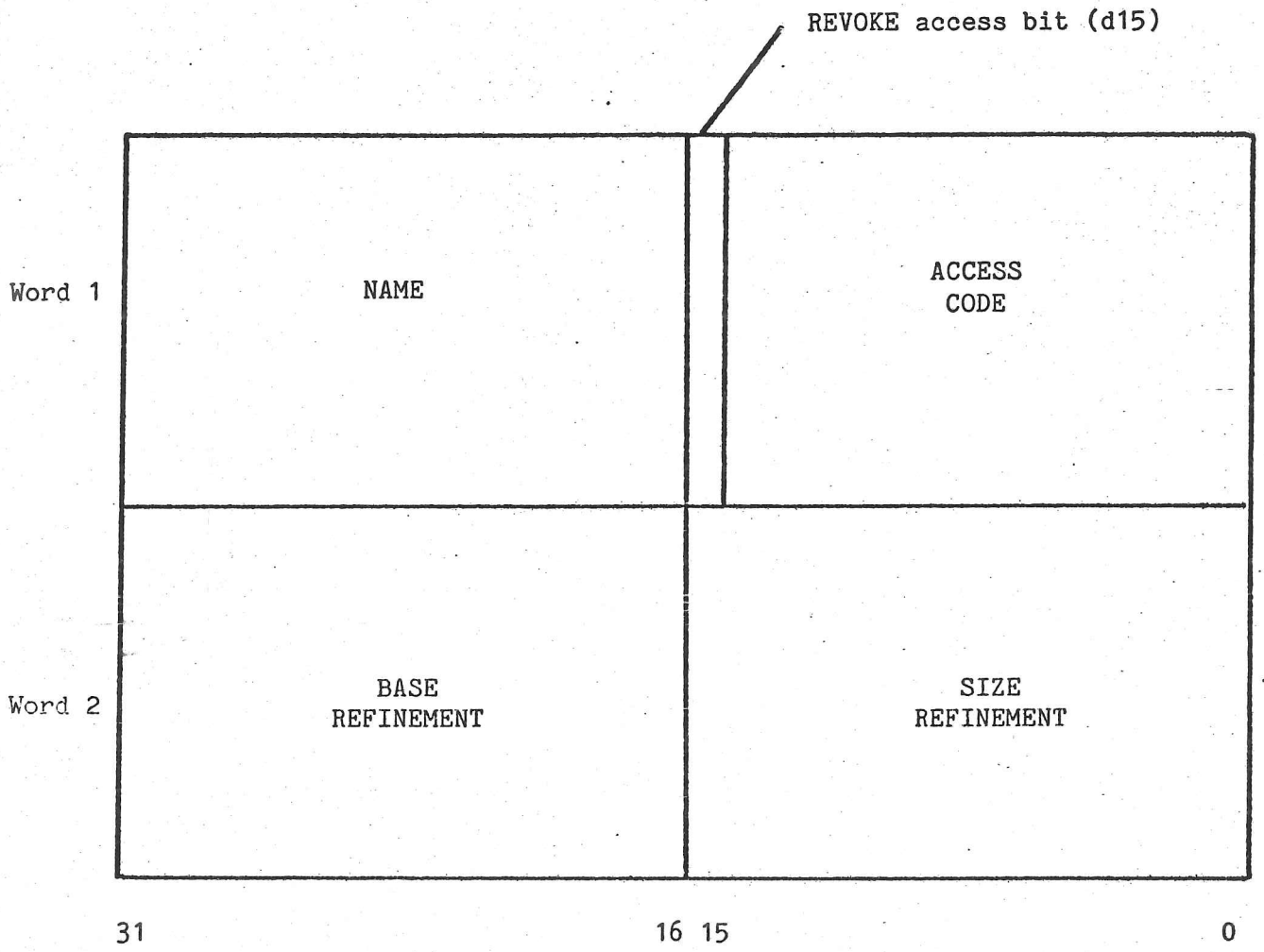


Figure 9.2-1 Capability Format

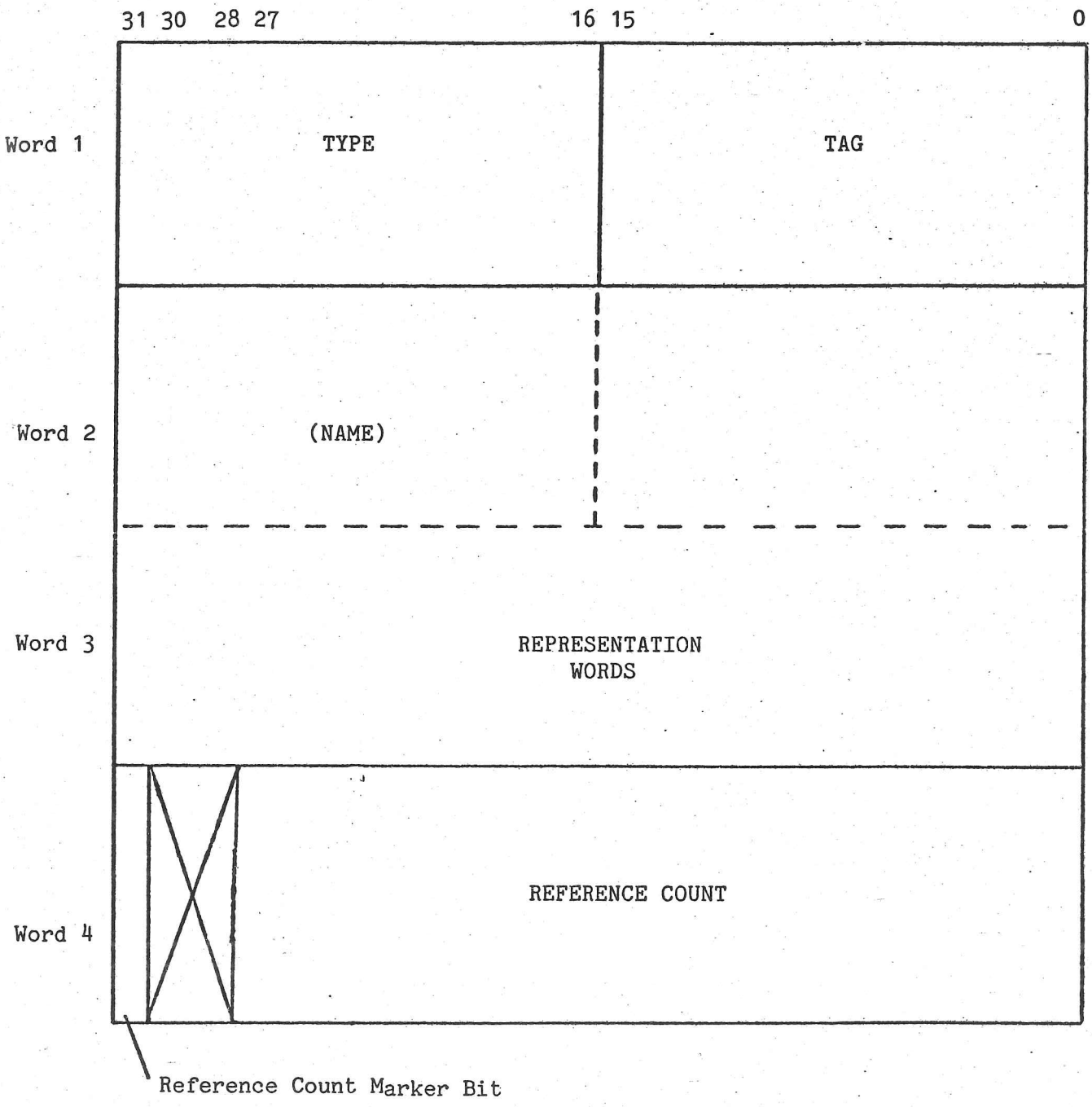


Figure 9.2-2 Map Slot Format

currency of the operating system.

The tag field (sixteen bits) is not interpreted by the kernel and can be used by software as a label for the slot and is initialised when it is set up. One potential use of these tags is so that the software for translating filing system names to map slot numbers can record a key in a map slot that can be used to locate information about the corresponding object in its translation tables.

The representation words in a map entry are used to describe the substance of an object and may either be a capability or simple binary data. In the latter case, the name field of the representation will have the value 65535, which imposes the minor restriction that it is not possible to have extended objects with null capabilities as their representation. This restriction is enforced by all of the kernel type-extension facilities.

The use of the twenty-eight bit reference count and the reference count marker bit is deferred until Section 9.8.

In the rest of this description it will be convenient to refer to the type of an object as an attribute of the capabilities for it although type codes are only held in map entries. Thus, the term 'segment capability' means a capability that names a map slot defining a segment-type object. It is also useful to let the term 'object' denote the map slot holding information about the representation of an object. When talking about 'objects', it will be clear from the context whether the actual object itself or the map entry that describes it is intended.

9.3. Segments.

The unit of memory protection is the segment which is a contiguous set of words in store of an arbitrary length up to a maximum of 65535 words. The access codes defined for segments are read- and write-capability, and read, write and execute data. Segment type map entries are basic objects, which is to say that they have a data type representation as shown in figure 9.3-1.

The twenty bit absolute base field defines the starting location of the segment in physical memory and therefore, the

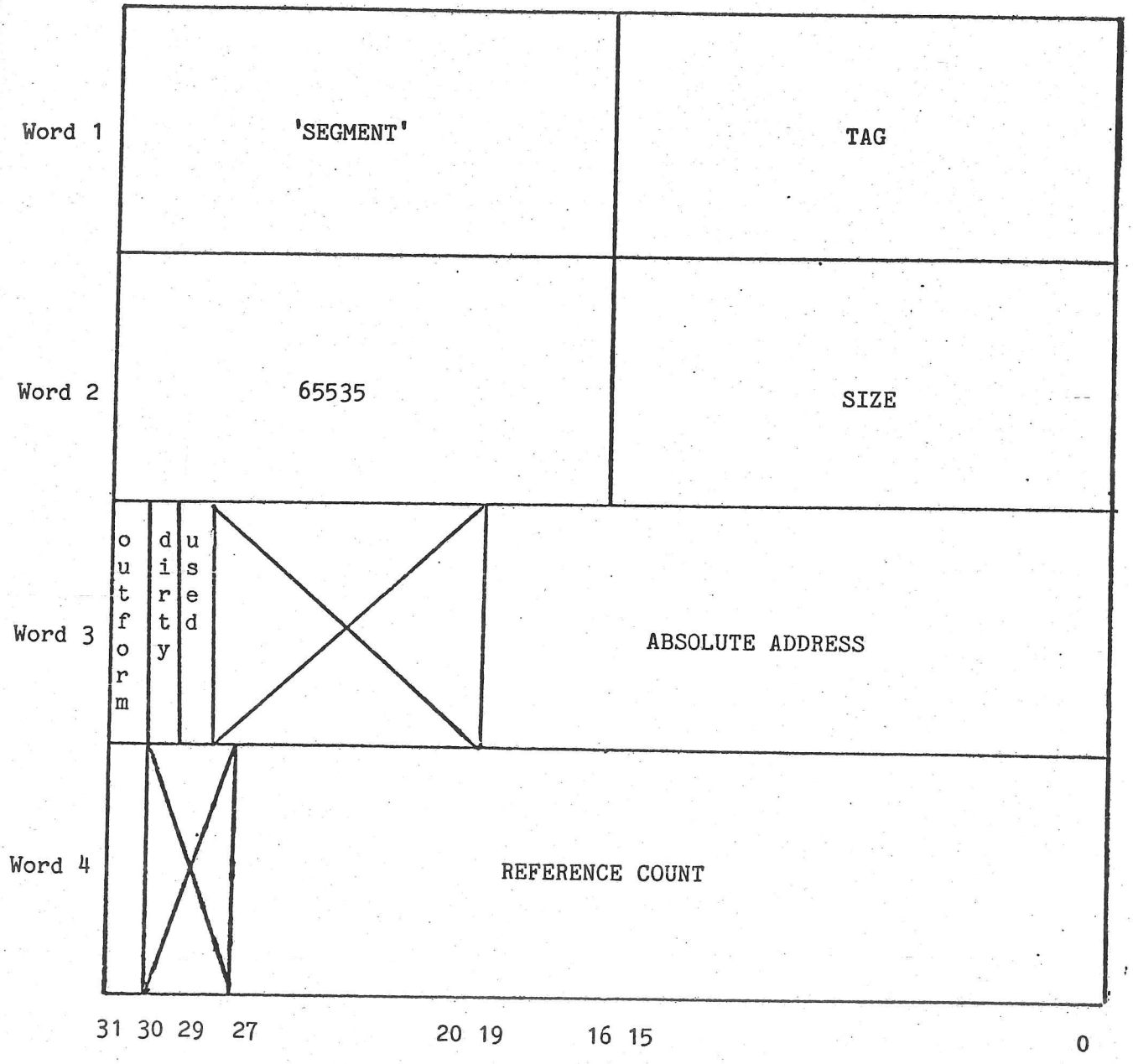


Figure 9.3-1 Segment Map Entry Format

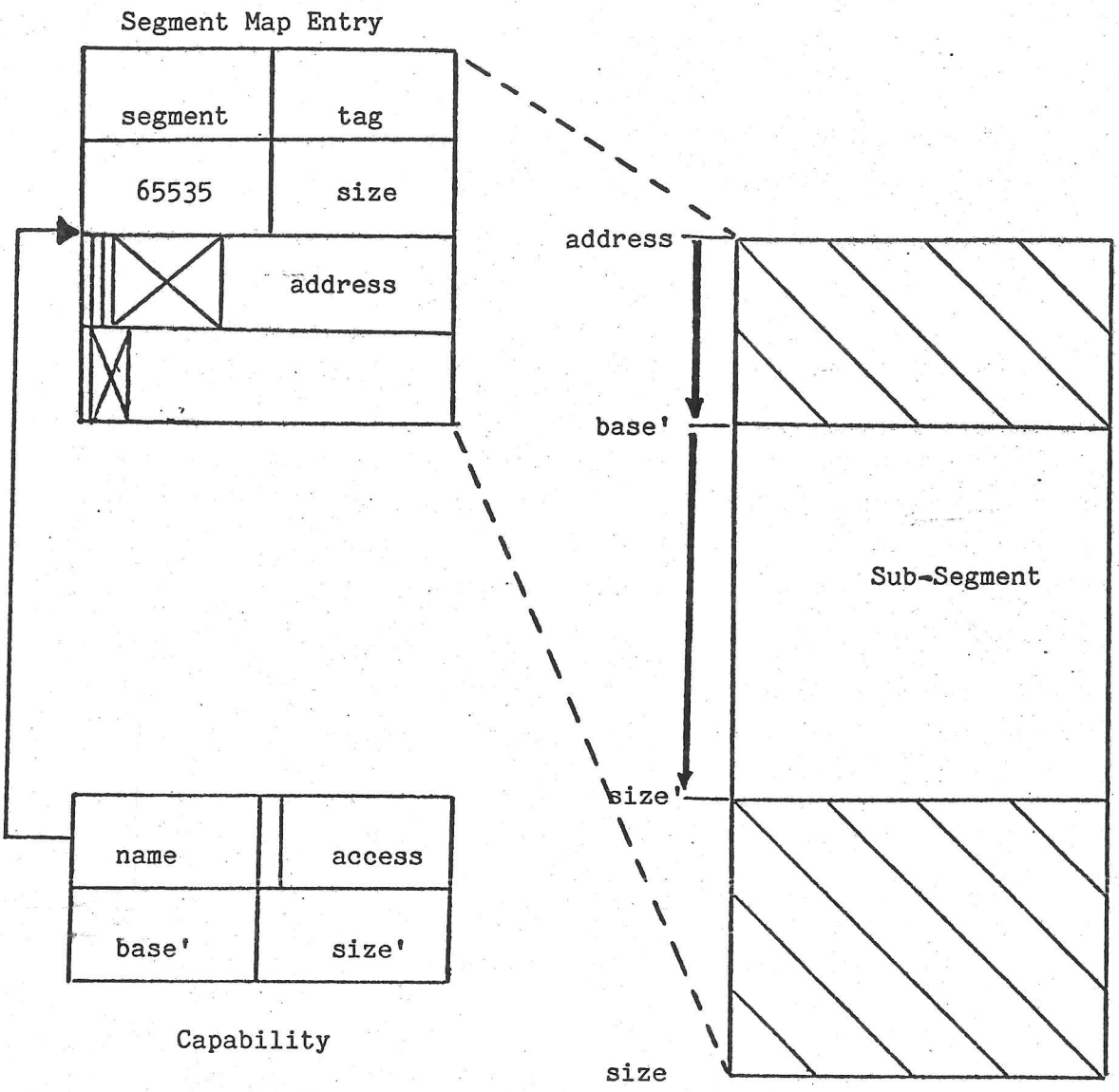
maximum memory address that can be accommodated is 1024K words, which is in fact a hardware limit imposed by the addressing logic.

The sixteen bit size field defines the length of the segment in words, although individual capabilities may select just a small portion of the whole. Capability segments are aligned so that capabilities occupy adjacent pairs of words starting from the base of the segment and most instructions reference capabilities by an index such that the first word of the capability will be found at the word in the segment whose offset is twice the index.

The 'outform' status bit can be used for organising virtual memory swapping. The kernel signals a fault if the evaluation of a segment capability yields a map entry in which this bit is set. When the bit is on, the operating system may utilise the other fields of the entry to hold information about disc addresses and so on; at all other times, the absolute base and size fields are interpreted normally.

The other two status bits, 'dirty' and 'used', record information about the types of access made to a segment. The 'used' bit is set to one if it is read as zero when the capability for a segment is evaluated. The 'dirty' bit is set if it is read as zero when the capability for a segment is evaluated in the course of a store demand that includes 'write data' or 'write capability' access. The software can reset these bits and monitor them from time to time to discover which segments are accessed frequently and which ones have been modified.

Sub-segmentation is performed by the base and size refinement fields of segment capabilities as illustrated in figure 9.3-2. The base refinement field must be less than or equal to the size of the absolute segment and the sub-segment begins at the absolute address formed by adding the absolute address of the segment to the base refinement of the capability. The size of the sub-segment is the smaller of the size refinement field of the capability and the remaining length of the absolute segment beyond the base of the sub-segment. The refinement mechanism provides a means for selecting small portions of larger structures and this is particularly useful in the manufacture of argument capabilities



Refinement calculation:

```
IF   base' > size
THEN error
ELSE address of sub-segment := address + base';
     size    of sub-segment := min (size', size - base')
```

Figure 9.3-2 Segment Refinement

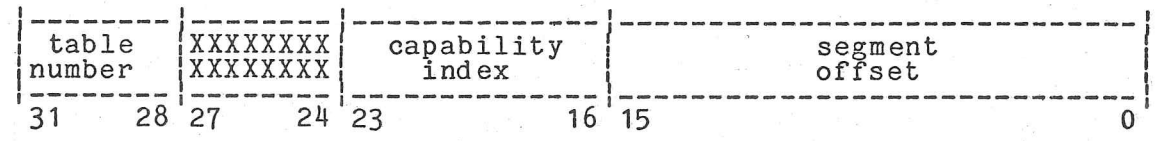
during inter-process communication.

In capabilities for objects other than segments, the refinement fields are not significant and have no effect on the process of capability evaluation.

9.4. Addressing.

The fact that capabilities and data are partitioned rules out the possibility of using Fabry's capability-based addressing scheme for CAP-III. Instead, the addressing architecture of CAP-I is adopted with the important restriction that there is only one domain per process. The CAP-III kernel has an efficient inter-process communication facility and the analogue of a CAP-I process consisting of several domains in CAP-III is a team of processes with shared capabilities for common objects. This means that the problems that can arise when CAP-I protected procedures pass addresses around or share multi-segment data structures do not cause concern, but the difficulties of addressing clashes remain for inter-process communication. However, it is much better practice to transfer capabilities rather than addresses, and the CAP-III message system is suited to this.

The root of a process's address space is its domain descriptor, which is a capability segment local to the process. The first sixteen slots in this segment define a set of capability tables which are capability segments. If one of the slots in the domain descriptor is a null capability, the corresponding table is deemed to be absent. Each table may contain up to 256 addressable capabilities and the complete set of all of the capabilities in the tables forms the process's domain of protection. The format of a thirty-two bit virtual address is shown below:



The table number selects one of the sixteen capability tables and the capability index then nominates a particular capability from within the table. These two fields of an address taken together are known as a capability specifier. The offset part of an

address only applies to segment capabilities and indexes an individual word in the segment defined by the capability selected by the capability specifier part of the address.

The virtual address translation mechanism will fault the evaluation of a capability if it is unable to read the domain descriptor or capability table with read-capability access, if any of the address fields index beyond the end of the appropriate segment, if the capability for a capability table in the domain descriptor is null, or if any of the capabilities for the various capability segments turn out to be for types of objects other than segments.

The structure of a CAP-III address space and an example of address translation is shown in figure 9.4-1. It should be noted that both capability segments and data segments have the same type ('segment') and thus the interpretation of a segment capability depends on the access code within the capability. It is perfectly permissible for the software to have a data-type capability and a capability-type capability for the same segment in order that some part of the operating system can create or modify capabilities within the segment. In particular, the software responsible for translating between permanent and active names will need this privilege in order to create additional instances of capabilities for already active objects. The kernel caches evaluated capabilities for kernel-defined objects in the hardware capability unit and if the software overwrites a capability using the contrivance described above, it is necessary to ensure that the kernel notices the modified value of a capability. For this purpose, there is the following instruction:

FLUSH n (d31-16) capability specifier.

Any entry for the capability specified by n(d31-16) in the hardware capability unit is disabled, to force the re-evaluation of the capability from the updated segment if it is used again.

This facility is only intended for use by operating systems software but it is perfectly in order for other programs to use it as the instruction causes no harm, except to force a spurious

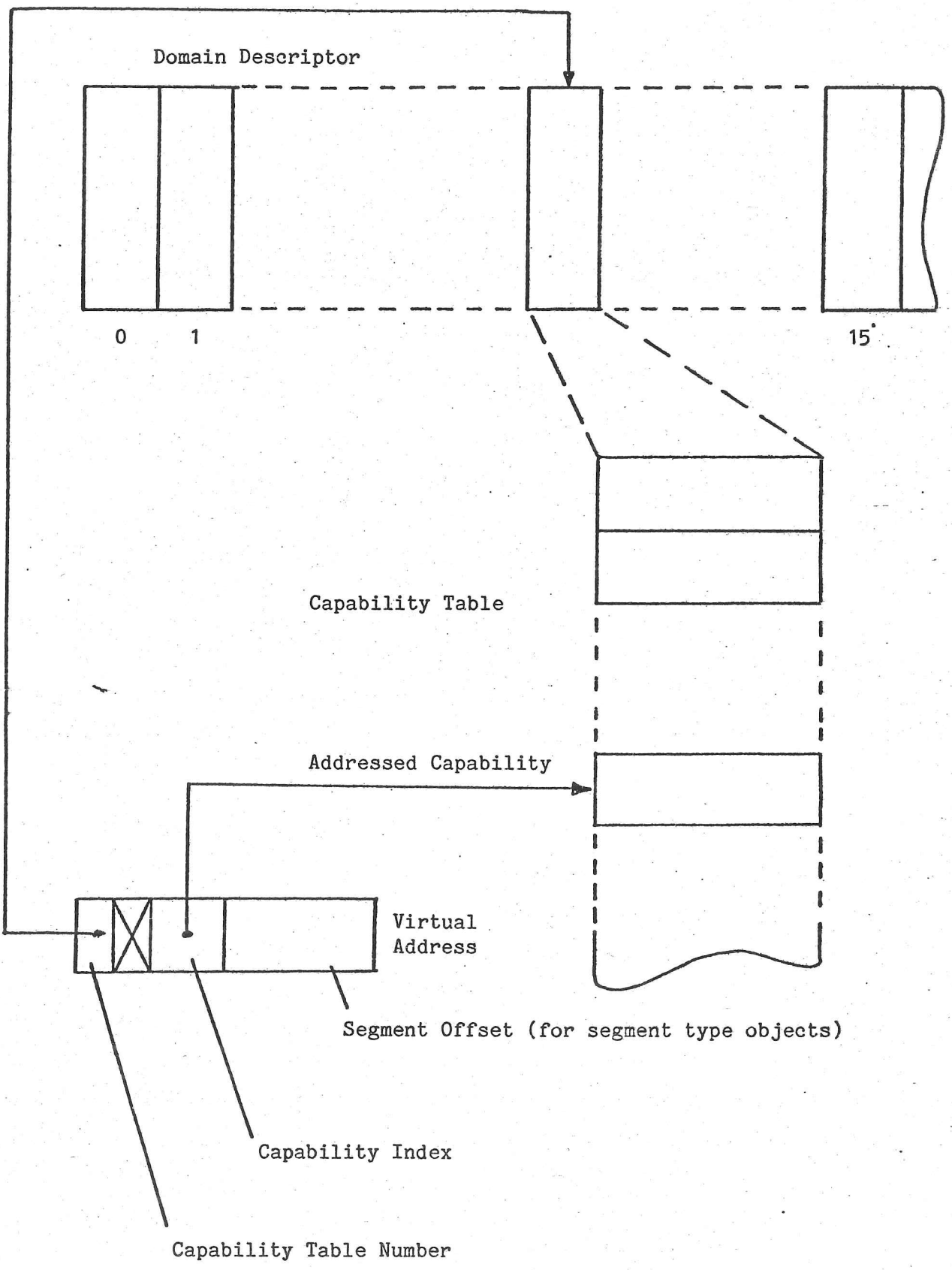


Figure 9.4-1 CAP-III Addressing Structure

re-evaluation of the nominated capability.

9.5. Information Orders.

There are three instructions for obtaining information about capabilities and objects:

```
OBJINF   n (d31-16)   capability specifier.

         ba(d31-16) := tag field of object n(d31-16).
         ba(d15-0)  := access code of capability n(d31-16).

SEGINF   n (d31-16)   segment capability specifier.

         ba(d31-16) := length in words of segment n.
         ba(d15-0)  := access code of segment capability n.

CSEGINF  n (d31-28)   capability table number.

         ba(d31-16) := length in words of capability table n.
         ba(d15-0)  := access code for capability table n.
```

In these and other orders, arguments are interpreted as addresses which select capabilities which in turn lead to the description of particular objects. Thus, 'object n' denotes the object defined by the capability at address n. If the object located in this manner is inappropriate to the function of the instruction, such as SEGINF on a non-segment capability, a fault is signalled.

9.6. Capability Transfer.

It is useful to be able to move capabilities between capability tables in the current process. The capability transfer suite of orders are concerned purely with capabilities and do not affect the objects protected by them. Transfers are carried out by making a copy of the source capability and then overwriting the previous contents of the destination capability slot. It must be possible to read the source and destination capability tables and to write to the destination table with capability-type access. The capability transfer orders are specified thus:

```
MOVECAP  ba(d31-16)   source capability specifier.
         n (d31-16)   destination capability specifier.
```

The capability is copied, without modification from source to destination. The previous capability at the destination capability slot is lost and any entry in the hardware capability unit for it is flushed out so that

the capability will be re-evaluated to pick up its new value.

REFINE
ba(d31-16) source capability specifier.
ba(d15-0) access mask.
n (d31-16) destination capability specifier.

In addition for segment source capabilities:

b(a+1)(d31-16) base refinement.
b(a+1)(d15-0) size refinement.

For segment capabilities, the source segment is inspected to ensure that the base refinement does not exceed the current size of the sub-segment nominated by the source capability. The access of the copy is the logical 'and' of the access code in the source capability and the access mask. For non-segment source capabilities, the second word of the capability is copied unaltered; otherwise the base of the copy is the sum of the base of the source capability plus the base refinement, and the size of the copy is the size refinement or the remaining size of the source capability after base modification, whichever is the smaller. For capability segments, it is only permitted to have an even base refinement otherwise capabilities could be misaligned in store. The previous capability at the destination slot is lost and any entry for it in the hardware capability unit is flushed out.

MOVECAPA
ba(d31-16) source capability specifier.
n (d31-16) destination capability segment specifier.
n (d15-0) destination capability segment offset.

The capability specifier part of n selects a capability for a capability segment. Provided that the destination offset is even, (to avoid alignment problems) the source capability is copied without modification to the two words starting at offset in the destination segment. This order is used to transfer capabilities from the capability tables of an address space into general capability segments.

9.7. The P-store and Peripheral Devices.

Device transfers are initiated either by block transfer or by single character transfer instructions depending upon the nature of the device being driven. One argument of the I/O orders is always a device specifier that consists of a sixteen bit capability specifier and a seven bit device number. Before starting a transfer, the I/O orders evaluate the capability nominated by the device specifier to see if it names a segment of memory whose absolute span embraces a word of store, the absolute address of which is given by the device number. Thus, permission to use a device is controlled by the possession of a capability for a word of store associated with the device. The device numbers are in the range nought to thirty-one so that the first thirty-two words of store are tied down for device control and are known collectively as the P-store.

Reading or writing to the P-store has no external effect on devices themselves; the mechanism is purely a contrivance for compatibility between CAP-I and CAP-III and takes advantage of the efficiency of the memory protection arrangements to provide a convenient and fast access check on the use of peripherals. It would be perfectly acceptable to introduce 'device' objects known to the kernel, with device numbers built into them, as a mechanism more in the spirit of the CAP-III kernel design.

All buffering of peripheral transfers is carried out by the Modular One front-end computer and the kernel can transfer a buffer across the link at a comparable rate to the CAP store cycle speed. For this reason, transfers are carried out in a single burst during which no other activity occurs. This greatly simplifies the internal organisation of the kernel and means that, during a block transfer, it is only necessary to evaluate the capability for the CAP buffer at the start of the transfer and, because no other program may execute during a transfer, there is no possibility that the buffer will be swapped out or the capability for it being otherwise invalidated. If CAP had devices that use asynchronous channels directly attached to it, the kernel would have to lock down buffer capabilities in the hardware capability unit and fault any attempt to destroy or modify them.

The kernel does possess this facility although it is used for a different purpose and it will be described in Section 11.3.

9.8. Reference Counts.

Whenever the kernel creates an object by allocating a slot in the map, its reference count is set to one. If a capability for an object is copied, its reference count is incremented and if the kernel overwrites a capability, the latter's count is decremented. In this way, by testing the value of its reference count, the kernel can detect whether or not any references exist to a map slot.

The kernel will automatically return to the free list any map slot whose reference count falls to zero when it is decremented so that the map is kept free of useless entries. If the liberated slot is an extended object, the kernel goes on to decrement the count of the capability which is sealed in its representation and so on, potentially releasing a long chain of slots. It is the responsibility of the operating system to flush out the contents of any capability segment that is no longer active, because clearing out all of the capabilities in a segment could be a deeply recursive and time consuming activity for the kernel to perform.

Unfortunately, reference counts are not sufficient to detect every sort of garbage. In particular cyclic structures can arise; for example, an extended object represented by a capability segment that in turn contains a capability for the extended object is a simple looped structure, and one can construct more complicated ones. Cyclic structures that are inactive will remain in the map because the loops that they contain will prevent reference counts from falling to zero.

To solve this problem, there is a need for a map garbage collector. Since the garbage collector will be system wide, it must run asynchronously to avoid holding up the machine while the map is scanned. In the CAP-I system where each process has a private, synchronous software garbage collector for its Process Resource List, the occurrence of a garbage collection has a noticeable effect on the speed of execution of a process and

therefore, if the CAP-III garbage collector was synchronous, the effect would be greatly magnified across the system as a whole, with a disastrous result on response times and throughput.

The main difficulty with an asynchronous garbage collector is that the map and active capability segments are changing as the garbage collector runs. To cope with this, the kernel sets the reference count marker bit in a map entry whenever it increments a reference count, creates a new map entry or puts a slot on the free list. A simple garbage collector can proceed as follows:

a) Scan the map unsetting all of the reference count marker bits.

b) Use the FREEQ order to determine the free list and set all of the marker bits for slots in the list.

c) Scan the map and all active capability segments marking slots in the map for which there is a valid capability.

d) All slots whose marker bits are unset can be returned to the free list because they are no longer active.

e) Go back to a) and repeat indefinitely.

This garbage collector will detect any slots that were inactive at the start of its scan, and any that become inactive thereafter will be found in subsequent cycles. The size of the map and the frequency with which the garbage collector runs are parameters that must be adjusted in the light of experience in order to obtain the optimum system performance with the minimum overheads. The algorithm given above is based on the CAP-I filing system garbage collector [Birrell and Needham 78].

CHAPTER TEN.

TYPE-EXTENSION AND REVOCATION OPERATIONS.

10.1. Sealing and Types.

The type-extension mechanisms of the CAP-III kernel are based on the facilities proposed by Redell [74], with a number of additions. In the last chapter, two forms of map entry were described, corresponding respectively to basic and extended objects. The representation of basic objects consists simply of binary data, the interpretation of which depends upon the type of the object. Extended objects on the other hand, have a single capability as their representation and objects that are made up of several components can be manufactured by sealing a capability segment that holds capabilities for all of the constituent objects. This scheme has the advantage over the universal objects of HYDRA that all objects have fixed size entries in the map and the problems of handling small bundles of capabilities can be left to the operating system, rather than being the responsibility of the kernel.

The kernel supports three basic primitives with variants for basic and extended objects: SEAL is used to create new objects, UNSEAL to interrogate the representation of an object and ALTER to modify representations. To use one of these primitives it is necessary to quote a capability for a type object that bears the appropriate access code from the set seal, unseal and alter. Type objects are basic objects whose type is recognised by the kernel. The representation of a type object includes a sixteen bit type mark that is found in the type field of all objects belonging to the class nominated by the type object. Operations upon type objects are controlled by a type object whose type mark is 'type'. During the process of sealing, a type object acts as a proforma for setting up a new object, by informing the kernel of the type mark that is to be placed in the type field of the object's map entry. In the other two operations, UNSEAL and ALTER, a type object is used rather like a key to unlock a protected object and the key will only fit if the type mark of the type object is

identical to the type field of the object in question. A type object map entry is shown in figure 10.1-1.

The principal difference between type objects in the CAP-III kernel and Redell's scheme is that Redell places the name of a type object in the type field of protected objects, as opposed to the arbitrary type codes employed by the kernel. Redell's approach is conceptually more simple and unifies the management of types with all names, but in the CAP-III kernel, where names contribute towards reference counts, it is difficult to free slots automatically in the map if the scan of a chain of extended type map entries has to be altered from a linear progression to a tree walk that includes pointers to type objects. It would be unreasonable to ignore the contribution of type fields to the reference count of a type object, because if the count of the type object did reach zero, all of the objects of that type would be left pointing at an invalid map entry. In the CAP-III system, the management of type names must be carried out by the operating system type-object manager, rather than by the kernel unique name mechanisms.

In the case of extended objects, there is no restriction on the type of capability that is sealed in the presence any particular type object. In this way it is possible for an extended object to have different forms of representation and it is up to the type managers of objects with multiple representations to take steps to cope with the range of capabilities that might be extracted by the UNSEAL operation.

The kernel will fault any attempt to unseal data from an extended object or a capability from a basic object, although ALTER will permit the representation of an object to be switched between the two forms. In the next two sections, the variants of SEAL, UNSEAL and REVOKE will be enumerated for data sealing and capability sealing respectively.

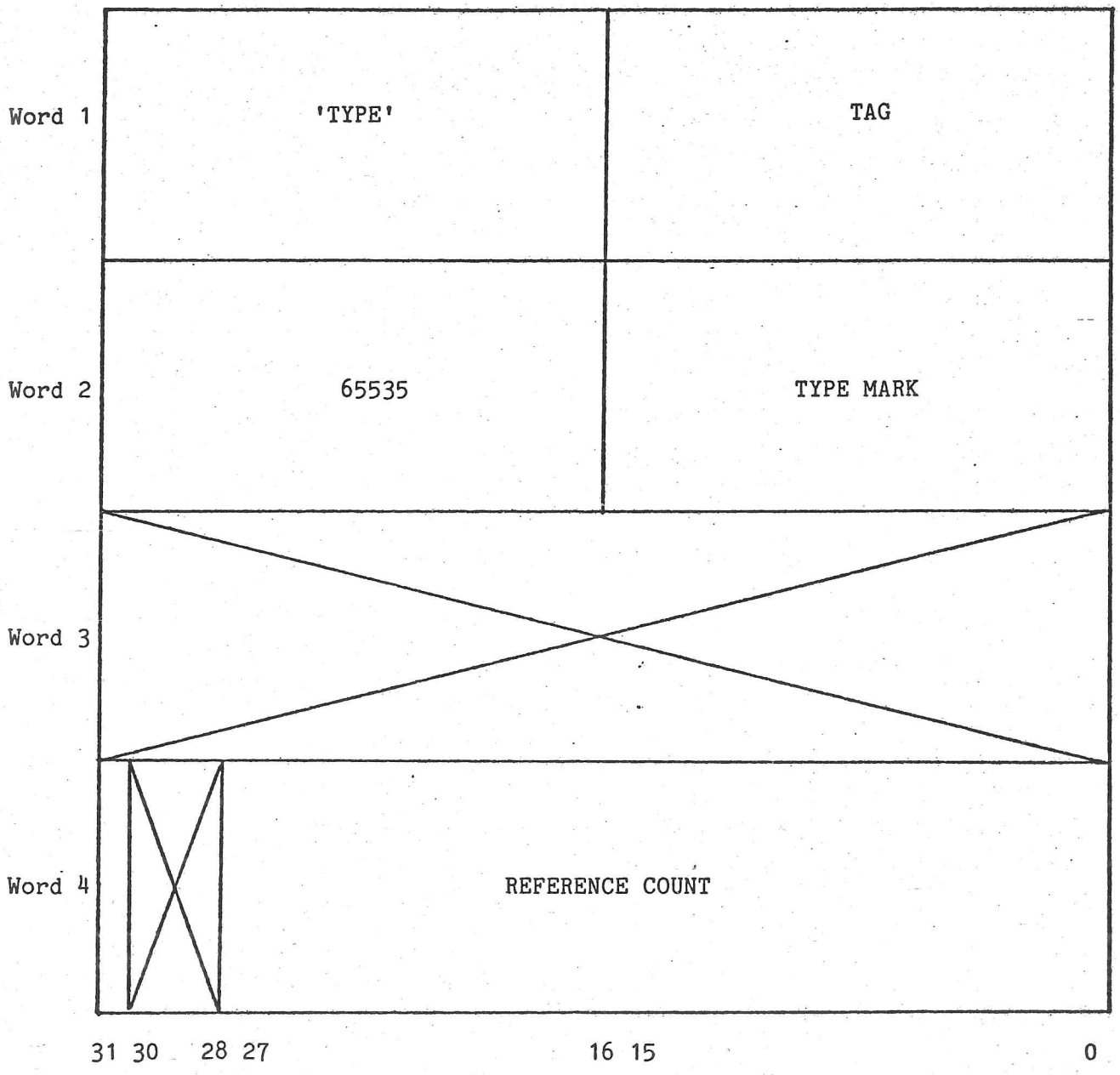


Figure 10.1-1 A Type Object Map Entry

10.2. Basic Objects and Data Sealing.

SEALD (seal data)
ba(d31-16) type object capability specifier.
ba(d15-0) tag for new object.
[bm], [bm+1] representation data.
bn(d31-16) destination capability specifier.

The type object capability is checked for seal access and then a map slot slot is detached from the map free chain and initialised as follows (using the terminology of Section 9.2): the type field is set to be the type mark of the type object, the tag field is set to be ba(d15-0), the representation words are set to be [bm], [bm+1] with the top sixteen bits of [bm] forced to be all ones, the reference count is set to one and the reference count marker bit is set. Finally, a capability pointing to the newly created slot with all the access bits except d15 (the revoke access bit) set, is written to the destination slot. The previous capability in this slot is lost and any entry for it in the hardware capability unit is flushed out. This instruction is illustrated by Figure 10.2-1.

UNSEALD (unseal data)
ba(d31-16) type object capability specifier.
bm(d31-16) basic object capability specifier.
[bn], [bn+1] destination buffer.

The type object capability is checked for unseal access, the type mark of the type object is checked to match the type field of the basic object and then the two words of data forming the representation of the object are copied to the two words of store [bn], [bn+1]. This instruction is illustrated by Figure 10.2-2.

ALTERD (alter data)
ba(d31-16) type object capability specifier.
bm(d31-16) source object capability specifier.
[bn], [bn+1] new representation data.

The type object capability is checked for alter access, the type mark of the type object is checked to match the type field of the source object and the two words of representation in the map entry for the object are overwritten by the contents of the two words of store [bn], [bn+1] with the top sixteen bits of [bn] forced to be all ones. If the previous representation of the

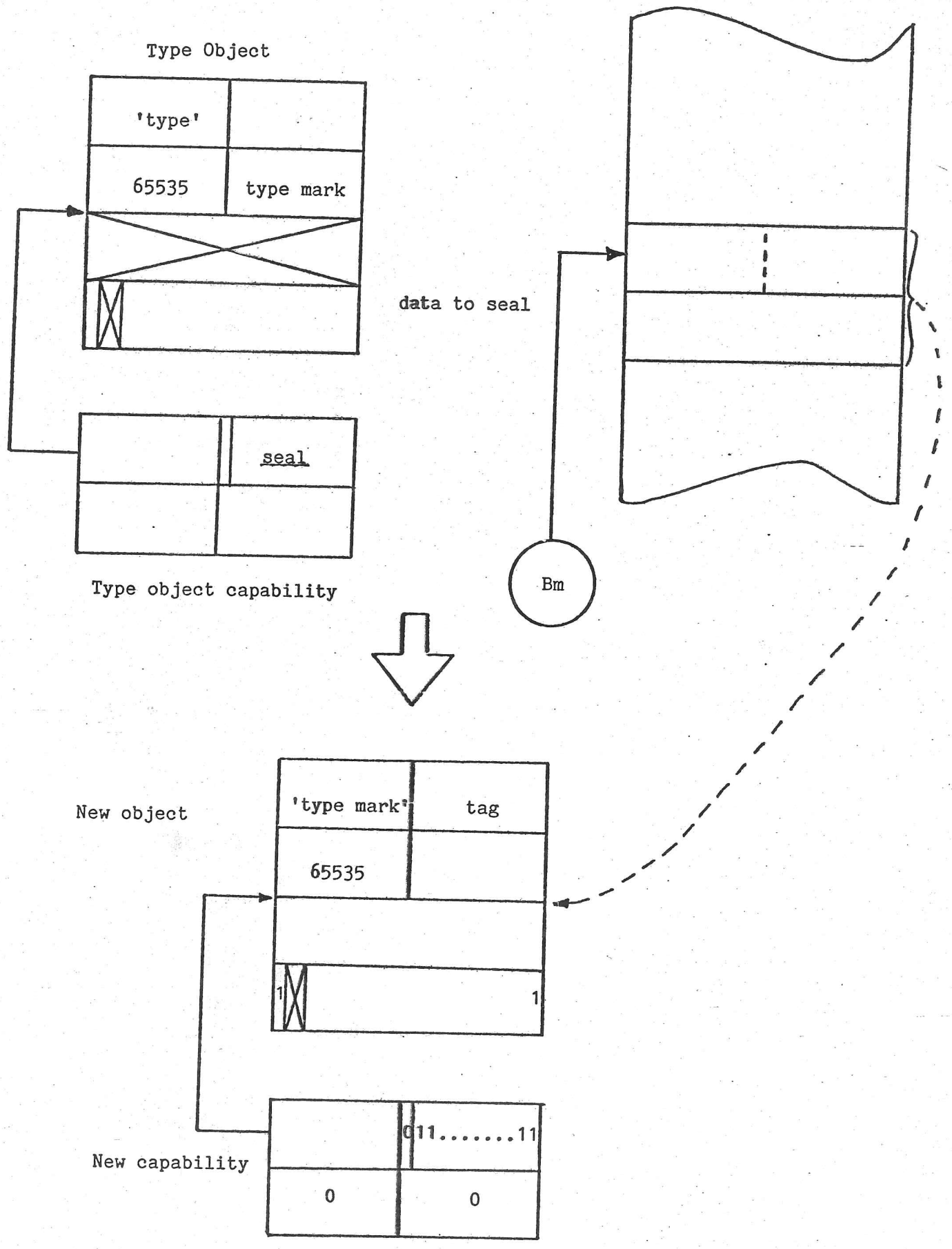


Figure 10.2-1 Action Of The SEALD Instruction

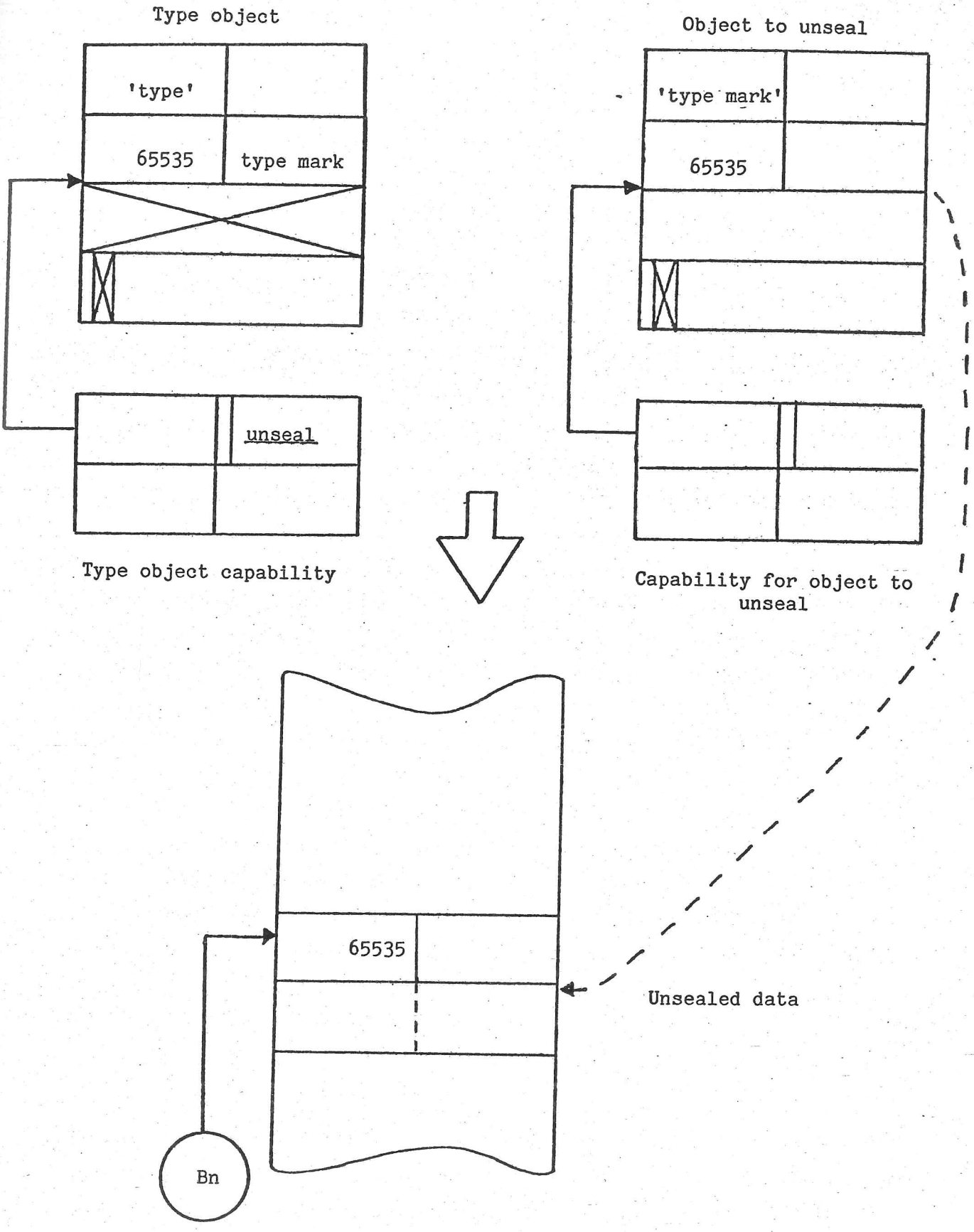


Figure 10.2-2 Action Of The UNSEALD Instruction

source object was a capability, the reference count of the capability is decremented and processed in the standard manner. Finally, any capability in the hardware capability unit derived from the object which has just been modified is flushed out so that it will be subsequently re-evaluated to take account of the new representation. This instruction is illustrated by Figure 10.2-3.

As an example of the use of the data sealing instruction suite, consider the management of segment objects. When a new segment is created by SEALD, the segment type manager will initialise the data represention of the segment to hold the absolute location and size of the segment and set the usage bits to zero. If subsequently it is required to swap in a segment that is on backing store, the segment manager can scan every segment object with UNSEALD to read the usage bits for use in calculating the cost of swapping out currently in-store segments to make room for the one to be brought in. The ALTERD order can then be used to reset the status bits in segment objects once they have been inspected and also to switch on the outform bit in any segment that is to be swapped out. ALTERD may be used to change the representation data of a segment that is repositioned in store by the simple expedient of modifying its absolute address field. These uses of ALTERD justify the extensive scan and flush of the hardware capability unit after exercising the order, as clearly the system has to guarantee that all capabilities for the segments that have been tampered with are freshly evaluated to avoid the risks of accessing the wrong region of memory.

It should be noted that neither data sealing nor the ALTER operation is present in Redell's design and they were invented for the CAP-III kernel to unify and enhance the range of operations that can be carried out on all objects, whether basic or extended. The capability sealing operations described in the next section (with the exception of ALTERC) are much closer to Redell's orders.

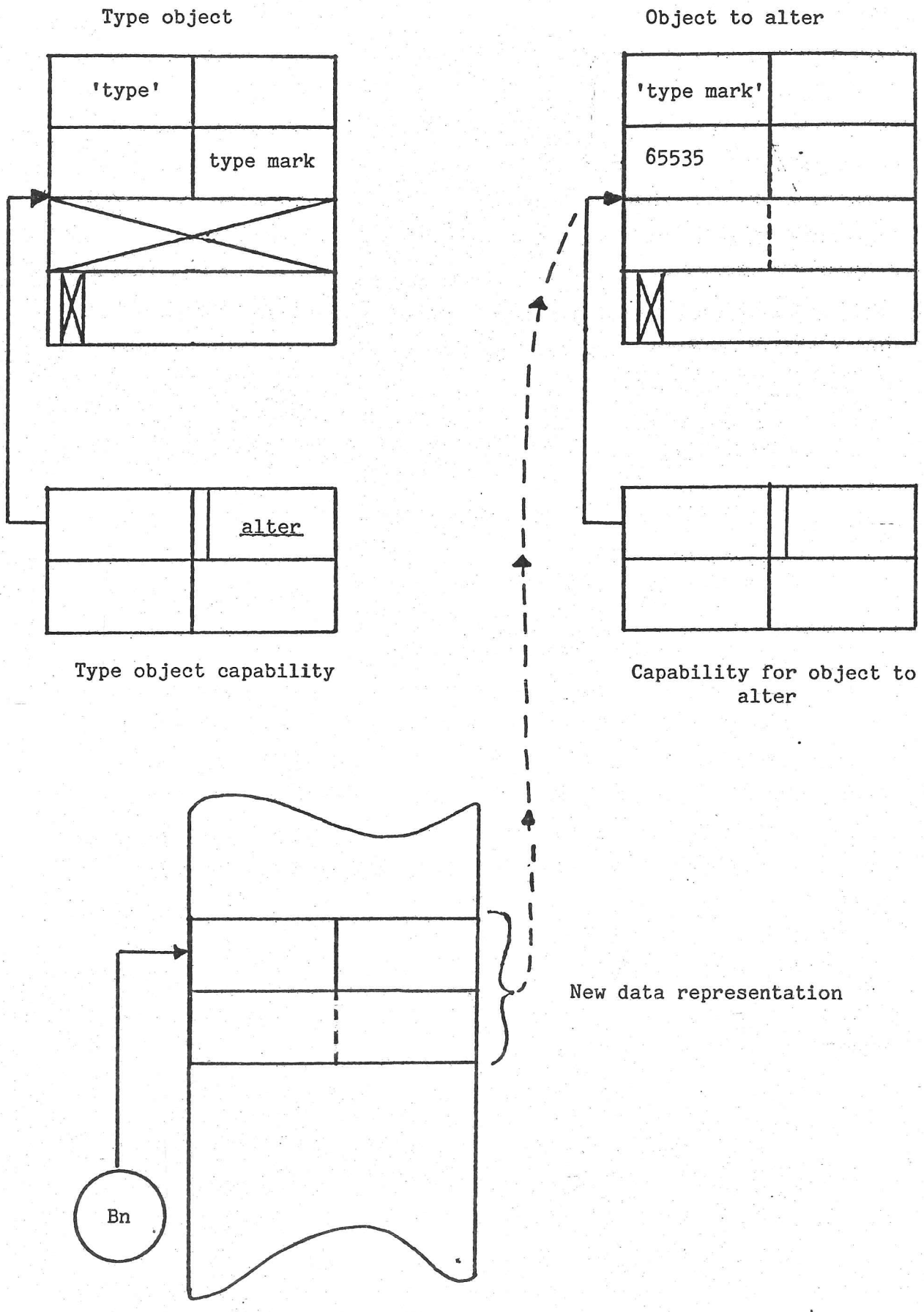


Figure 10.2-3 Action Of The ALTERD Instruction

10.3. Capability Sealing and Extended Objects.

Unlike capabilities in Redell's system, the capabilities supported by the CAP-III kernel contain access code information and, in the case of segment capabilities, refinement data. When a capability is sealed it is necessary to include all of this data in the representation of the extended object, otherwise the process of unsealing would not know how to set up the fields in any capabilities that are extracted from extended objects. The capability suite of type extension orders is listed below:

SEALC (seal capability)
ba(d31-16) type object capability specifier.
ba(d15-0) tag for new object.
bm(d31-16) representation capability specifier.
bn(d31-16) destination capability specifier.

The type object is checked to hold seal access and then a slot is taken off the chain of free map entries and initialised as follows: the type field is set to be the type mark of the type object, the tag field is set to be ba(d15-0), the representation is set to be a copy of the capability specified by bm (which must not be null), the reference count is set to one and the reference count marker bit is set. Finally, a capability pointing to the newly created slot, with all of the access bits except d15 set, is written to the destination capability slot. The previous capability in this slot is lost and any entry for it in the hardware capability unit is flushed out. As a side-effect of this order, the reference count of the object named by the representation capability will be incremented by one. It is not possible to seal a null capability. This instruction is illustrated by Figure 10.3-1.

UNSEALC (unseal capability)
ba(d13-16) type object capability specifier.
bm(d31-16) extended object capability specifier.
bn(d31-16) destination capability specifier.

The type object capability is checked for unseal access and the type mark of the type object must match the type field of the extended object. A copy of the capability representation of the extended object is made in the destination capability slot. The previous

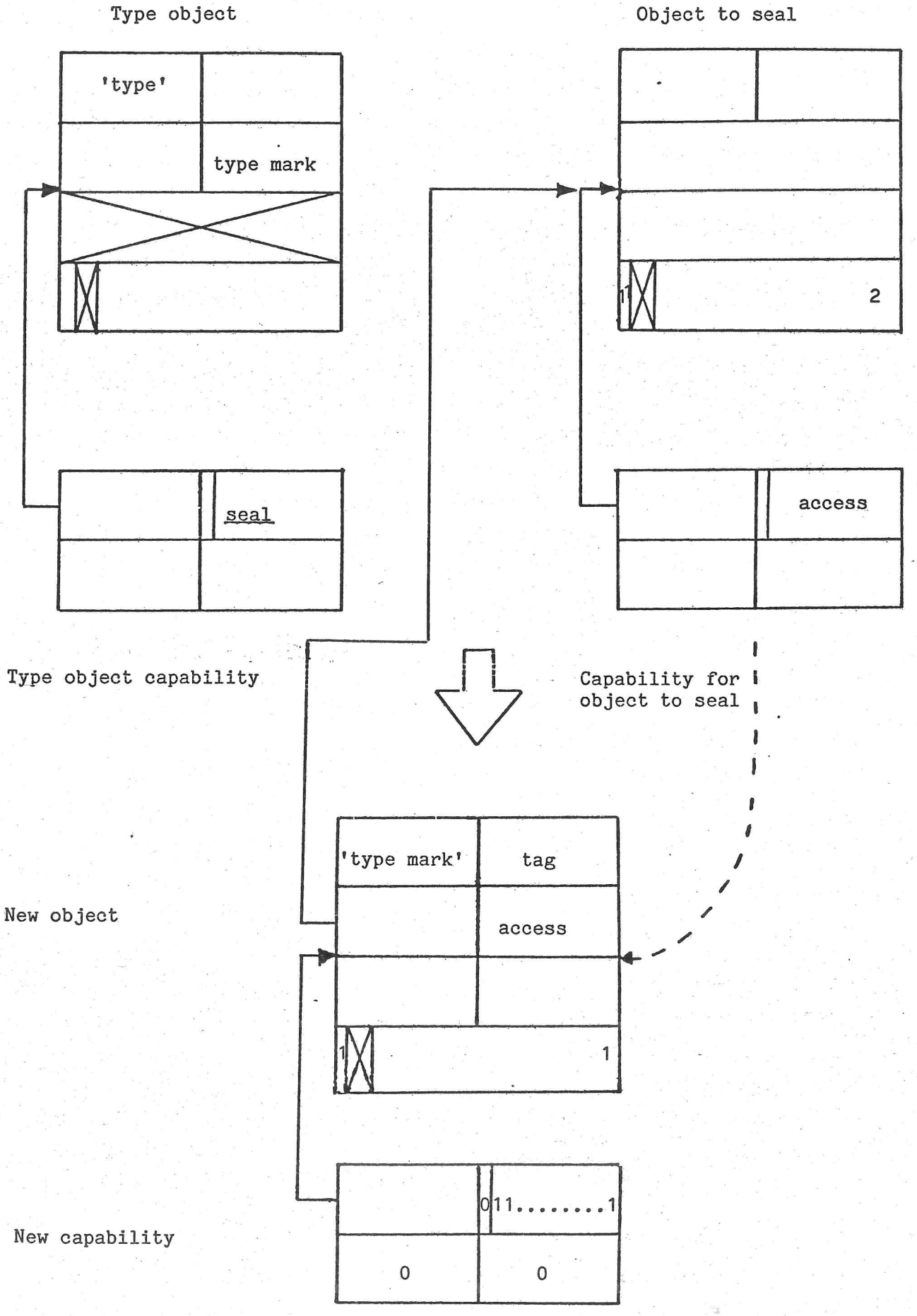


Figure 10.3-1 Action Of The SEALC Instruction

capability in this slot is lost and any entry for it in the hardware capability unit is flushed out. As a side-effect of this instruction, the reference count of the map slot pointed to by the sealed capability will be incremented by one. It is not possible to seal a null capability. This instruction is illustrated by Figure 10.3-2.

ALTERC (alter capability)
ba(d31-16) type object capability specifier.
bm(d31-16) source object capability specifier.
bn(d31-16) destination capability specifier.

The type object is checked for alter access and the type mark of the type object is checked to be identical to the type field of the source object. The two words of representation information in the map slot for the source object are overwritten by a copy of the capability specified by bn(d31-16) which must not be null. This has the side effect of incrementing the reference count of the map slot pointed to by the new representation capability. If the previous representation of the object was also a capability, the reference count of the slot that it pointed to is decremented and processed in the standard manner. Finally, any capability in the hardware capability unit that was derived from the modified object is flushed out, so that it will be subsequently re-evaluated to pick up the new representation. This instruction is illustrated by Figure 10.3-3.

10.4. Revocation.

The kernel revocation mechanism like the type-extension scheme, is closely modelled on Redell's work. There is a special sort of map slot, called a revoker, that does not stand for an object in its own right, which is used to control access to objects. Revoker map entries are recognised by the kernel and the layout of one is shown in figure 10.4-1.

If, in the course of evaluating a capability, the name field points to a revoker map slot, the kernel will modify the computed access code to be the intersection of the access mask in the revoker and the access bits in the capability before going on to

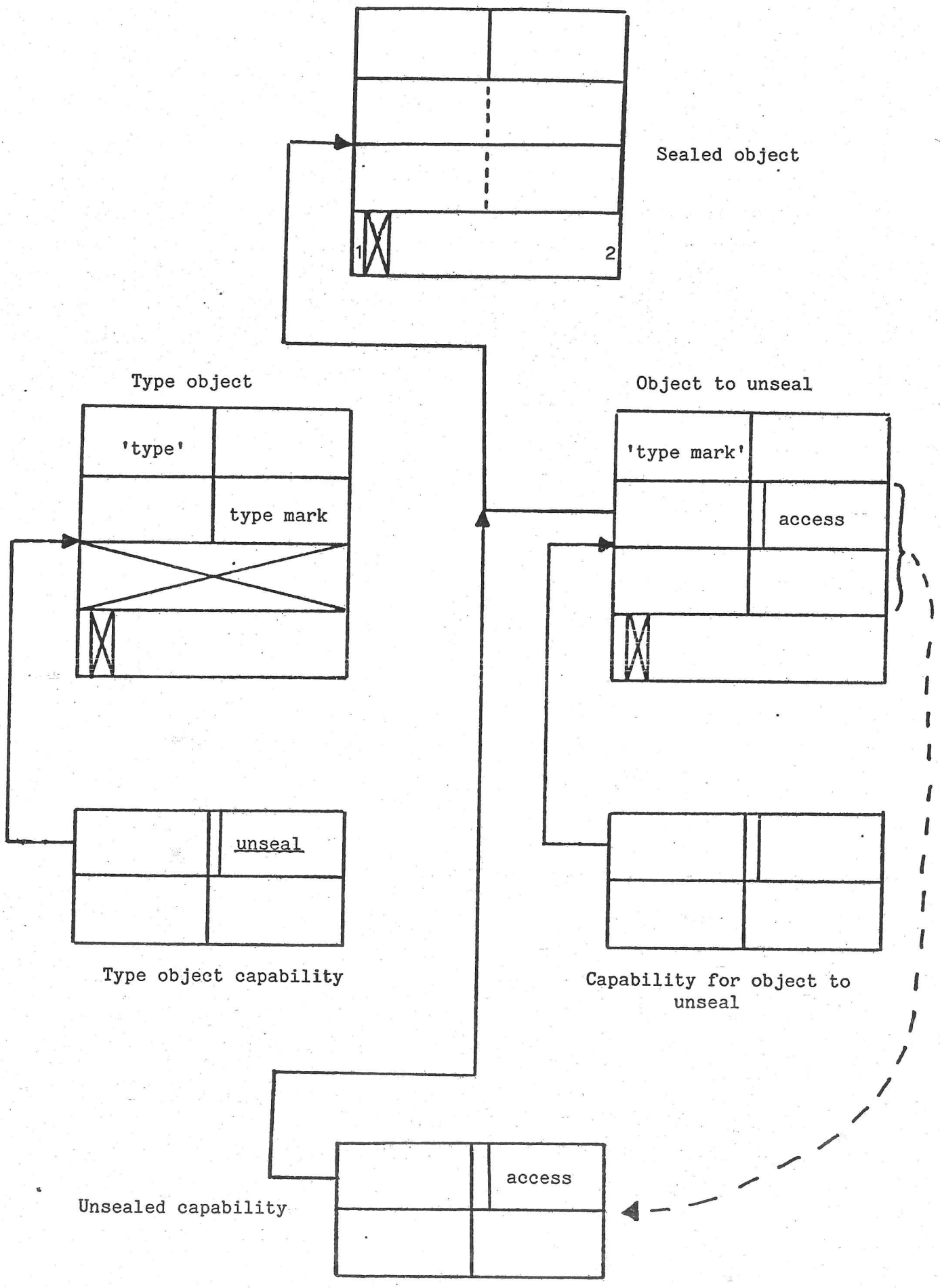


Figure 10.3-2 Action Of The UNSEALC Instruction

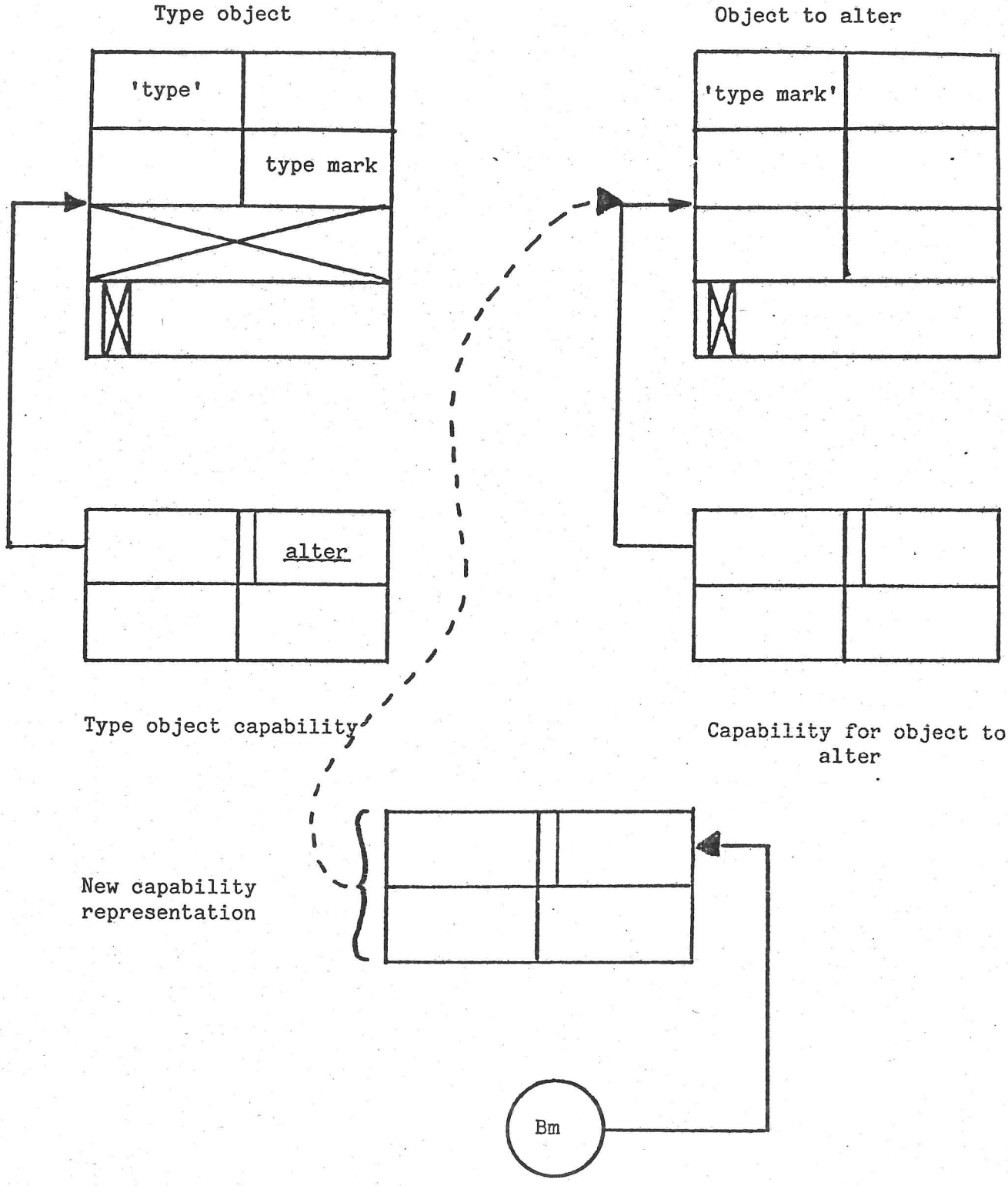


Figure 10.3-3 Action Of The ALTERC Instruction

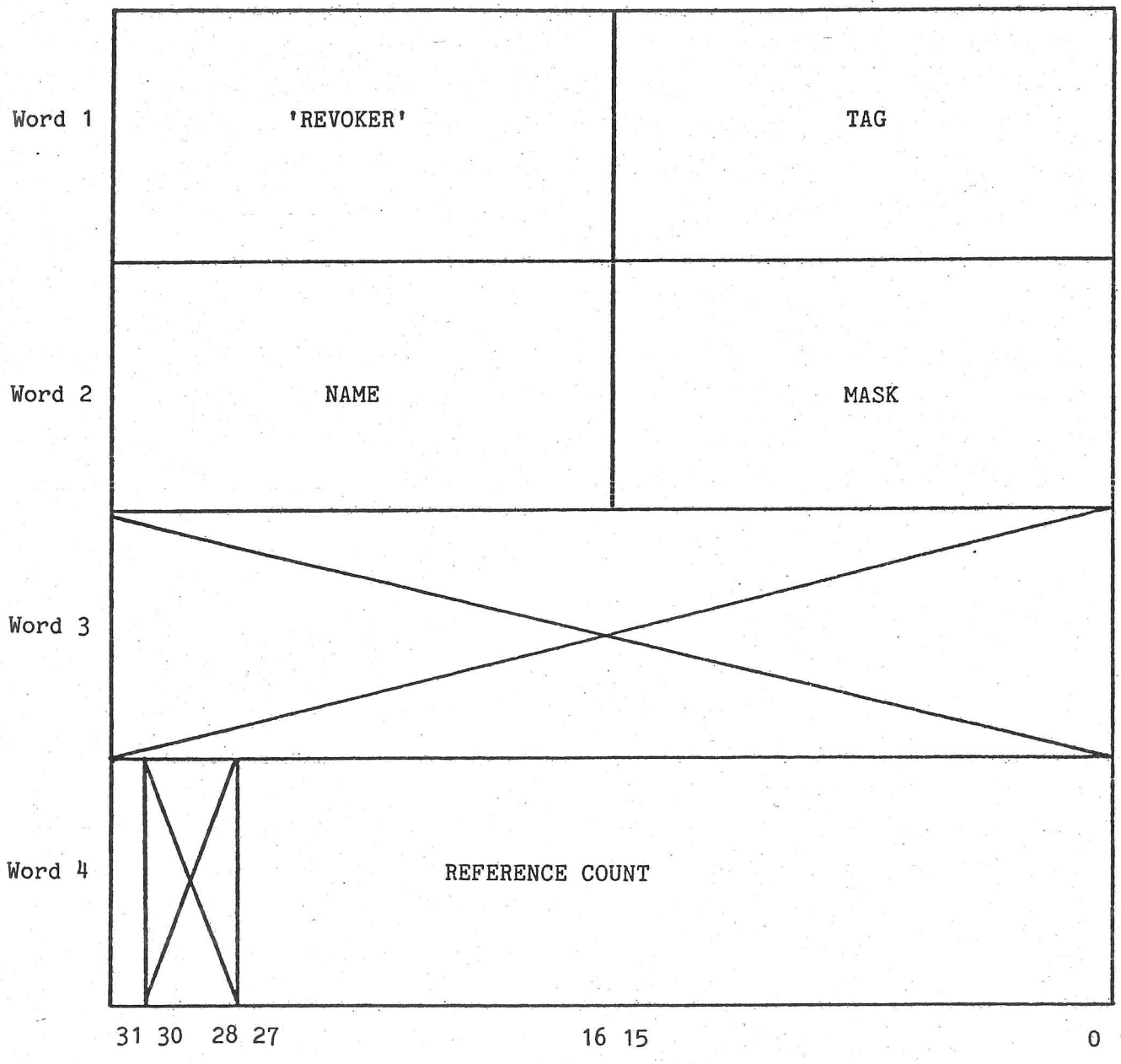


Figure 10.4-1 Revoker Map Entry Format

the map slot nominated by the name field of the revoker. If this slot too is a revoker, the process of access masking is repeated until a non-revoker map slot is reached. The final slot is the one that stands for the object protected by the original capability. Where, previously, the name of a capability has been taken to point at an object, what was actually meant was that the name pointed at a (possibly null) chain of revokers ending with the object in question. Thus, in most cases, any intervening revokers between a capability and an object are transparent to the user, apart from the modification of access codes, so that for example, OBJINF when applied to a capability that points at a chain of revokers will report, not the access code in the capability, but the computed access code together with the tag field of the root object.

Map slot reference count management is done a little differently; whenever a name, be it in a capability or a map slot, is copied or deleted, it is the reference count of the first object in the chain pointed at by the name that is affected, so that revoker map slots will be treated uniformly with those that denote objects from the point of view of automatic deletion.

The access mask of a revoker can be changed by the REVOKE instruction. REVOKE must be applied to a capability, which has a name that immediately points at a revoker and the revoke access bit (d15) that is generic to all capabilities must also be set in its access code. The function of the revoke bit is analogous to the use of lockers in Redell's system in that the presence of the revoke bit conveys the privilege of being able to exercise revocation. If it is desired to copy a revocable capability as a parameter without passing on the right of revocation, a copy of the capability should be made using the access code masking facilities of the REFINE order to cancel the revoke access bit.

Redell makes objects revocable by sealing them in revokers and then returning a capability that points at the revoker, but in the CAP-III system, sealing a capability involves preserving all of the fields of the capability and that would leave no space in a revoker map entry for an access mask. At one stage in the kernel design whole capabilities were sealed and the revoke operation was

defined only to reduce the access code of the sealed capability. This was unsatisfactory as ideally it should be possible to support temporary as well as permanent revocation. Users cannot be allowed to increase the access code of a sealed capability, otherwise they could easily gain extra privileges in their capabilities. To solve this problem the mechanism was revised so that the act of making a revoker does not seal the original capability. Instead a revoker was arranged to be interposed between the capability and the first object that it pointed to. With this organisation it is safe to allow users to increase access in a revoker's access mask because the privileges that they can gain are constrained by the access code in the capability that has been made revocable. Revoker sealing is carried out by the SEALC instruction which behaves in a different way for revoker type-objects from the way it behaves with other type-objects. The specification of the two instructions concerned with revocation follows:

SEALC (capability sealing - revokers only)

ba(d31-16)	revoker type-object capability specifier.
ba(d15-0)	tag for revoker map slot.
bm(d31-16)	source capability specifier.
bn(d31-16)	destination capability specifier.

The revoker type-object capability is checked for seal access and a slot is taken from the chain of free map entries and initialised as follows: the type field is set to be 'revoker', the tag field is set to be ba(d15-0), the most significant sixteen bits of the first representation word are set to be a copy of the name field of the capability, the least significant sixteen bits (the access mask) are set to be all binary ones, the second word of representation is not used, the reference count is set to one and the reference count marker bit is switched on. Finally, a version of the source capability, with d15 (the revoke access bit) of its access code forced to be one, is moved to the destination capability slot. The previous capability in this slot is lost and any entry for it in the hardware capability unit is flushed out. A side-effect of this order is to increment the reference count of the map slot pointed to by the name field of the source capability. It is not

possible to seal a null capability. This instruction is illustrated by Figure 10.4-2.

REVOKE n (d31-16) revocable capability specifier.
n (d15-0) new access mask.

The access code of the revocable capability is checked to contain the revoke access bit and a check is also made that the capability points directly at a revoker map slot. The new access mask specified by n(d15-0) is written in the revoker map slot and any capabilities in the hardware capability unit that were derived from a chain of map slots including the modified slot are flushed out so that they will be computed afresh to take account of the new access mask.

The SEALD, UNSEALD, ALTERD, UNSEALC and ALTERC instructions all signal a fault if an attempt is made to use them in the presence of the revoker type object.

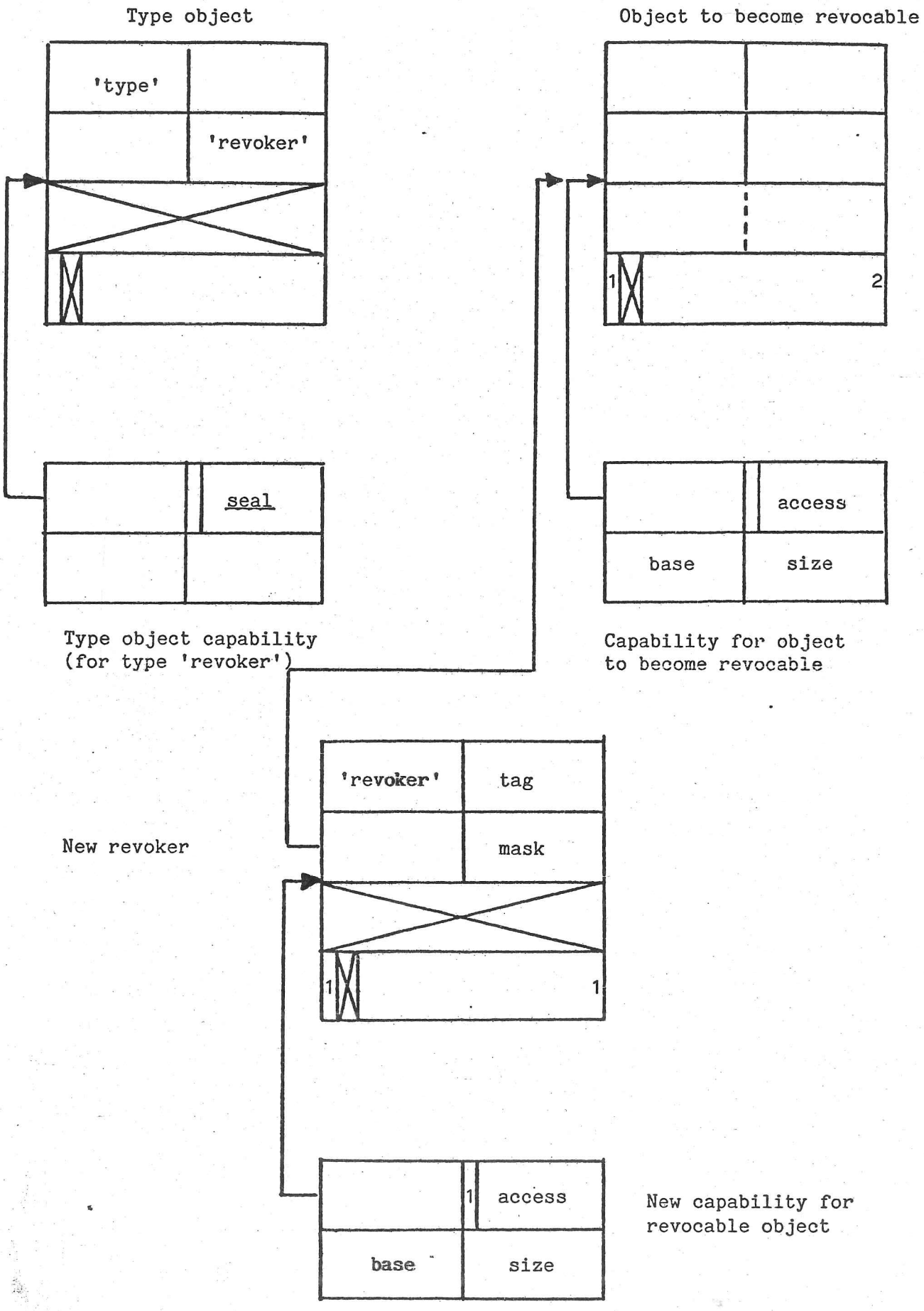


Figure 10.4-2 Action Of The SEALC Instruction For Revokers

CHAPTER ELEVEN.

DOMAIN AND PROCESS STRUCTURE.

11.1. Preliminaries.

The CAP-III kernel attempts to unify inter-process and inter-domain communication by making each protection domain a separate process in conjunction with a message system that is equivalent, in both speed and power, to the domain call machinery of the CAP-I memory protection system. There is no within-process communication facility which means that the analogue of a CAP-I process with many protected procedures will be a cooperating group of processes in CAP-III and it may be hoped that the greater potential parallelism of the CAP-III arrangements will have beneficial effects in the area of efficiency. The message system is designed to handle procedure call like communication in a simple and direct fashion, while at the same time providing support for more complicated protocols. The global naming scheme employed by CAP-III makes it possible for all varieties of objects to be sent as the contents of messages without any need for translating names or duplicating data structures. There are no problems concerned with parallel execution within a single domain because each process can control its own activations by choosing to accept and either handle just a single message, or many messages, at a time.

The advantages of a non-hierarchical domain structure within CAP-I processes suggests that in CAP-III it would be most suitable to have a non-hierarchic process structure and that all processes should be equal in status. However, the CAP-III kernel needs to be able to notify an operating system about faults, interrupts and scheduling requests. For this purpose one process, known as the Interrupt Process, is distinguished from all others. The kernel will cause this process to run whenever it wishes to inform the operating system about some event and it is also the initial process to be run by the kernel when the system is loaded. The remaining processes in the system are treated equally by the kernel, although the operating system can elect to set up a

dynamic hierarchy of control by making the scheduling of some processes the responsibility of others; these coordinator processes will notify the Interrupt Process of their intentions to get them carried out.

11.2. Data Structures.

A process is represented in the map by a process object which contains a capability for the domain descriptor of the process. The first sixteen slots in this capability segment are for the capability tables of the process and there are two further slots that define other parts of a process's apparatus as described in the previous chapter. The seventeenth slot defines the process base which is a data segment. These data structures are illustrated in Figure 11.2-1. Like the domain descriptor, the process base is private to the process. Part of it is a dump area to hold the contents of the processor registers whenever control leaves the process. There is a time-slice word that contains a negative count and this is incremented at regular intervals while the process is running. Whenever the count reaches zero an interrupt is generated to signify that the process has exhausted its ration of time. Another word holds a wake up waiting flag that is used to prevent a program from accidentally ignoring some event, such as the arrival of a message or a peripheral interrupt. The flag is set whenever the kernel wishes to notify a process about an event and discovers that the process is already active. A process is prevented from waiting when the flag is set, and any attempt by the process to hold up results in an immediate resumption so it can then poll message channels and peripherals to discover the exact nature of the wake up. The condition of a process is held in a state word in the process base and this can take one of two values 'active' or 'held up'. An active process is one that is free to run whenever the processor is available and a held up process is one that is awaiting an event. The priority word in the process base holds a numeric value that is used when the kernel has to choose between processes that are free to run; in such a case the numerically highest priority will take precedence.

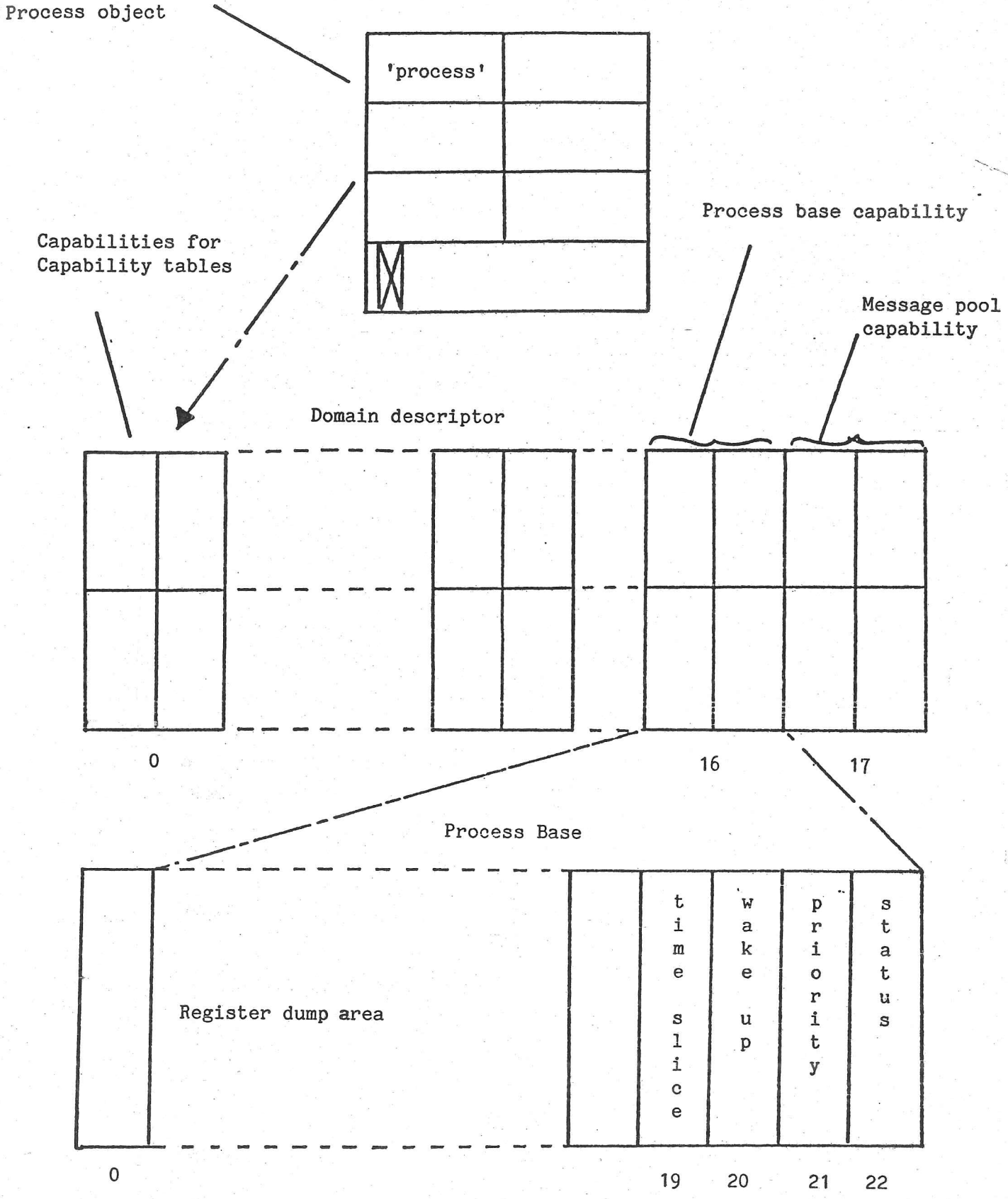


Figure 11.2-1 Process Data Structures

The messages that can be sent between processes take the form of capability segments, known as message blocks, which are allocated from larger segments known as message pools and are shown diagrammatically in Figure 11.2-2. The eighteenth slot in a process's domain descriptor contains a capability for the pool from which the process's message blocks are to be issued. A message pool can be local to a single process or common to a group of processes and there is no need for message pools to be resident in memory, as the kernel treats the message system segments uniformly with ordinary segments. This allows us to escape from the resource control problems of Watson's system with its single, central, resident pool. CAP-III message pools can be part of virtual memory and swapped or extended at will. The first two capabilities in a message pool are head and tail pointers to a chain of inactive message blocks within the pool. These pointers, in common with others in the message system data structures, take the form of refined capabilities for sub-segments of message pools so that the kernel reference count mechanisms will lock down any apparatus belonging to an active message transaction. The null pointer is indicated by a null capability.

Message blocks are of fixed length and can hold five capability arguments. There is no provision for data arguments as such, but of course it is possible to pass a capability for an area of store that embraces some data parameters. There is a link capability which is used to hold a pointer to the next message block in a chain, such as the free chain in a message pool or a queue of messages waiting on a message channel. A message block holds information about the size of the pool from which it was allocated and its offset within the pool, so that it is possible for the kernel to undo the refinement data in a message block capability when it wishes to gain access to the pool to return a dead message to the pool's free list. This may take place in a process different from the one which constructed the block in the course of a non-reply type transaction or during fault processing. A message block may be labelled with a tag when it is created and this tag can then be used by processes that multiplex messages to correctly identify replies to them. To facilitate replying to

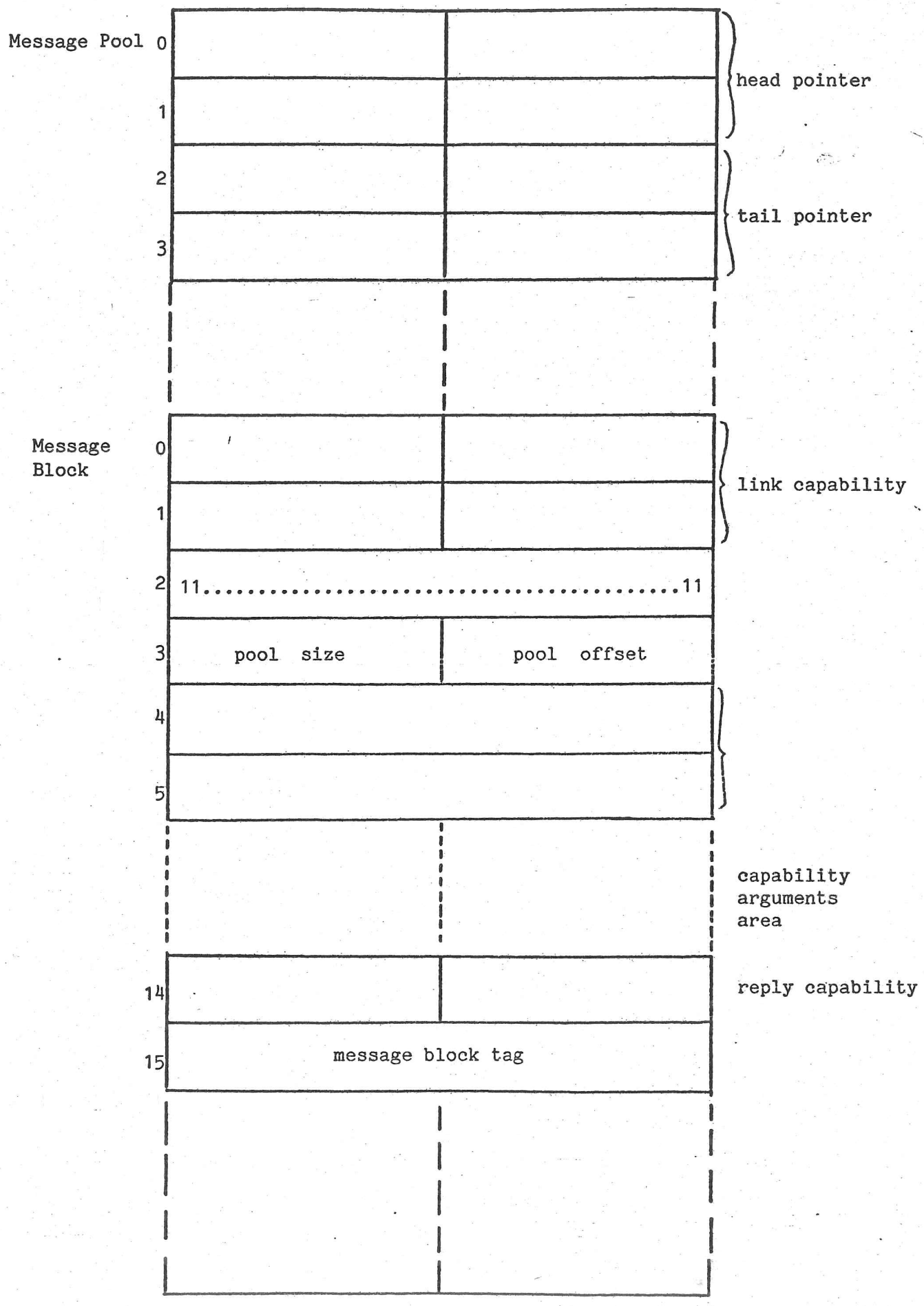


Figure 11.2-2 A Message Pool and Message Block

messages, a capability for a message channel, known as the reply channel, can be incorporated into a message block so that the recipient of the message can return it without requiring a pre-existing channel to the sender. A non-reply type message will have a null capability instead of a channel capability in this field. Users only see message blocks as capabilities for message objects and they are not given access to the contents of a block except through the kernel operations to load and unload argument capabilities.

Message blocks as such are never given directly to processes. Instead they are always encapsulated in message objects which are extended objects (whose type 'message' is recognised by the kernel) and have a segment capability for a message block as their representation. This is done so that the kernel can cause a process to relinquish access to a message block by updating the contents of a message object no matter however the process has duplicated and distributed capabilities for it.

Messages are sent along message channels which are described by channel objects that have a capability segment as their representation and a channel is shown in Figure 11.2-3. The first two capabilities in the segment act as head and tail pointers for the queue of messages despatched on the channel that have yet to be received. If there are no outstanding messages the pointers will be null. The remaining, third capability in the channel segment is a capability for a process object defining the process to be woken up whenever a message arrives on the channel. The refinement field of this capability is utilised to hold a count of the number of messages queued upon the channel. There are two access codes, send and receive associated with message channels that are used to control the transmission and reception of messages on the channel respectively.

A more general scheme would be to introduce 'mail boxes' into which any process could deposit messages and can be served by a number of processes. However this would require the kernel to keep track of all the processes waiting on a channel and then to wake them all up when a message arrives. This is a complicated

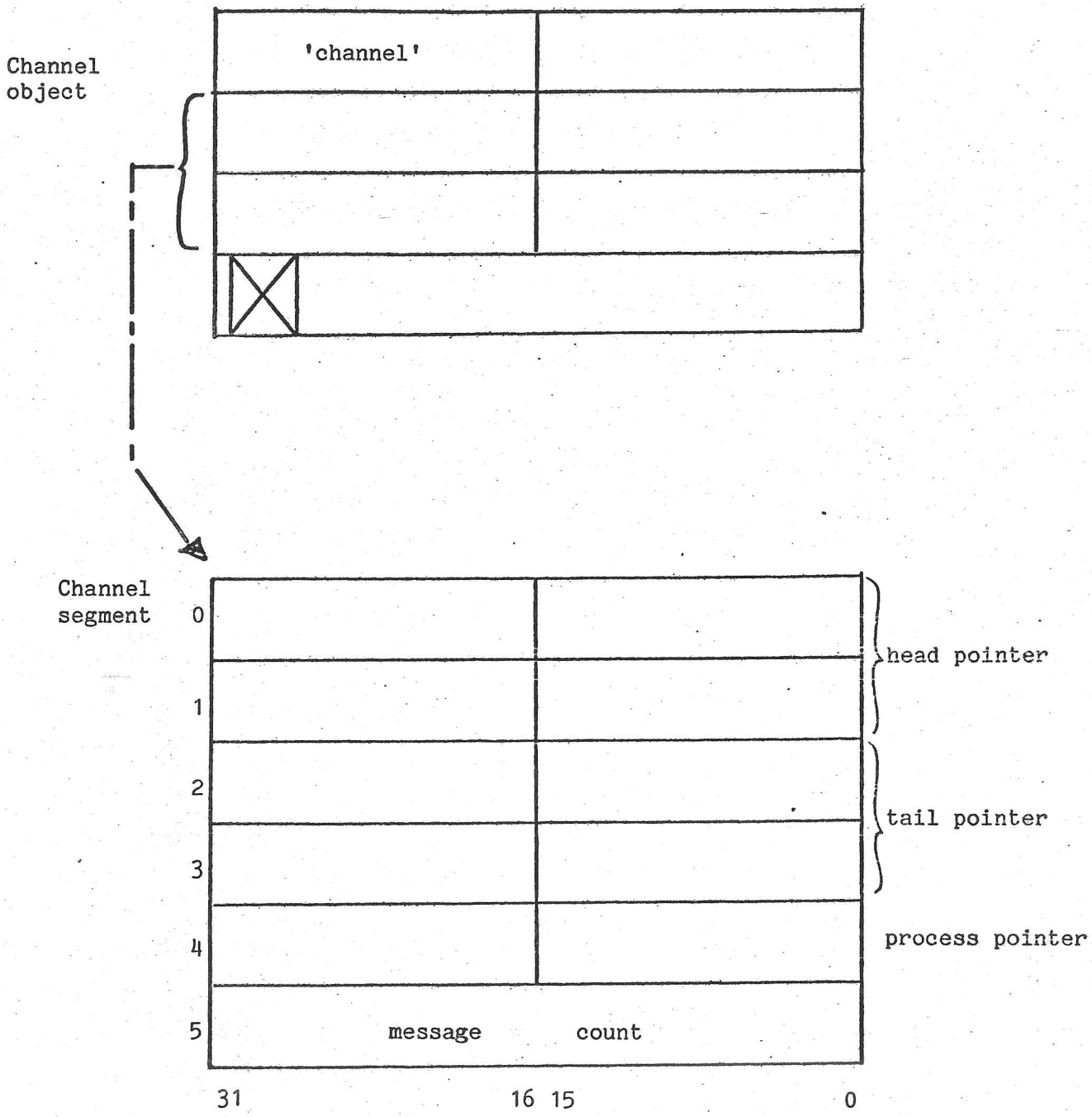


Figure 11.2-3 Message Channel Format

task that it would be difficult to do efficiently in microprogram. Instead, a slightly simpler approach has been chosen, in which only one process can be attached to a channel at any one time. There is complete freedom in the number of channels by which a process can send and receive messages.

A process can obtain a message object ready for use by application of the MAKEBLOK order which takes a block off the process's message pool free chain. It is possible to specify the tag of the message block and also to set up the reply capability to be either a channel capability or null. The result of this operation is a capability for a message object encapsulating the message block.

Capability arguments are put into a message by the PUTARG instruction and can be extracted by GETARG which are rather similar in function to the MOVECAP instruction for shipping capabilities between capability tables.

When a message block is finished with, the KILLBLOK instruction can be used to overwrite any capabilities in the block with null capabilities and then to return the block to the pool from which it originated. The destruction of the capabilities within the block prevents dead messages from wastefully keeping slots tied down in the map. Once a message has been killed, it is necessary to prevent the message object from being used to access the message block any more, and this is done by replacing the capability for the message block in the message object with a data type representation that will cause any attempt to extract a capability from the message object to fail. The invalidation of the message object in this way will cause any occurences of the message block in the hardware capability unit to be lost. This contrivance is used in several places in the message system whenever it is desired that a process should lose access to a message. The instructions concerned with message blocks are listed below:

```
MAKEBLOK ba(d31-0)     message block tag.
         bm(d31-16)    reply channel or null capability specifier.
         bn(d31-16)    destination capability specifier.
```

If the reply capability is not null, it must possess

the send access code. The head and tail pointers of the current process's message pool are inspected to see if there is a free message block available. If there is not, a fault is signalled and the instruction terminates, otherwise the head block on the free chain is detached and initialised as follows: the size and pool offset fields are set up to describe the current message pool, the tag field is set to be ba(d31-0) and the reply capability is copied to the reply capability slot in the block. A message object is set up in the map with a capability for the message block as its representation and a capability for this object is copied to the destination capability slot. The previous capability in this slot is lost and any previous entry for it in the hardware capability unit is flushed out.

KILLBLOK n (d31-16) message object capability specifier.

If there is an unused reply capability in the message, a fault is signalled and the instruction terminates. Otherwise the pool size and offset fields within the block are used to reconstruct a capability for the message pool from which the block was allocated and then the block is attached to the free chain in the pool. All of the capabilities in the message block are set to be null and the message object is then invalidated to ensure that the message object cannot be used again and any entries for it in the hardware capability unit are flushed out.

PUTARG ba argument number.
bm(d31-16) message object capability specifier.

The argument number must be in the range 0 to 4 inclusive. The capability specified by bn is copied to the (ba)-th argument capability slot in the message block.

GETARG ba argument number.
bm(d31-16) message object capability specifier.
bn(d31-16) destination capability specifier.

The argument number must be in the range 0 to 4 inclusive. The (ba)-th argument capability in the

message object is copied to the destination capability slot. The previous capability in this slot is lost and any entry for it in the hardware capability unit is flushed out.

11.3. The Interrupt Process.

The kernel transfers control to the Interrupt Process whenever an interrupt or fault occurs. If a fault arises when the Interrupt Process itself is running, the kernel will print some diagnostic information on the CAP intimate teletype and resign. The major implication of this is the restriction that the Interrupt Process must be resident, as it must not be subject to any virtual memory faults. Interrupts are held off by the kernel when the Interrupt Process is running but they will automatically lead to a resumption of the process if it tries to transfer control elsewhere.

The Interrupt Process can start up other processes by using the WAKEUP order which takes a capability for a process object as its argument. To succeed, the corresponding process must be marked as active in its process base, otherwise control will remain in the Interrupt Process and a characteristic code will be delivered by WAKEUP. To start up the process, the kernel will preserve the processor registers in the Interrupt Process's process base and reload them from the process base of the target process. Then the capabilities of the Interrupt Process will be made inaccessible and control can be switched to the new process. When control is subsequently redirected back to the Interrupt Process, in response to an external interrupt or some other event, a register specified by the WAKEUP instruction will be set to a characteristic code which can then be interpreted to discover what has happened.

The Interrupt Process can determine which device caused an external interrupt by using the WAIT 1 instruction which will return the number of the device with the longest outstanding interrupt. The Modular One front-end computer will maintain a queue of interrupts and it may take several applications of the WAIT 1 operation to clear the entire queue. The invalid device

number zero is returned if the queue of waiting interrupts is exhausted.

If the Interrupt Process has no useful work to perform because all processes are held up awaiting some event, by obeying the WAIT 0 instruction it can request to be suspended until an external interrupt arrives. It is normal practice to follow a WAIT 0 by a WAIT 1 to determine where the interrupt came from.

The wake up waiting flag of the Interrupt Process is used to prevent the process from losing interrupts that occur while it is running. The flag is set whenever an interrupt arrives while the process is active and the flag can only be cleared by use of WAIT 1. If the process tries to obey either WAKEUP or WAIT 0 when the flag is set, the process will resume immediately and must attempt to clear the condition.

It is essential to the kernel interrupt handling mechanisms that the process base and domain descriptor of both the currently active process and the Interrupt Process are not flushed out of the hardware capability unit otherwise it will be impossible to dump and restore processor registers. A fault will be signalled if either the REVOKE or the ALTER operations attempt to modify a capability that would in turn lead to the removal of the critical capabilities from the unit.

The specification of the Interrupt Process orders is given below. In ordinary processes the WAKEUP order is defined to be a null operation.

WAKEUP n (d31-16) capability specifier for a process object.

If the Interrupt Process wake up waiting flag is set then:

```
ba := #e000ffff
```

and execution continues in the Interrupt Process. Otherwise if the process base of the nominated process is marked as being held up then:

```
ba(d31-16) := #a000
ba(d15-0)  := tag field of process object
```

and execution continues in the Interrupt Process. If, on the other hand, the process was marked as active, the

processor registers are dumped into the Interrupt Process's process base and restored from the target process's process base. When control eventually returns to the Interrupt Process, ba will be set as follows:

ba(d15-0) := tag field of currently running process.

ba(d31-28) := interrupt code thus:

```
#0 - current process has become held up
#1 to #9 - unused
#a - see above
#b - unused
#c - time-slice exhausted
#d - device busy hold up (d27-16 is a device number)
#e - external interrupt
#f - fault                (d27-16 is the fault code).
```

WAIT n (d0) wait code.

n = 0

If the wake up waiting flag of the Interrupt Process is set, execution continues normally; otherwise, the Interrupt Process is suspended until an external interrupt arrives.

n = 1

ba := number of head device in Modular One interrupt queue (0 if queue is empty).

11.4. Scheduling.

The kernel carries out some simple scheduling operations during message transactions by manipulating the priority and status words in process bases. When one process transmits a message to another, the kernel first inspects the state of the destination process. If it is active, the kernel will set its wake up waiting flag to indicate the presence of more work to do; otherwise, if the process was previously held up, its state word is set to be 'active'. In the latter case, the kernel will then go on to compare the priorities of the two processes involved and if the priority of the destination process is the greater of the two, control is switched to that process.

A process may request to be held up until some event occurs, such as the arrival of an interrupt or a message. In this case, the kernel first checks that the process's wake up waiting flag is not set and then marks the state word of the process base with the value 'active' before returning to the Interrupt Process with the

interrupt code #0. The automatic 'binary' scheduling carried out by the kernel means that the process that is running when the Interrupt Process is eventually re-entered after a WAKEUP may not necessarily be the process that was originally started off. This is the reason for including process object tags in interrupt codes so that the Interrupt Process can identify the current process. A process that has been marked as being held up will only become active again due to the arrival or an event or by a message of the modification of the state word in its process base by explicit software action.

Processes are forcibly held up if they attempt a transfer on a peripheral device that is busy. In these circumstances, the program counter of the process is set back so that when the process is resumed, probably in response to an external interrupt, the failing order will be retried. To resume such a process the operating system must set the state word of its process base to the value 'active'. Peripheral transfers carried out by the Interrupt Process are treated specially in that the kernel will hold up the Interupt Process if it attempts to access a busy device until an external interrupt arrives.

A process may elect to be held up immediately after transmitting a message and this results in the process being marked held up, provided that its wake up waiting flag is not set. If the priority of the destination process is greater than or equal to the priority of the sending process, the kernel transfers control to the former; otherwise, if the destination process priority is lower, the kernel will enter the Interrupt Process with the interrupt code #0 because there may be more deserving processes of an intermediate priority that can be run.

Thus it may be seen that the kernel is responsible for simple scheduling decisions involving two processes and a software process coordinator need only be invoked when a comprehensive general reschedule is required. This intervention of the kernel in the scheduling of processes greatly contributes to the effectiveness of the CAP-III message system.

11.5. The Message System.

The SEND instruction is used to despatch messages and takes two arguments: a channel capability specifier and a message capability specifier. The channel capability must hold the send access code, and the message object must have a message block as its representation. The message is chained onto the tail of the queue of the messages attached to the channel, and then the message object named by the message capability has its representation invalidated to prevent further access to the message in the sending process. The kernel will then schedule between the current process and the destination process according to the rules outlined above.

A process may find out how many messages are waiting on a message channel by use of the MESSAGES instruction which has the specifier of a channel object as its argument and returns the message count in the channel segment as its result.

A message may be received by use of the RECEIVE instruction which has a channel capability specifier and a destination capability slot as its arguments. The channel capability must hold the receive access code. If there are no messages queued upon the channel, the current process's program counter is set back and the process is held up until the arrival of an event signal when the channel will be inspected again. This affords a mechanism for making a process wait for a message on a single channel. If it is required to poll a group of channels, the MESSAGES operation should be used first of all to see if there is a message on one of the group of channels before a RECEIVE order is obeyed. If there is a message on the channel, RECEIVE will construct a new message object whose representation is the head message block queued on the channel and it will copy a capability for this object to the destination capability slot. The tag of the message block is also made available so that replies can be distinguished. The head message segment will be stripped off the channel queue and the channel message count will be decremented.

The newly made message object is suitable for interrogation by GETARG to extract argument capabilities and then result

capabilities can be loaded into the message with PUTARG. It is also possible to retransmit the message to another process with the SEND order, in which case the retransmitting process will lose all further rights of access to the message. This facility is of use to processes that act as gate-keepers for other processes. It should be noted that when the final recipient of a message replies to it, the reply returns directly to the originating process of the message and no action is required of the process that took part in forwarding the message.

If a process runs out of things to do and wishes to await the arrival of a message it can execute the WAIT order which will cause it to be held up, subject to the value of its wake up waiting flag. When a process is woken up after a WAIT it may inspect its incoming message channels with the MESSAGES order to decide upon which channel the message arrived. If the process is only interested in a single channel, it may wait upon it alone by using the RECEIVE order.

The remaining primitive in the message system suite is called REPLY which has a message capability as its argument. REPLY will extract the reply channel capability from the message and, if it is null, the message will be disposed of identically to the way in which KILLBLOK behaves; so that the effect of replying to a non-reply type message is to throw the message away. If, on the other hand, the reply capability is valid, REPLY is equivalent to SEND with the reply channel as its argument together with the additional feature of overwriting the reply capability in the message segment with a null capability. This is so that the reply channel capability cannot be exercised again. The reply capability in a message can be considered to be an analogy of a 'reply-paid envelope'.

There are two instructions called SENDW and REPLYW, that are the waiting variants of SEND and REPLY respectively. They are functionally equivalent to the conjunction of a SEND or REPLY order and the WAIT order. These two orders are used to indicate that a process has no more useful work to do after transmitting a message.

The normal cycle of events for two processes communicating in a procedure call like fashion, analogously to inter-domain calls in other systems, is for the calling process to manufacture a message using the MAKEBLOK order, load it with argument capabilities using PUTARG and then transmit it along a message channel to the target process. The sending process will use the SENDW instruction so that it goes to sleep until some event, such as a reply to this message, occurs. The called process will be woken up and the RECEIVE order can be used to pick up the newly arrived message. Arguments in the message can be extracted with GETARG and then, after processing be the message, result capabilities can loaded into the message with the PUTARG order before returning it with the REPLYW instruction. This latter order will also put the called process back into the waiting state until the arrival of a further message, whereupon the cycle can be continued. The return of the message block will re-awaken the sending process which can pick up the message with RECEIVE and extract the results with GETARG before disposing of the message with KILLBLOK.

By using other combinations of the message orders it is possible for a process to handle requests upon several channels concurrently and to construct non-reply type messages or pass received messages on to other processes in support of more complex communication protocols. In these cases, the tag field in message objects provides a simple mechanism to enable a process to recognise replies that arrive in an order different from that in which the original messages were sent out.

11.6. Specification Of The Message System.

WAIT ba(d27-16) information code.

If the wake up waiting flag of the current process is set, clear it and continue execution. Otherwise, preserve the state of the process in its process base and transfer control to the Interrupt Process setting the interrupt information code as follows:

d31-28: #0
d27-16: ba(d27-16)
d16-0 : tag of map entry for current process.

SEND ba(d31-16) message object capability specifier.
n (d31-16) channel capability specifier.

The channel capability is checked for send access. The message block specified by the message capability is chained onto the queue of messages on the channel. Adding the message onto the channel queue has the effect of incrementing the channel message count. The process associated with the channel is then woken up and a schedule between the sending process and the target process takes place.

SENDW ba(d31-16) message object capability specifier.
n (d31-16) channel capability specifier.

This order is similar in effect to SEND except, provided that its wait up waiting flag is unset, the current process is held up.

REPLY n (d31-16) message object capability specifier.

If the reply capability in the message block specified by the message capability is a null capability, the effect of this order is identical to KILLBLOK. Otherwise the reply channel capability is extracted from the message block and replaced by a null capability whereafter the order behaves similarly to SEND with the reply channel as its channel argument.

REPLYW n (d31-16) message object capability specifier.

This order is similar in effect to REPLY, except, provided that its wake up waiting flag is unset, the current process is unset.

RECEIVE bm(d31-16) message channel capability specifier.
bn(d31-16) destination capability specifier.

The channel capability is checked for the receive access code. If there are no messages queued upon the message channel, the program counter of the current process is stepped back and the process is held up and an enforced entry is made to the Interrupt Process. Otherwise the leading message block is taken off the message queue and a capability for a message object defining it is moved to the destination capability slot. The message block tag of the received message is loaded

in Ba(d31-0). This results in a decrement of the channel message count. The previous capability in the destination capability slot is lost and any entry for it in the hardware capability unit is flushed out. Finally register Ba is set to be the value of the tag field of the message block.

MESSAGES n (d31-16) message channel capability specifier.

ba(d31-0) := the count of the number of messages waiting upon the message channel.

CHAPTER TWELVE.

ORGANISATION OF THE KERNEL MICROPROGRAM.

12.1. Initialisation.

The kernel is loaded into microstore from disc by a bootstrap microprogram that is input by the intimate paper tape reader. The bootstrap, known as DISCBOOT, also clears the stores and loads the initial system code ready for running. Before control is transferred to the kernel, DISCBOOT will reset the state of the processor and clear any error indicators to put the machine into a tidy, standard initial condition. The communication link with the Modular One computer is restarted to prevent any incompleted operations from the last run interfering with the freshly loaded system.

The DISCBOOT utility is shared by all CAP microprograms, so it is up to the kernel to set up its own starting environment. It is passed a number of parameters, extracted by the bootstrap from the user system memory image on disc and these are used to tell the kernel the absolute location of the map and the map offset of the process object for the initial process. The kernel can then set up in the capability unit those capabilities necessary to start the user system. The kernel initialisation consists of of about 250 microinstructions.

12.2. Basic Instruction Set.

Most basic user instructions are implemented entirely by the stage one sequence and a single micro-order in the Function Memory (Section 8.2), but the more involved basic orders, particularly those for floating point operations, require additional space in microstore where they account for some 650 words. This part of the CAP-III kernel is more or less identical to its counterpart in the CAP-I microprogram so that the two basic instructions sets will be compatible this is useful when the transfer of programs between the two systems is concerned.

12.3. Peripheral Control.

The microprogram for interpreting peripheral instructions and organising device transfers is also similar in both systems for compatibility. This part of the microprogram is quite complex. The CAP microprogram can read and write to the Modular One's memory and can also accept interrupts across the link. To start a transfer, the kernel will write a transfer request to the Modular One's store to attract the attention of the other machine. The link program in the Modular One will then read the request, allocate buffer space and, for an input transfer, start to accept data from the appropriate device. When this stage is completed, the Modular One will interrupt the CAP to signal that the transfer is ready and the CAP can then carry out a fast store-to-store copy to effect the transfer. To finish off, the CAP will indicate to the Modular One that the transfer has been done and the Modular One will reclaim the buffer, which if it was for an output transfer, will result in its being transmitted to the appropriate peripheral device. Most of the code in the kernel I/O system is concerned with administering the protocol between the two machines and optimising efficiency by slaving the parameters for the current device in microstore to reduce the number of interactions between the machines. The slaving persists until a device order is encountered for a different device in which case the parameters of the old device have to be flushed out and those of the new device picked up. This part of the kernel is nearly 600 words long.

12.4. Interrupt Handling.

External interrupts and internal processor parity errors cause the microprogram to be diverted to an interrupt location in microstore whenever control returns to stage one. Parity errors are usually catastrophic and result in the kernel giving a diagnostic print-out on the intimate teletype before resigning. In the case of an external interrupt, the interrupt service routine will inspect the state of the system; if control is in an ordinary process, an entry is made directly to the Interrupt Process by disabling the hardware capability registers for the current process and restoring those of the Interrupt Process,

after dumping and restoring the processor registers from the appropriate process bases. On the other hand, if the Interrupt Process is already running, then its wake up waiting flag is set unless it was in the dormant WAIT 0 state, in which case the it resumes immediately.

An interrupt is generated every time a twelve bit count of the number of executed microinstructions overflows. If control is in an ordinary process when this signal arrives, the time-slice word in its process base is incremented by one and if the resulting value is zero, an enforced entry is made to the Interrupt Process to signify that a time-slice is exhausted. In practice, one thousand of these counts are equivalent to about one second of real time although the exact relationship obviously depends on the mix of instructions executed.

In the last chapter, there was a description of the behaviour of the kernel in response to faults. In most cases the code to signal faults is called if the kernel finds an unreasonable argument or request, but some faults are reported in direct response to microprogram traps generated by protection violations and arithmetic overflows.

After a diagnostic print-out on the intimate teletype caused by a fault in the kernel or a parity error, it is possible to resume execution by typing the character 'C' (for continue) or to jump to a fault entry point in the current process by typing the character 'J'. The kernel can be stopped temporarily by typing an 'X-on' character, which will produce the standard diagnostics and then the kernel can be resumed by typing the character 'C' as before. This particular facility is useful for conducting performance measurements as part of the diagnostics include the value of a counter that will record hardware statistics selected by control switches on the processor. The microcode concerned with processing interrupts and reporting faults comes to approximately 350 words.

12.5. Use Of The Capability Unit.

The structure of the capability unit was outlined in Section 8.3. Capability register 0 is always used to hold a descriptor for the map, and it is set up from the parameters passed over by DISCBOOT at initialisation time. Whenever the kernel wishes to inspect or modify the contents of the map, it selects register 0 in last mode and uses relative offsets within the map to address store so that any map pointers going outside the bounds of the map will result in a protection violation.

The most important field in each capability as held in the capability unit, from the point of view of the following discussion, is the tag field which consists of two sub-fields as described earlier. One holds the number of the capability register for the capability table from which the register in question was loaded, while the other normally gives the capability offset of the loaded capability in its parent segment. For example, a register for a capability table will have a tag that contains the register number of the domain descriptor containing the capability and its table number, the latter being the offset of the table capability in the domain descriptor.

A process is represented in the capability unit by a register whose tag is derived differently: the parent register sub-field is zero (pointing to the map register) and the offset sub-field is the least significant eight bits of the map index of the process's entry in the map. The full index is kept in the size field of the register and, because process capability registers are only used to fix a process's capabilities in the capability unit, the access field is zeroed to prevent any access to store through the register. Thus, when the kernel switches context between processes, it can scan the capability unit in normal mode with a tag based on the hashed map index of the target process in order to find the process's capability register. If a normal mode match occurs, the kernel checks that the value of the size field of the located register matches the map index of the process to be found. If this check succeeds, the register is accepted, otherwise a free register is found and allocated to the process. It can be expected that only a few searches of the unit for processes will

fail because of one process present in the unit having the same hash key as another that is found first by the normal mode search, leading the kernel to think that the first process is absent from the unit.

In the CAP-I memory protection system, processes are held differently in the unit: a process is represented by a capability for the PRL of the process with a tag which matches the address of the PRL segment in the process's coordinator's address space. In the CAP-III kernel, this contrivance cannot be used because there is no hierarchy of address spaces. However it must be possible to pick up a process if it is already in the capability unit because otherwise, the cost of evaluating the structure of a process is high, and the wasteful reloading of capabilities would reduce the efficiency of the message system by a considerable degree.

The process capability register is the root of a tree of registers which controls access to the domain of protection contained within the process as shown in Figure 12.5-1. The domain descriptor capability register is a direct descendent of the process register and has the tag <process register, 0>. This register is used to read capabilities for the process base and capability tables. The former would be found in a register with the tag <domain descriptor register, 16> and the latter in registers with tags <domain descriptor register, n> where 'n' is the table number in the range 0 to 15. The individual capabilities in a capability table are tagged thus <table register, capability index> as descendents of the tables from which they are evaluated.

The TGM is set to hold the capability register numbers of the entries in the unit for the capability tables of the currently running process. In Section 8.3 it was stated that the virtual address translation mechanism will translate the capability specifier of an address into the tag <contents of TGM indexed by table number in address, capability index in address> before scanning the capability unit, therefore only capabilities belonging to the process whose tables are set up in the TGM will be found and other capabilities in the unit remain inaccessible until the TGM is reloaded.

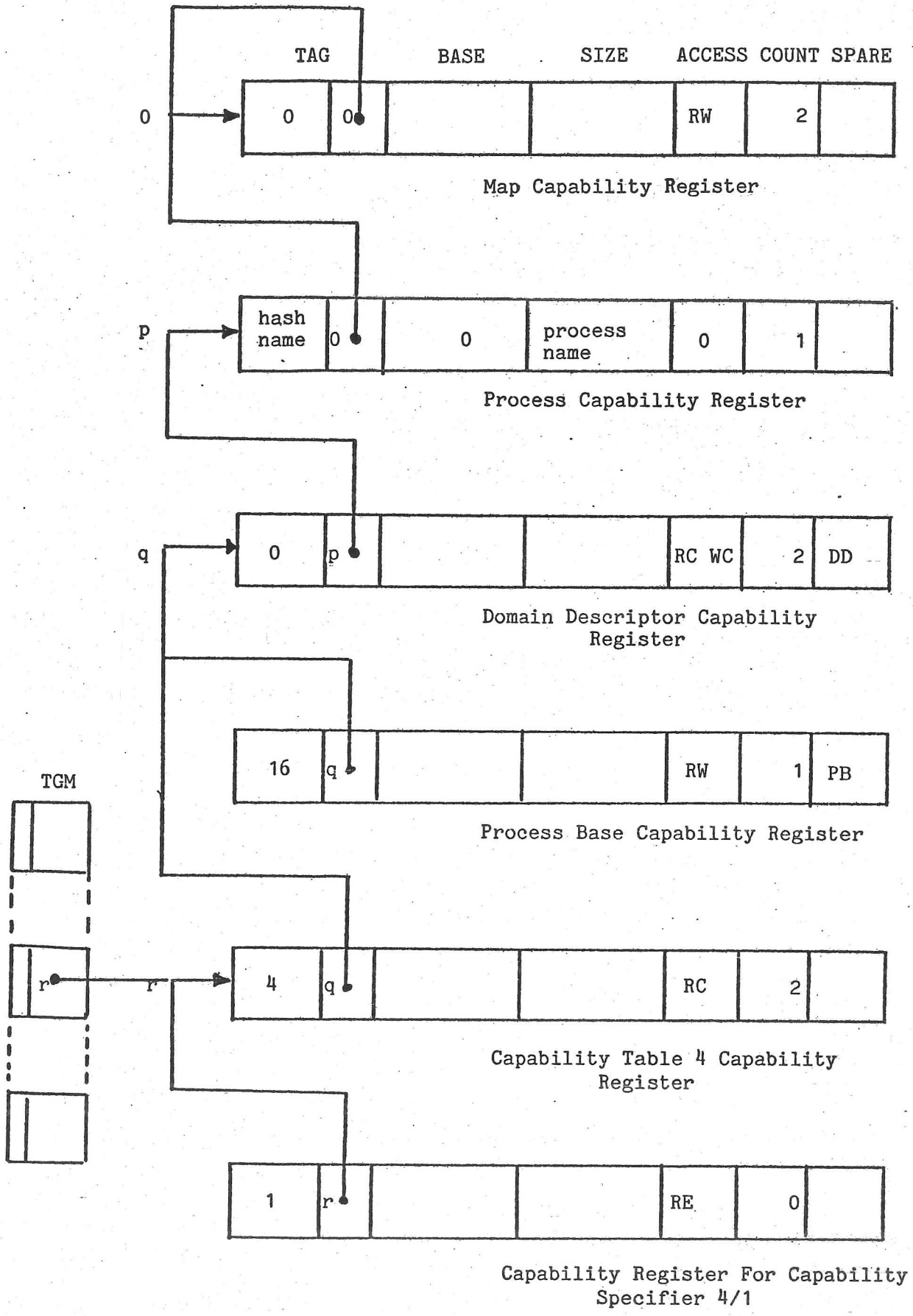


Figure 12.5-1 Use Of The Capability Unit

12.6. Reference Counts in The Capability Unit.

The tree structure of capabilities in the capability unit is held in place by the use of reference counts kept in the count field of capability registers and these counts record the number of pointers, in tag fields and microprogram registers, to the register in question. Thus the count of a register for an ordinary capability will be zero and will contribute one towards the count of the register for the table from which it was loaded. That in turn adds one to the count of the register for its domain descriptor and this keeps the count of the process capability register at one.

When the kernel wishes to find a free register for allocation to a new capability, it scans the unit, starting at the point at which a normal mode search based on the tag of the new capability would begin, looking for the first register with a zero count which can then be cleared and set up with the value of the new capability. The action of clearing the register includes decrementing the count of the parent register which, if its count falls to zero, may become a candidate for re-allocation later on.

The mechanics of the interrupt handling and enforced Interrupt Process entry on faults and hold-ups requires that the kernel locks down the process base and domain descriptor of the Interrupt Process and the current ordinary process (if there is one) so that process registers can be dumped and restored during a context switch. This is done by keeping the reference counts of these objects one higher than the actual number of references to them; this will fix them in the capability unit. When a switch is made between two ordinary processes, the kernel restores the reference counts of the process being left so that its registers can be reclaimed if necessary.

If a capability in store is overwritten, the kernel must also invalidate its entry in the capability unit and this is done by setting a value which has the significance of 'unset' in the access field of the register corresponding to the capability so that any attempt to address memory through it will fail with an access violation. Furthermore, if the register has a non-zero

count, say because it is a domain descriptor or a capability table, all of the capabilities in the tree descending from the register must also be marked unset.

The kernel only keeps the following hardware types in the capability unit:- segments, messages and channels; a discourse on organisation of the latter two types is deferred until later on.

12.7. The Reset Cycle.

It is now possible to describe the kernel '<u>reset cycle</u>' which loads a process and its capabilities into the capability unit. Before switching to a new process, the kernel allocates a process capability register, a domain descriptor register and a process base register, but only the first of these is initialised and the other two are left unset. During the context switch, the kernel will attempt to address the process base of the target process in last mode to load the processor's arithmetic registers, but this will fail with an access violation because the process base register is unset and it will result in an enforced trap to microstore address eighteen which activates the protection violation routine called TRAP18.

The TRAP18 code will inspect the failing register and discover that it is unset and is a process base (this can be deduced from the tag and from marker bits kept in the spare field). TRAP18 will then go to the reset cycle code for evaluating the process base capability, which will attempt to read store using the domain descriptor register and suffer another TRAP18 call because this register is also unset. The reset cycle code is then entered again to set up the domain descriptor by reading the map entry whose index is held in the process capability register. As the map capability register is never unset, this reset cycle can complete and once the domain descriptor has been loaded, the context switch that fell foul of the unset capabilities can be restarted.

Once again, TRAP18 will be called because of the the process base register which is still unset, but the reset cycle will be able to read the domain descriptor this time, with the result that the process base register will be set up and the context switch

can be restarted once again.

Now the process base of the new process can be read successfully and used to prime the arithmetic registers. The final act of a context switch is to set all of the sixteen TGM registers to zero; this makes the capabilities of the previous process inaccessible to the virtual address translation machinery.

However, for the time being, there are no capabilities available for the new process and an attempt to obey an instruction within it will cause a trap to location seventeen in microstore to signify that a capability for the process's current code segment was not found. The TRAP17 code found at this address initially looks at the TGM register indexed by the capability table part of the failing address. If it is zero, the table has not yet been set up and the kernel scans the capability unit with a tag consisting of the domain descriptor register and the table number to see if the table is present and if so, sets the TGM register to point at it. If a register for the table is not already set up in the unit, the kernel will allocate a spare register and mark it unset. The failing instruction in the new process will be retried and if the capability is still not found when the TGM is set up, there will be a second call of TRAP17 which will result in the allocation of a capability register for the missing capability.

Once the allocation is complete, the kernel will suffer a TRAP18 call because the register is unset, and while trying to read it from the parent capability table, there will be another TRAP18 as that register is also unset. These traps are resolved by two reset cycles in a manner analogous to the evaluation of the process base capability described earlier.

The sequence of events outlined above is the full reset cycle that is necessary if all of the capabilities of a process are absent from the capability unit. In practice, much of a process's apparatus will be found in the unit and in consequence, it may only be necessary to carry out a few, simple, partial reset cycles before a process can start running. As the process executes, there are likely to be several more TRAP17 and TRAP18 calls to

recover capabilities that have had their registers re-allocated since the process last ran, or for capabilities that have not been used before.

The reset cycle is written to restart user instructions either after allocating or setting up a single capability register because it is a potentially recursive task to initialise a capability register. The microprogram instruction set is not well suited to this, so an iterative approach is adopted instead.

The segment reset cycle, which is triggered by the TRAP18 routines to set up a capability register, will read a capability from the appropriate parent capability segment, using the parent capability register in last mode and then index the map through register 0 to locate the segment's representation, taking account of any intervening revokers to modify the access code. The access code is checked to be either entirely data or entirely capability type and is copied to the access field of the register. The base and size fields are set up from the results of the refinement calculation using data in both the capability and the map entry as described in Section 9.3.

The kernel uses the spare field of a capability register to contain marker bits that will be set if the register is used for a domain descriptor, a process base, a message or a channel. By inspecting these bits and the values of tag fields, it is possible for the TRAP18 routine to decide what sort of capability register has been violated and to set about loading it from the appropriate parent segment. There is another marker bit called 'active' that is used to flag the process bases and domain descriptors of the Interrupt Process and the currently running process (if there is one). The kernel routine for unsetting a capability register will complain if it is applied to a register marked 'active' and this prevents locked down registers from being flushed by REVOKE or the ALTER orders.

Each stage of a reset cycle involves at least five store cycles; two to read a capability, three to read a map entry and two extra store cycles for every revoker object that intervenes in the path between the capability and the corresponding root object

in the map. However, the size of the unit is such that a capability will remain in existence for a reasonable period of time before its register is reclaimed for a different capability. The kernel takes steps to avoid gratuitously throwing away useful capabilities so that the cost of evaluating a capability is spread out over many instructions by the slaving properties of the capability unit.

12.8. Messages and Channels.

As well as the segments describing the structure of a process and segments of data or program code, the kernel also keeps the segments sealed within message and channel objects in the capability unit. These latter objects are not set up in response to reset cycle faults; instead, the message system code will do a dummy normal mode scan of the capability unit to find a register with a tag that matches the address of the channel or message in question and will check to see if its access field is set. If the access field is not set, the kernel will evaluate the appropriate capability for the channel or message object from which a segment capability will be taken and used to set up the register. Only when the register is set up does the message system try to access data through it to avoid the need for the ordinary reset cycle becoming involved with messages and channels.

The message pool of a process is found in a capability register with the tag <domain descriptor register, 17>. This register is only set up by explicit loading during the MAKEBLOK order which always checks for the existence of a correctly configured register before addressing the message pool segment.

The objects used by the message system are cached in the capability unit to optimise the performance of the inter-process communication operations because it may be expected that some of a process's capabilities for these objects will persist in the unit between message transactions and will be readily at hand the next time a message is to be processed.

12.9. Segment Usage Bits.

The previous description of the actions of TRAP18 was, in fact, rather simplified by ignoring the control of the segment usage bits to try to avoid obscurity. When a segment capability is loaded into the capability unit, the reset cycle inspects the usage bits in the segment's map entry. If the segment is marked as both 'used' and 'dirty', then it is set up normally; otherwise, the calculated access code is copied to a location of microstore associated with register to be updated and, if the segment is neither 'used' nor 'dirty', the access field of the register is set to 'no access'. If the object is marked 'used' but not 'dirty', only the read type bits of the access code are set in the access field. On an access violation, the TRAP18 routine inspects the access field of the failing register and if it is unset, goes directly to the reset cycle to evaluate the capability. Otherwise the faulted access request is compared to the full access code held in the microstore record corresponding to the failing register and if this code is less potent than the request, an access violation fault is reported. In the case when the full access code is suitable and the request is of read type, the 'used' status bit is set in the segment's map entry and the read type access bits of the full code are moved to the capability register access field. If the request was of write or read-and-write type, both the 'used' and the 'dirty' bits are set in the map entry and the full access code is moved to the access field of the register in its entirety. After one of these pseudo TRAP18 entries, the current instruction is restarted and, since an access code has been loaded into the capability unit, the memory cycle will succeed.

The body of microprogram devoted to evaluating segment capabilities, capability unit management and the segment reset cycle is nearly 400 microinstructions in length.

12.10. Protection Orders.

There are a number of utility routines for use by the protection functions of the kernel, the most important of which is a reset cycle for non-hardware objects which are not kept in the

capability unit. This reset cycle will take the address of a capability and evaluate it to find the root object named by the capability. The routines for reading and writing capabilities in capability tables address the capability unit in last mode and jump into the segment reset cycle code if the capability table is not set up in the capability unit. When a capability is over-written, the capability unit is searched in normal mode with the address of the modified capability so that any machine register can be cleared by marking its access code as unset. Other utility routines deal with map slot reference counts and the chain of free map entries.

The REVOKE order (Section 10.4) must unset any capabilities in the capability unit that were derived from paths that go through a modified revoker map slot. This is done by keeping a table which, for each capability register, records the map slot number of the entry from which the register was set up. The REVOKE instruction will determine the root map slot of the path from the revocable capability currently being exercised and will scan the table of map slots, unsetting the access code for all of the registers for which the table entry matches the root slot number. If one of the registers that is unset has any descendents, then they are also unset. The flush carried out by REVOKE may be more violent than is strictly necessary; it will remove capabilities for the root object that did not include the changed revoker in their evaluation, as well as those that did. This could be avoided by keeping an additional table to record the length of the path between the capability and the root object, but it is debateable whether or not the saving on spurious capability evaluations would be adequate compensation for the extra code required in the microprogram. The routines for unsetting all instances of an object in the capability unit is also used by the ALTERC and ALTERD instructions together with the message system when it clears message objects but in these cases, the slot to flush out is the map slot that has been modified and not a slot shielded by a revoker.

Objects that are not hardware types are not kept in the capability unit because its registers are unsuitable for

representing objects other than segments. In any case, it may be reasonably assumed that there will be many more references to segments than to non-hardware type objects so that the overhead of having to evaluate the latter every time should not prove too demanding.

The distribution of code in the protection orders is as follows: utility routines, 200 words; type-extension and revocation, 250 words; capability transfer, 100 words; information orders, 25 words; message system and process switching, 1000 words. There is remarkably little to comment about in the implementation of the protection orders save that they conform to the design rules that were laid down in Chapter Two concerning the complete validation of arguments before modifying capabilities or the map so that a call to the kernel with dubious parameters cannot cause harm or confusion.

No accurate performance analysis analogous to Cook's work for the memory protection system has yet been carried out for the CAP-III kernel, but a number of ad hoc measurements made during the testing phase of the kernel development suggest that instructions like MOVECAP and SEAL, that carry out fairly straightforward operations on capabilities and map entries, take between ten and fifty times as many machine cycles as basic instructions, whereas for the message system, the ratio is nearer several hundreds to one. Compared to the cost of an ENTER/RETURN sequence in the memory protection system this figure is a little disappointing, but on the other hand, it compares favourably with the cost of message transactions in the older system that consume many thousands of machine cycles. It is to be expected that the benefits gained here will more than adequately compensate for the disparity with the ENTER and RETURN operations.

CHAPTER THIRTEEN.

REVIEW AND EVALUATION.

13.1. Some Attributes of the Kernel.

The primary motivation for the design and construction of the CAP-III kernel was to reduce the overheads of a powerful protection system by the use of microprogramming, and to a considerable degree this has been successful. The functions of the kernel are simple, which means that they are implemented by concise and straightforward sequences of code. Coupled with the hardware optimisations afforded by the hardware capability unit and instruction decoding logic of the CAP, this leads to fast and efficient operation. Most important, the memory protection facilities, which might reasonably be expected to be the most heavily used part of the kernel are very cheap, mainly because of the slaving properties of the hardware capability unit. The microprogramming of the inter-process communication system makes it considerably faster than any software implemented scheme and enables us to construct protection domains that can be made very short and simple-minded without fear of suffering unduly from the overheads of frequent domain changes.

Protection in the CAP-III kernel is fine-grained, which is to say that it is possible to protect many items of differing complexity, ranging from a segment that is only a few words in length up to filing system directories and other such highly structured, multi-component objects. The ability to be able to select and protect just part of an item is crucial to the attainment of minimum privilege.

The slaving of capabilities in the hardware capability unit allows a domain to have a working set of many capabilities. There is less of a psychological limit on the number of capabilities which it is reasonable to exercise than there would be in a system with only a few explicitly addressed capability registers; this encourages the full separation of the privileges of a domain.

The kernel is entirely self-contained and places no reliance on the integrity of the software system running on the machine, which greatly contributes towards its ruggedness. The kernel always addresses data structures, including the map and capability segments, through the hardware capability unit so that any invalid addresses supplied by users or resulting from incorrect calculations by the kernel will be trapped by the hardware protection mechanisms. It has been noted earlier that the kernel always tidies up after reporting a fault so that malfunctioning programs cannot expose any loopholes in the protection system.

The uniformity and simplicity of the protection mechanisms for both primitive hardware objects and user defined protected objects offers not only an economy of code inside the kernel but also minimises the amount that has to be learned by users, who will be more likely to exploit to the full mechanisms that they can easily comprehend.

13.2. An Evaluation of the Kernel.

The global naming scheme of the kernel is conceptually more simple than the nested address space structure of the memory protection system of CAP-I and involves less memory being tied down in protection data structures. In the memory protection system, the route between a capability in a user process and the representational information for the object it protects is eight store cycles long: two each in a capability table, the Process Resource List, a coordinator capability table and the Master Resource List. In the CAP-III system, only five store cycles are required: two in a capability table and three in the map. It is reasonable to expect that the number of protected objects in the CAP-III map will be close to the number of segments, protected procedures and software capabilities in the CAP-I operating system, which means that the resident memory requirements of the map for the kernel will match those of the MRLs and PRLs in the older system.

Short names were adopted in the kernel because in the earlier stages of design and implementation, it was found that hashing long unique names was a costly operation both in terms of the code

required in the microprogram and the time taken to find objects in the map. There were also difficulties in trying to provide mechanisms for keeping the microprogram image of a global name space in step with the permanent record on backing store. What has been lost by the kernel is the ability to allow users to keep capabilities in a filing system without translating names. It may be noted that in the CAP-I system it is necessary to carry out such a translation and the efficiency of the operating system has not suffered noticeably, so, it may be hoped that the same will apply to CAP-III. There are one or two complications involved in filing active capabilities for extended objects because the operating system will have to ensure that the structure of a filed extended object is duplicated in the kernel map and its own data structures. Given the simple organisation of objects and the centralisation of information about them in the map it should be straightfoward for the software to cope with this problem.

It is much easier for the kernel to carry out automatic garbage collection, because of the central collection of all information about objects in the single global map, than it is for the memory protection system to garbage collect Process Resource Lists. In CAP-I, each process has to have its own garbage collector for the process's resource list, and as there are no microprogram maintained reference counts, the garbage collectors will be activated frequently with a corresponding degradation of the amount of useful computation carried out. The CAP-III kernel only requires that the system-wide software garbage collector runs to flush out complex looped structures (that it is safe to assume will only arise occasionally) so that the garbage collector need only scan the system at low frequency and this imposes very little in the way of overheads. On the other hand, there is the expense of maintaining reference counts, but as this activity is carried out by the kernel microprogram the cost should be small.

The partitioning approach for segregating capabilities from data is forced upon the kernel mainly by the hardware of the CAP machine, but there is also an interesting dependency between tagged capabilities and forever-unique names that would make tagging unsuitable for short unique naming schemes. In Section

4.1, it was noted that in a tagged architecture, it is impossible to keep track of capabilities because they can be scattered arbitrarily throughout memory. It is therefore necessary to employ hashed forever-unique names so that names will always be valid, because the system cannot tell when the names are no longer required. From this observation the applicability of Fabry's capability-based addressing scheme must be restricted to forever-unique name systems because of its need for tagged capabilities.

The type-extension facilities of the kernel are very basic, but they have the expressive power of more elaborate mechanisms such as those found in HYDRA. The sealing and unsealing operations provide the essential machinery for concealing the representation of an object from its users, while at the same time permitting a duly authorised type manager to get inside any objects that it controls. Because of their simplicity, the kernel primitives are very fast, but on the other hand, the checking functions of the slower HYDRA amplification template scheme must be carried out by the software of a type-manager. It is likely that software can perform the verification of arguments as quickly as a general purpose template system because the verification code can be specific to the particular interface in question and can have knowledge of the expected arguments built in, rather than having to be able to cope with all possible requirements.

The introduction of basic objects and data sealing to the extended object and capability sealing mechanisms proposed by Redell [74] has lead to a uniform set of operations for creating objects, manipulating them and interrogating their representations. This saves a lot of space in the kernel microprogram because the type extension facilities can be used internally by the kernel in the processing of hardware objects. The revocation features of the kernel integrate well with the type-extension scheme and the other protection operations. Revocation can be temporary or partial, although by adopting a dependent capability system, the ability to make the privilege of revocation itself revocable has been sacrificed. It is unlikely that this latter facility would be heavily used and caretaker

protection domains can be used if the facility is required.

The unified process and protection domain structure is successful mainly because of the versatile microprogrammed message system which can be applied equally well to simple procedure-like interfaces and to more complex ones involving processes that handle many messages at one time. There is little problem of resource control for message blocks in the CAP-III system because it supports many message pools that need not be locked down in memory, and the involvement of the kernel in the scheduling of processes contributes towards the speed of message transactions by cutting down on the number of times that the operating system scheduler has to be invoked.

13.3. Relationship with the CAP-I Memory Protection System.

One of the original motivations for the research described in this thesis was to extend the facilities of the CAP-I memory protection system and to investigate some alternative strategies for implementing protection mechanisms, so it is informative to emphasise briefly the differences between the two systems.

Experience would suggest that using short term global names to reference objects in a central resident table is at least as efficient as the nested resource list approach and is conceptually much simpler. However, global names do require a closer microprogram involvement with garbage collection because space in the map is a scarce resource across the entire system, whereas slots in a resource list are only drawn upon by a single process, so that garbage collection can be done by a program running synchronously in the process without any drastic effect on the system as a whole. Earlier in this chapter it was noted that, because of global naming, capability evaluation in the CAP-I kernel requires fewer store cycles and so the overheads of protection are correspondingly less expensive.

In the CAP-I system the type-extension and revocation facilities of the CAP-III kernel can only be modelled by the use of protection domains. Thus it might be expected that the protection of abstract objects to be sharper and more economical in CAP-III. Also, because the kernel has a uniform mechanism for

describing all types of objects and controlling access to them in a system built around the kernel, there will be less requirement for software to mimic the basic access control, type checking and naming functions of the microprogram.

The total replacement of the hierarchical process and non-hierarchical domain structure of CAP-I by a non-hierarchic single-domain process architecture within a microprogrammed message system greatly reduces the amount of operating system code required to support multiprogramming and it provides a uniform set of communication primitives for all modules of the system. It is hoped that the lower cost of message transactions will increase the speed of the system and that, despite the slight slowness of the CAP-III message system compared to the CAP-I protected procedure call machinery, users will have no qualms about implementing complex tasks as a set of inter-communicating processes in the new system, in the same way that they would use protected procedures in the older design.

It is reasonable to claim that the kernel provides a complete, powerful and efficient protection kernel around which it will be possible to build sturdy and trustworthy operating systems.

REFERENCES.

Anderson 73
"Information Security in a Multi-User Computer Environment", Anderson J., Advances In Computers, Vol. 12, Academic Press, New York, 1973, pp1-35.

Birrell 78
"Systems Programming In A High Level Language", Birrell A., Ph.D. Thesis, University of Cambridge, 1978.

Birrell and Needham 78
"An Asynchronous Garbage Collector For The CAP Filing System", Birrell A. and Needham R., Operating Systems Review, Vol. 12, No. 2, Apr. 1978, pp31-33.

Bourne et al. 74
"Algol68C Reference Manual", Bourne S., Birrell A. and Walker I., University of Cambridge, 1974.

Branstad 73
"Privacy and Protection In Operating Systems", Branstad D., Computer, Vol. 6, Jan. 1973, pp43-49.

Burroughs 61
"The Descriptor -- a Definition Of The B5000 Information Processing System", Burroughs Corp., Detroit, Michigan, 1961.

Cohen and Jefferson 75
"Protection In The HYDRA Operating System", Cohen E. and Jefferson D., Proceedings Fifth Symposium On Operating Systems Principles, Operating Systems Review, Vol. 9, No. 5, Nov. 75, pp141-160.

Cook 78
"An Evaluation Of A Protection System", Cook D, Ph.D. Thesis, University of Cambridge, 1978.

Denning 70
"Virtual Memory", Denning P., ACM Computing Surveys, Vol. 2, No. 3, 1970, p135.

Denning et al. 74
"Selectively Confined Sub-systems", Denning D., Denning P. and Graham G., IRIA - International Workshop on Protection In Operating Systems, 1974, pp 55-62.

Dennis 65
"Segmentation And The Design Of Multiprogrammed Computer Systems", Dennis J., Journal of the ACM, Vol. 12, No. 4, Oct. 65, pp589-602.

Dennis and Van Horn 66
"Programming Semantics For Multiprogrammed Computations", Dennis J. and Van Horn E., Communications of the ACM, Vol. 9, No. 3, Mar. 1966, pp143-155.

England 74
"Capability Concept Mechanism And Structure In System 250", England D, IRIA - International Workshop on Protection In Operating Systems, 1974, pp 63-82.

Fabry 68
"Preliminary Description Of A Supervisor For A Machine Oriented Around Capabilities", ICR Quarterly Report, No. 18, ICR University of Chicago, Aug. 1974.

Fabry 74
"Capability-Based Addressing", Fabry R., Communications of the ACM, Vol. 17, No. 7, Jul. 74, pp403-412.

Fenton 74
"Memoryless Sub-systems", Fenton J., Computer Journal, Vol. 17, No. 2, 1972, pp143-147.

Feustel 73
"The Advantages Of a Tagged Computer Architecture", Feustel E., IEEE Transactions on Computers, Vol. C-22, No. 7, Jul. 1973, pp644-656.

GEC 72
"4080 Computer Technical Description", GEC Computers Ltd., England. 1972

Graham 68
"Protection In An Information Processing Utility", Graham R., Communications of the ACM, Vol. 11, May 1968, pp365-369.

Herbert 78
"CAP Hardware Manual", Ed. Herbert A., Computer Laboratory, University Of Cambridge, 1978.

Hoare 74
"Monitors: An Operating System Structuring Construct", Hoare C., Communications of the ACM, Vol. 17, No. 10, Oct. 74, pp549-557.

Hoffman 73
"Security and Privacy In Computer Systems", Hoffman L., Ed., Melville Publishing Co., Los Angeles, 1973.

Jones 73
"Protection In Programmed Systems" Jones A., Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1973.

Lampson 69
"Dynamic Protection Structures", Lampson B., FJCC AFIPS Conference Proceedings, Vol. 35, 1969, pp27-38.

Lampson 71
"Protection", Lampson B., Proceedings Fifth Princeton Symposium On Information Sciences And Systems, Princeton University 1971, pp437-443. (Reprinted in Operating Systems Review, Vol. 8, No. 1, Jan 74, pp18-24).

Lampson 73
"A Note On The Confinement Problem", Lampson B., Communications of the ACM, Vol. 16, No. 10, Oct. 1973, pp613-615.

Lampson and Sturgis 76
"Reflections On An Operating System Design", Lampson B. and Sturgis H., Communications of the ACM, Vol. 19, No. 5, May 1976, pp251-265.

Lauer 74
"Protection And Hierarchical Addressing Structures", Lauer H., IRIA - International Workshop on Protection In Operating Systems, 1974, pp137-148.

Lauer and Needham 78
"On The Duality Of Operating System Structures", Lauer H. and Needham R., Proceedings of the Second International Symposium On Operating Systems, Oct. 1978.

Levin et al. 75
"Policy/Mechanism Separation In HYDRA", Levin R., Cohen E., Corwin W., Pollack F. and Wulf W., Fifth Symposium On Operating Systems Principles, Operating Systems Review, Vol. 9, No. 5, Nov. 75, pp132-140.

Liskov 76
"An Introduction To CLU", Liskov B., Computation Structures Group Memo 136, Laboratory for Computer Science, MIT, Cambridge, Mass., 1976.

Needham 72
"Protection Systems And Protection Implementations", Needham R., FJCC AFIPS Conference Proceedings, Vol. 41, Part I, 1972, pp571-578.

Needham 77
"The CAP Project - An Interim Evaluation", Needham R., Proceedings Sixth Symposium On Operating Systems Principles, Operating Systems Review, Vol. 11, No. 5, Nov. 77, pp17-22.

Needham and Birrell 77
"The CAP Filing System", Needham R. and Birrell A., Proceedings Sixth Symposium On Operating Systems Principles, Operating Systems Review, Vol. 11, No. 5, Nov. 77, pp11-16.

Needham and Walker 77
"The Cambridge CAP Computer And Its Protection System"., Needham R. and Walker R., Proceedings Sixth Symposium On Operating Systems Principles, Operating Systems Review, Vol. 11, No. 5, Nov. 77, pp1-11.

Needham and Wilkes 74
"Domains Of Protection And The Management Of Processes", Needham R. and Wilkes M., Computer Journal, Vol. 17, No. 2, May 1974, pp117-120.

Neumann et al.
"On The Design Of A Provably Secure Operating System", Neumann P., Fabry R., Levitt K., Robinson L. and Wensley J., IRIA - International Workshop On Protection In Operating Systems, 1974, pp161-176.

Organick 72
"The MULTICS System: An Evaluation Of Its Structure", Organick E., MIT Press, Cambridge, Mass., 1972.

Parnas 72
"On The Critera To Be Used In Decomposing Systems Into Modules", Parnas D., Communications of the ACM, Vol. 15, No. 12, Dec. 72, pp1053-1058.

Redell 74
"Naming And Protection In Extensible Operating Systems", Redell D., Ph.D. Thesis, University Of California, Berkeley, 1974. (Also available as MIT Project MAC Technical Report TR-140).

Richards 69
"BCPL - A Tool For Compiler Writing And Systems Programming", Richards M., AFIPS Conference Proceedings, Vol. 33, 1969, pp 557-566.

Saltzer and Schroeder 75
"The Proctection Of Information In Computer Systems", Saltzer J. and Schroeder M., Proceedings of the IEEE, Vol. 69, No. 9, Sep. 1975, pp1278-1308.

Schroeder 72
"Cooperation Of Mutually Suspicious Subsystems In A Computer Utility", Schroeder M., Ph.D. Thesis, MIT, Cambridge, Mass., 1972. (also available as MIT Project MAC Technical Report TR-104).

Stroustrup 77
"On Unifying Module Interfaces", Stroustrup B., Operating Systems Review, Vol. 12, No. 1, Jan. 78, pp90-98.

Sturgis 73
"A Postmortem For A Time-Sharing System", Sturgis H., Ph.D. Thesis, University of California, Berkeley, 1973. (Also available as Xerox Palo Alto Research Center Technical Report CSL74-1).

U.S. Department Of Health, Education and Welfare 73
"Records, Computers And The Rights Of Citizens", U.S. Department Of Health, Education and Welfare, MIT Press, Cambridge, Mass., 1973.

Van Wijngaarden et al. 76
"Revised Report On The Algorithmic Language Algol68", Edited by Van Wijngaarden A., Mailloux B., Peck J., Koster C., Sintzoff M., Linsey C., Meertens L. and Fisker R., Springer Verlag, Berlin, 1976, p9.

Walker 73
"The Structure Of A Well-Protected Computer", Walker R., Ph.D. Thesis, University of Cambridge 1973.

Ware 67
"Security And Privacy In Computer Systems", SJCC AFIPS Conference Proceedings, Vol. 30, 1967, pp287-203.

Watson 78
Ph.D. Thesis, University of Cambridge, 1978. (In preparation).

Wilkes 51
"The Best Way To Design An Automatic Calculating Machine", Wilkes M., Manchester University Inaugural Conference, Jul. 1951, pp16-18.

Wulf et. al. 74
"HYDRA: The Kernel Of A MultiProcessor Operating System", Wulf W., Cohen E., Corwin W., Jones A., Levin R., Pierson C. and Pollack F., Communications of the ACM, Vol. 17, No. 6, Jun. 74, pp337-345.