OPERATING AND PROGRAMMING SYSTEMS SERIES

PETER J. DENNING, *Editor*

The Cambridge CAP Computer and Its Operating System

M.V. Wilkes
R.M. Needham

THE COMPUTER SCIENCE LIBRARY

The Cambridge CAP Computer and Its Operating System

THE COMPUTER SCIENCE LIBRARY

Operating and Programming Systems Series
PETER J. DENNING, *Editor*

1	Halstead	A Laboratory Manual for Compiler and Operating System Implementation
2	Spirn	Program Behavior: Models and Measurements
3	Halstead	Elements of Software Science
4	Franta	The Process View of Simulation
5	Organick and Hinds	Interpreting Machines
6	Wilkes and Needham	The Cambridge CAP Computer and Its Operating System

OPERATING AND PROGRAMMING SYSTEMS SERIES

6

The Cambridge CAP Computer and Its Operating System

M. V. Wilkes
R. M. Needham
University of Cambridge

NORTH HOLLAND
NEW YORK • OXFORD

Elsevier North Holland, Inc.
52 Vanderbilt Avenue, New York, N.Y. 10017

Distributors outside the United States and Canada:

Thomond Books
(A Division of Elsevier/North-Holland Scientific Publishers, Ltd.)
P.O. Box 85
Limerick, Ireland

Library of Congress Cataloging in Publication Data

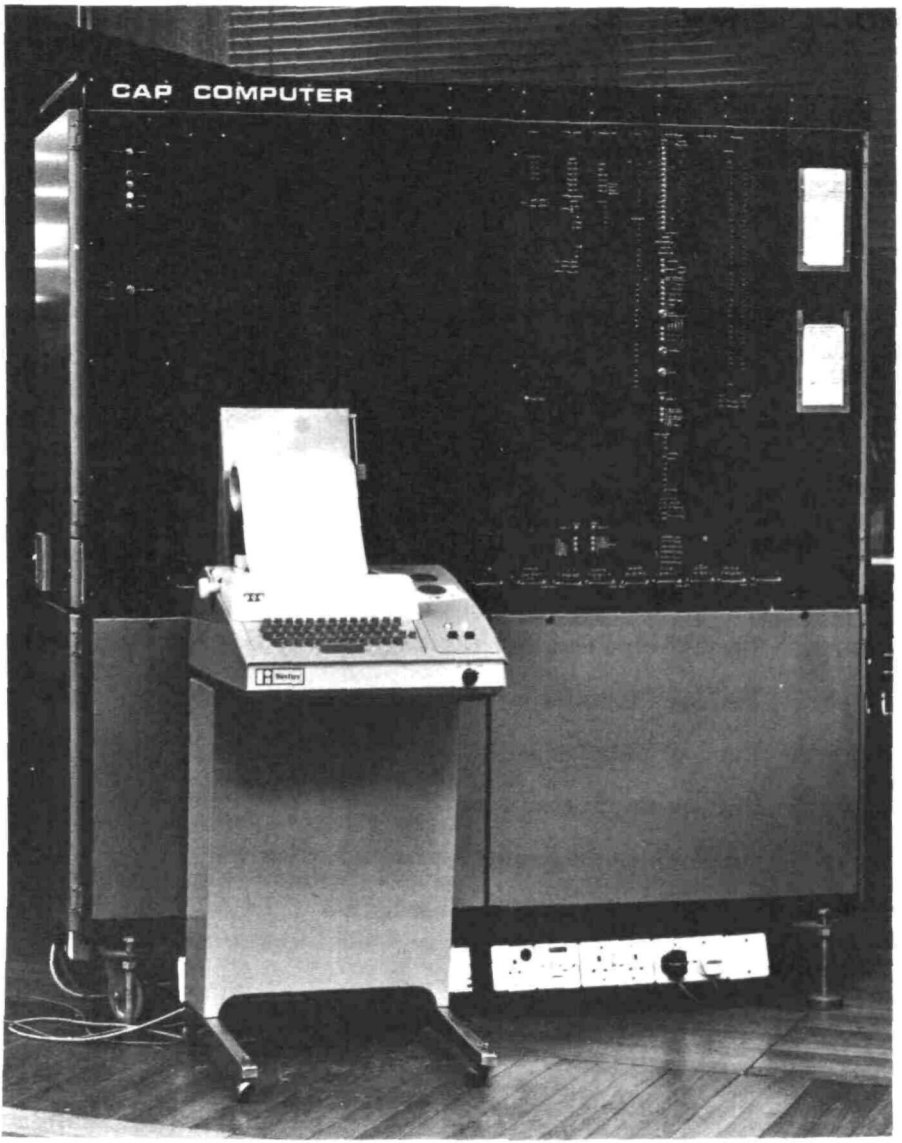
Wilkes, Maurice Vincent.
The Cambridge CAP computer and its operating system.
(Operating and programming systems series) (The Computer science library)

Bibliography: p.
Includes index.
1. CAP (Computer) I. Needham, Roger Michael, joint author. II. Title.
QA76.8.C16W54 001.6'4 79-17858
ISBN 0-444-00357-6
ISBN 0-444-00358-4 pbk.

Manufactured in the United States of America

Contents

Preface		vii
Acknowledgments		ix
Chapter 1	CAP Fundamentals	1
Chapter 2	The CAP Hardware and Microprogram	19
Chapter 3	The CAP Operating System	32
Chapter 4	The CAP Filing System	45
Chapter 5	Discussion and Conclusions	57
Appendix 1	A Hardware-Supported Protection Architecture	75
Appendix 2	ALGOL 68C and Its Run Time System	92
Appendix 3	Specimen Programs	95
References		159
Bibliography		161
Index		163


CAP COMPUTER

Preface

Work started on the CAP project in 1970. The initial planning was done jointly by both authors, but as time went on it was the first author who assumed leadership of the project and was reponsible for most of the major decisions. The origin of the project lay in two visits paid by the second author in June 1968 and August 1969 to the Institute for Computer Research at the University of Chicago. Professor R. Fabry, then a graduate student, was there working under Profesor V. H. Yngve on the design of a computer with hardware-supported capabilities. We felt that this approach to memory protection broke new ground, and that it should be followed up by building a computer on a sufficiently large scale to support a complete operating system. As time went on the decision was taken (by RMN) to base the design on implicitly, rather than explicitly, loaded capability registers. This is perhaps the most distinctive feature of the CAP computer from the hardware point of view.

Professor D. J. Wheeler took on his competent shoulders the responsibility for the design of the CAP hardware. He was ably assisted by Mr. N. W. B. Unwin. Mr. V. Claydon was responsible for the mechanical work, and Mr. D. B. Prince and Mr. P. J. Bennett assisted with the commissioning. It contributed immensely to the success of the project that the design and construction of the CAP computer was carried through to high professional standards and fully met its specification. Professor Wheeler himself took a close interest in the project in all its aspects and we have profited greatly from the continuing discussions we have had with him. The CAP project was an exercise in the design of a coordinated hardware, firmware, and software system. A number of proposals for the protection structure were examined and it was decided to adopt one that had been worked out in some detail by Dr. R. D. H. Walker in his doctoral thesis. Dr. Walker remained in the Laboratory as a member of the Faculty and was responsible for much of the implementation. In particular, he designed and implemented the microprogram and the input–output system.

The choice of a programming language for an operating system is a crucial one. Early in the CAP project two workers in the Laboratory, Dr. S. R. Bourne and Mr. M. J. T. Guy, produced a compiler for a very small subset of ALGOL 68. It

appeared to us that this language had potential merit as a system programming language and we were able to provide support for a small group which, under Dr. Bourne, was eventually reponsible for the ALGOL 68C system. As a system program language ALGOL 68C has proved a distinct success, and we hope that the programs given at the end of the book will be of interest from this point of view, as well as from the point of view of the CAP system itself. Dr. A. D. Birrell, who had worked as a graduate student under Dr. Bourne, was responsible for transferring the ALGOL 68C compiled to the CAP computer and writing the run time system. He went on to make a substantial contribution to the operating system, including the filing system. Many of the major protected procedures in the system were written or specified by him.

R. M. N.
M. V. W.

Cambridge, England
May, 1979

Acknowledgments

We should like to express our thanks to our senior colleagues who have all contributed to the project in some way or another, especially Dr. N. E. Wiseman, Dr. M. Richards, and Dr. J. K. M. Moody. The project owes much to successive generations of graduate students whose work and ideas have gone to make up the system. Their names and those of other workers in the Laboratory who have contributed will be found in the CAP bibliography at the end of the book. We should like to add the name of Dr. J. Hext of the University of Sydney who spent a year with us early in the project. We are also conscious of our debt to many colleagues in other laboratories, especially to Professor R. Fabry (whom we have already mentioned), Professor J. H. Saltzer, Dr. M. D. Schroeder, Dr. B. W. Lampson, and Professor Anita Jones.

We have similarly received much help from our colleagues in the work of preparing this manuscript. Dr. Walker read the draft carefully and saved us from committing a number of errors. We have a special indebtedness to Dr. A. J. Herbert: Not only did he read the manuscript and help us get it into its final form, but he took responsibility for the production of the camera-ready copy for the printer. Without his help there is no doubt the book would have contained many more inaccuracies and imperfections than in fact it does. Mrs. M. K. Foale gave us most valuable assistance in editing the text and in creating the computer version from which the camera-ready copy was produced.

We are indebted to Macdonald and Jane's Publishers Ltd. for permission to reproduce as Figures 1.1 and 1.2 two diagrams from "Time-Sharing Computer Systems" by the second author. We are similarly indebted to the honorary Editor of the *Computer Journal* for permission to reproduce as Figure 5.1 a diagram from a paper entitled "Domains of Protection and the Management of Processes" which appeared under our joint names in Volume 17, page 117 (1974) of that journal. We should like to thank Dr. Herbert for kindly agreeing to our reprinting in full a paper presented by him at an international colloquium held at IRIA, Rocquencourt, in October 1978, and to thank IRIA and the North-Holland Publishing Company in Amsterdam for giving the necessary permission.

Finally, we should like to express our appreciation to Professor Peter J.

Denning of Purdue University, the editor of the series in which this book appears. Professor Denning has taken much interest in the CAP project from the beginning, and it is due to his suggestion and subsequent encouragement that the book has been written.

The design and implementation of the CAP computer and its operating system was a departmental project in the Computer Laboratory of the University of Cambridge. It received very generous financial support from the Science Research Council of the United Kingdom.

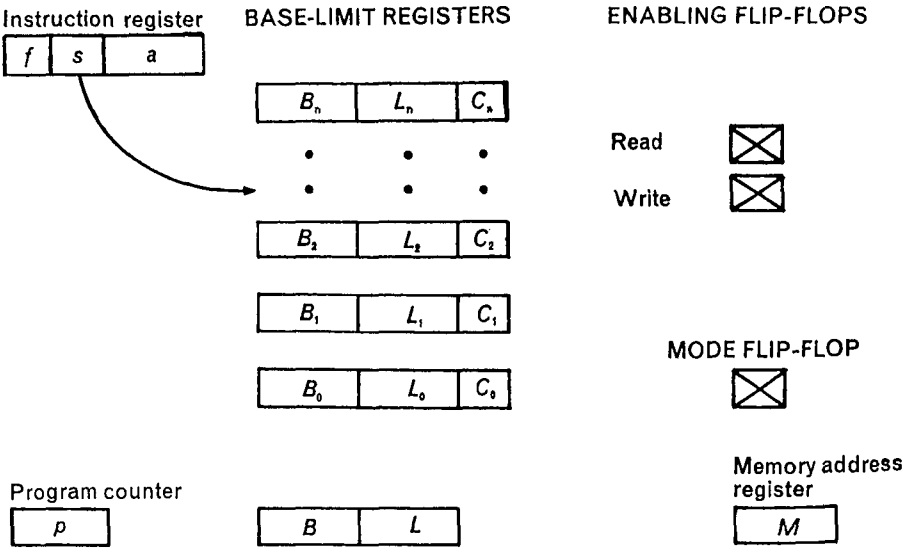
CHAPTER 1

CAP FUNDAMENTALS

The need for hardware memory protection was first met when operating systems which permitted the running of users' programs written in a variety of languages, including assembly language, came into use. One early system, the CTSS, had a separate memory to hold the supervisor and this memory could be accessed only when the computer was in privileged mode. More flexibility in the use of available memory space is required and it is now common for computers to be provided with a number of base-limit registers, each of which, when appropriately loaded, points to and delimits a segment of memory; when the computer is operating in normal mode only those parts of the memory that lie within segments pointed to by base-limit registers are accessible. Associated with each base-limit register are a few extra bits which define the type of access permitted, for example, read-only, read-write, execute-only, etc. If an attempt is made to access memory outside the permitted limits or to make an access of a forbidden type, the hardware switches the computer to privileged mode and sends control to a place in memory determined by the nature of the violation that has occurred. This type of organisation is illustrated in Figure 1.1, taken from Wilkes (1975), where a fuller discussion of such memory protection systems will be found.

In systems like that shown in Figure 1.1, loading of the base-limit registers can only take place when the processor is operating in the privileged mode; since only trusted programs belonging to the operating system are allowed to run in privileged mode, effective control is established over what goes into the base-limit registers and hence over which segments of memory are made accessible at any given time. Systems such as the CAP work on a different principle. There is nothing privileged about the operations of loading the base-limit registers, but the bit patterns that can be put into them are strictly controlled. Only words taken from a set that is closed as far as the user is concerned can be caused to go into the base-limit registers. In a rather similar way one can prevent the unauthorised firing of a gun by either restricting the loading of the gun to trusted personnel, or by mounting a guard on the ammunition. Either of these precautions, if rigorously enforced, is effective by itself and there is no need for both.

The difference between a conventional processor with a privileged mode and one with CAP-like architecture is emphasised by the use of different terminology. In the case of the conventional processor the



Setting of enabling flip-flops

C_s	Read	Write
R	*	
RW	*	*
E		

Execution sequence:
Set $M = B + p$ if $p \leqslant L$; otherwise trap
Fetch instruction into instruction register
Set enabling flip-flops from C_s
Set $M = B_s + a$ if $a \leqslant L_s$; otherwise trap
Execute instruction; trap if execution not enabled
Increment p
Repeat
Trap sets mode flip-flop

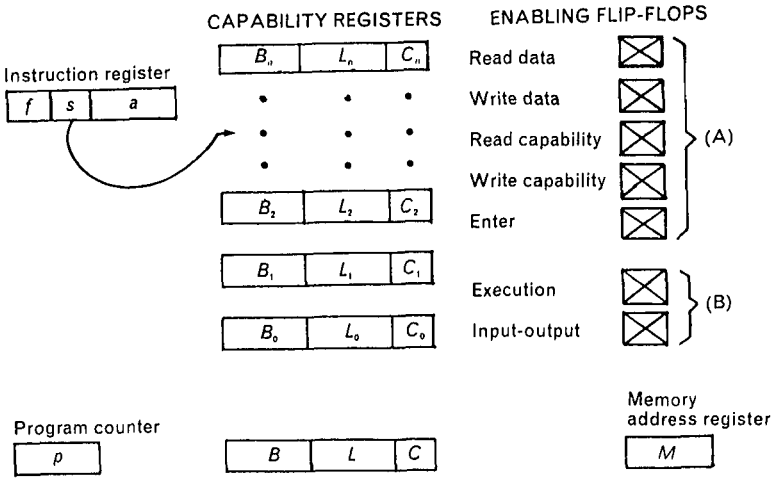
Figure 1.1 A Protection Architecture Using Base-Limit Registers

words that go into base-limit registers are called segment descriptors, whereas in the case of the CAP and similar machines the base-limit registers are renamed capability registers and the words that go into them are called capabilities. A capability contains a base, a limit, and an access code.

Capability Architectures

In a machine with a capability architecture the memory thus contains words of two different types, namely, data words (which term includes instructions) and capabilities. In order to be able to access a segment of memory it is necessary to possess a capability for it. Protection depends on the fact that capabilities cannot be forged. It must not be possible, unless the necessary high degree of privilege is available, for a bit pattern to be generated in an arithmetic register and transferred, via the memory or otherwise, to a capability register. Thus while capabilities may be manufactured by the part of the system whose function it is to do this, they may not be counterfeited elsewhere.

Data words and capabilities are stored in distinct segments and no segment ever contains a mixture of data words and capabilities. There are two different types of capability, namely D-type and C-type. Possession of a D-type capability for a segment enables words to be transferred from that segment to an arithmetic register, or vice versa; possession of a C-type capability enables words to be transferred from the segment to a capability register, or vice versa. In the CAP it also enables certain digits in the word to be transferred from the segment to an arithmetic register, but not the other way round. The former involves no loss of protection, but it does enable the bits in a capability to be inspected. If the only capabilities that were to exist giving access to segments containing capabilities were of C-type and the only ones that were to exist giving access to segments containing data were of D-type, then the system would be intolerably restrictive. Relaxation of the rules is, however, possible in a controlled manner. For example, in the CAP operating system a certain procedure, responsible for performing housekeeping tasks necessary after the occurrence of an interrupt, is given a D-type capability for a segment containing capabilities. Elsewhere that segment is accessed by means of a C-type capability. Thus, although we may speak of capability segments and data segments, we do not imply that capabilities for them are always of a particular type. There must be one very highly trusted system procedure that has both a C-type and a D-type capability for the same segment. This protected procedure may be likened to the Royal Mint, since it may perform legally



Setting of enabling flip-flops

C_s	Read data	Write data	Read capability	Write capability	Enter
R	*				
RW	*	*			
RC			*		
RWC			*	*	
E					
IO					
EN					*

C	Exe-cution	Input-output
R	*	
RW	*	
RC		
RWC		
E	*	
IO	*	*
EN		

Execution sequence:
Set $M = B + p$ if $p < L$; otherwise trap
Fetch instruction
Set enabling flip-flops (A) from C_s
Set enabling flip-flops (B) from C
Set $m = B_s + a$ if $a < L_s$; otherwise trap
Execute instruction; trap if execution not enabled
Increment p
Repeat

Figure 1.2 A Protection Architecture Using Capability Registers

what would elsewhere be an act of forgery; that is, it may manufacture capabilities. Figure 1.2, also taken from Wilkes (1975), shows the type of organisation just described.

Loading of the capability registers may be done explicitly by instructions included for that purpose, very much in the way that index registers are loaded. Plessey System 250 works in this way. In the CAP computer loading of the capability registers takes place implicitly when capabilities are referred to. The capability registers form, in effect, an associative memory that operates in very much the same way as the associative memory used in a paging system.

In the CAP and similar machines the rules governing the manipulation of capabilities are enforced by hardware. In systems of which HYDRA is an example, similar rules are enforced by software, the computer on which the software runs being a conventional one with a privileged mode. The fact that it is possible to implement capability systems in software should not be allowed to obscure the essential difference between computers with conventional hardware and those with capability type hardware; nor should the fact that base-limit registers and capability registers play a similar role in addressing memory.

The Control of Privilege

The conventional system with a privileged mode enables facilities for memory protection to be put at the disposal of the writer of the operating system. Similar facilities cannot, however, be put at the disposal of the designer of a subsystem (for example, a transaction processing subsystem, or a real time subsystem) which will run under the main operating system unless he is allowed to incorporate part of his subsystem within the operating system. Protection systems that have levels or rings of protection enable this difficulty to be overcome, provided that the writer of the operating system does not use all the levels himself, but leaves some to be used by writers of subsystems. In a capability system there is no difficulty in placing at the disposal of the writer of subsystems facilities which are exactly similar to those available to the writer of the main system. Moreover, they can be extended if necessary to the writers of sub-subsystems. Furthermore, there is nothing hierarchical about capabilities, whereas with rings of protection a process has access to everything in its own ring and to rings outside it. It is this non-hierarchical characteristic of capabilities that is perhaps their most distinctive feature.

A hierarchical organisation is appropriate to the management of the flow of control (since there must be some pinnacle of authority) but

it is unnecessarily restrictive in the case of protection. This fact must have become apparent to anyone who has tried to plan a system of locks and keys for a building with a master key, submasters, and so on. The realisation that a hierarchical system of memory protection is inadequate came almost simultaneously to operating system specialists and to programming language specialists. To the former one of the principal attractions of capabilities was that they seemed to lend themselves naturally to a non-hierarchical protection scheme. The latter took up ideas that had already appeared in SIMULA and other languages and generalised them under the name of abstract data types.

Memory protection can be thought of in two ways: in terms of addressing or in terms of lockout. The former is exemplified by the scope rules of a high level language. A programmer has no way of addressing a variable that is out of scope; he may think that he can do so, but the identifiers that he writes under this misapprehension will be invalid and his program will be rejected by the compiler. Similarly, the possession of a capability for an object gives the programmer a means of addressing that object; without the capability he cannot validly even refer to it. On the other hand, in a programming situation in which all variables are global, the programmer may have accurate knowledge of where the various objects are in memory or know valid symbolic addresses for them. However, if he refers to an object in a part of the memory that is locked out, when the relevant instruction comes to be executed the memory protection system will produce a run time trap.

There is a further requirement for an ideal memory protection system that has little to do with the original motivation for providing memory protection. This requirement is that at any time the running procedure should have access only to the data that it requires and to nothing else. Moreover, it should be possible by inspecting the program, or by examining a run time trace, to determine unambiguously which data are in fact accessible. Benefits flow from this both at the time when the operating system is being developed and when it is being run in service. During development a large class of program errors, namely those which lead to a procedure attempting to access data segments which it should not access, can be readily and surely located. When the system is put into service any residual bugs can similarly be found and dealt with in such a way that they are unlikely to reappear. At the same time, there will be a much reduced likelihood that they will cause widespread corruption of data held in the system. The same applies to bugs inadvertently introduced into the system when changes are made to correct defects or to introduce new facilities. If the effects of such

consequential bugs can be contained, then the management will make changes with more confidence and be more willing to respond to users' requests. A protection system that enables access privileges to be kept at all times to the minimum also goes a long way to containing the effects of hardware faults, since many of these will lead to run time violations before they have had time to cause widespread corruption of information in the system. To sum up, one may say that a protection system that enables access to be limited in the manner described makes the system more rugged and better able to withstand the strains imposed on it in practical operation.

THE CAP SYSTEM

Domains of Protection

The set of capabilities to which a process has access - that is, can cause to be loaded into the processor capability registers - constitutes the domain of protection.

A special instruction is needed for changing the domain of protection. Such a change is associated by convention with the process entering a new procedure and the instruction is known as an ENTER instruction. The ENTER instruction is, therefore, a special kind of procedure call. In the early studies of Fabry (see Wilkes, 1975) on which Figure 1.2 (page 4) is based, it was envisaged that, before the execution of an ENTER instruction, a number of capabilities would be loaded into capability registers and that these capabilities would continue to be available to the process when it had entered the procedure. Such capabilities would in effect be passed as parameters to the procedure; since one or more of them could be a capability for a capability type segment, there would be no limit to the number of capabilities that could be passed in this way. One of the capability registers could, by convention, be used to hold the capability for a segment containing global capabilities that were always to be available to the process.

In order for an ENTER instruction to operate correctly in the system shown in Figure 1.2 an enter capability, denoted by EN, must be loaded into one of the capability registers. The enter capability specifies a segment of memory that is, in fact, of capability type. The first entry in this segment is by convention a capability for the code segment of the procedure. The ENTER instruction brings about two actions. One is to transfer control to the code of the procedure (at a specified point, namely the beginning) and the second is to make

available to the process a regular capability for the segment specified by the enter capability. It will be seen that the sole use that can be made of an enter capability is to use it in conjunction with an ENTER instruction to enter the procedure to which it refers. The capability segment specified by the enter capability contains capabilities that are intended for use by any process that has occasion to run in the procedure, but only while it is actually doing so. There is no way in which the process can get access to these capabilities without entering the procedure. The code of the procedure must be so written that, at the time of exit from it, none of these capabilities are loaded in capability registers. In certain circumstances a simple jump might be used for leaving the procedure. This, however, would imply that the procedure was thoroughly trustworthy, since a jump to the wrong location could give rise to loss of protection. This potentially dangerous situation can be avoided if an ENTER instruction is used for leaving a procedure as well as for entering it. A procedure designed in this way may be called a protected procedure. There is a close parallel between protected procedures and abstract data types.

In the CAP project these ideas have been developed further. A program is regarded as being composed of a collection of protected procedures, so that at all times the process is running in one protected procedure or another. It is moreover recognised that, in accordance with the principles of structured programming, protected procedures will be used in a nested manner; consequently parameter passing is implemented by means of a stack and there is a RETURN instruction that has the effect of restoring the situation that existed before the corresponding ENTER instruction took effect. The RETURN instruction is used for leaving the procedure instead of another ENTER instruction, and has the same property of preventing any leakage of capabilities from one domain to another. The fact that the system is designed on the understanding that the programmer will use nested ENTER and RETURN instructions to give a hierarchical organisation to the flow of control does not in any way imply a hierarchical protection system. In fact, as already pointed out, in a capability architecture the two functions of organising the flow of control and of providing protection are completely divorced. In the CAP, a protected procedure consists of code and data segments encapsulated so as to form a single object. Very frequently the same code segments or data segments are used with different workspace segments by a number of protected procedures performing similar functions for different processes or for the same process but in different circumstances. These are referred to as instances of the same protected procedure.

Protection Environment for a Process

There is in the above discussion an implicit assumption that one process only runs in the computer at a given time. Something more is required if it is desired to design an operating system that will run a number of independent user jobs at the same time. In such systems each user job runs as a separate process, that is, it has its own virtual processor. One of the functions of the operating system, or more strictly of that part of the operating system sometimes known as the coordinator, is to create a sufficient number of virtual processors from the limited number - in practice one or two - of real processors with which the system is provided. This it does by time slicing. Each process has its own process base, in which its status is recorded when it is not actually running in one of the real processors. The user processes may be regarded as subprocesses of the main process that runs in the coordinator.

It is necessary to be able to give one user process privileges that are not given to another, and vice versa. Thus, on activating a user process, the coordinator must define a protection environment for it to run in. This environment includes all capabilities that the process will need, but in general, not all of them will be available to the process at the same time; as it runs it will move from one domain of protection to another, the changes being brought about by the execution of ENTER instructions in the manner described above.

Having defined the protection environment, the coordinator must cause the user process to be entered. Return to the coordinator can take place either as a result of the process executing the appropriate instruction, or as a result of the occurrence of an interrupt or a trap. In all cases the status of the process must be preserved. It may be noted that in a capability computer there is more information to be preserved than in a conventional computer. When a process is entered from the coordinator sufficient information must be recorded in the microprogram memory to make the return possible. Specially microprogrammed operations must therefore be provided both for entering and returning from a process. The way in which this is done in the CAP system is described below.

Relative Capabilities

In a segment capability as so far defined, the base is an absolute address in memory and the capability selects a segment out of the entire available memory. Such capabilities may be called absolute capabilities. A natural extension is to define a relative capability in

which the base is specified relatively to the base of some other segment; thus, a relative capability selects a subsegment out of a segment already defined. In addition to the three components already mentioned, namely, base, limit, and access code, a capability must now include a reference either to the whole memory or to some other capability. In formal terms this may be expressed as follows:

capability ::= (base, limit, access code, reference)
reference ::= whole memory | capability

There is thus a chain of reference leading back from a relative capability and ultimately terminating on an absolute capability. By following this chain back a relative capability may be evaluated, that is, replaced by an equivalent absolute capability. The access code in this absolute capability is computed during the process of evaluation so as to be in no respect more permissive than any of the access codes encountered on the way.

The ability to create relative capabilities facilitates the handing on by a process to subprocesses dependent on it of a selection of the memory access privileges that it has available. The fact that all capabilities in the system are ultimately dependent on a relatively small number of absolute capabilities also facilitates memory management and the revocation of outdated capabilities when that becomes necessary.

The Process Resource List

The set of capabilities available to the coordinator are contained in a segment known as the master resource list (MRL). Similarly, the set of capabilities available to a subprocess running under the coordinator are contained in a segment known as its process resource list (PRL); there is one PRL for each subprocess. PRLs contain relative capabilities defined ultimately with respect to absolute capabilities in the MRL. The MRL is the only place where absolute capabilities are to be found.

The capabilities directly available to a process at any time are contained in a number of capability segments. There is hardware support for up to 16 such segments but, in the ordinary way, processes use six only. These are numbered from 1 through 6, but are also referred to by the letters G, A, N, P, I, and R. The G capability segment contains global capabilities and remains unchanged during the life of the process. The P, I, and R capability segments are changed when an ENTER instruction takes effect and the process enters a new protected procedure. P is commonly used to contain capabilities for code segments, I to contain

capabilities for a stack and other segments peculiar to the process, and R to contain capabilities for data segments permanently associated with the procedure. These uses are, however, a matter of convention. Not all protected procedures make use of both I and R. The N and the A capability segments are implemented as the top and next to the top items on a stack known as the C-stack. An ENTER instruction pushes the stack down and the old N capability segment becomes the new A capability segment. A MAKEIND instruction creates a new N capability segment on top ready for the next call. A RETURN instruction pops the stack up and reverses the effect of ENTER.

Implementation of ENTER, RETURN, and MAKEIND Instructions

The actions of the ENTER, RETURN, and MAKEIND instructions that have been described above are implemented in the following manner. Slots 1 through 6 in the process base are designated to hold offsets in the PRL of the capabilities for the 6 capability segments G, A, N, P, I, R. The entry in the first slot remains unchanged throughout the lifetime of the process. Offsets in the PRL of capabilities for the three segments that are special to a protected procedure are held in the enter capability for the procedure, and are copied by the ENTER instruction into slots 4, 5, and 6 in the process base, where they overwrite the entries that were there before. The ENTER instruction also copies the entry from slot 3 to slot 2, thus making the old A capability segment become the new N capability segment. The MAKEIND instruction forms a new capability segment at the top of the stack and puts the capability for it into either slot 2 or slot 3 in the PRL, these slots being used alternately by MAKEIND instructions. The ENTER instruction preserves on the stack sufficient information about the former entries in the slots in the process base and the PRL for them to be reinstated in due course by a RETURN instruction. The first item in the P segment of a protected procedure is a capability for a segment containing the code and the final act of the microprogram implementing the ENTER instruction is to transfer control to the beginning of this segment.

The example on the next page shows the contents of the PRL of a process, together with the contents of its process base, before and after entering a new protected procedure.

PRL		Process Base			
			(a)	(b)	(c)
0	seg PB	0	-	-	-
1	seg C-stack	1	1	1	1
2	seg A_1	2	2	3	3
3	seg N_1	3	3	-	2
4	seg I_1	4	6	11	11
5	enter 6,4,7	5	4	15	15
6	seg P_1	6	7	10	10
7	seg R_1	7	-	-	-
8	-	8	-	-	-
9	seg G	9	-	-	-
10	seg R_2				
11	seg P_2				
12	-				
13	-				
14	enter 11,15,10				
15	seg I_2				
16	-				

The PRL contains a mixture of segment capabilities and enter capabilities. The contents of the process base are shown as follows:

(a) when the process is running in protected procedure 1.

(b) after the execution of an ENTER instruction using the enter capability at offset 14 in the PRL. The process is now running in protected procedure 2. Nothing has been changed in the PRL, but the segment corresponding to the capability at offset 2 should now be known as A_2 instead of N_1.

(c) after the subsequent execution of a MAKEIND instruction. A new N segment, N_2, has been created on the C-stack and (although this does not appear in the table) a capability for it has been put at offset 2 in the PRL overwriting that for A_1. The execution of a RETURN instruction in protected procedure 2 restores the process base to the state shown at (a), and also restores the PRL to its original state.

It will be observed that the capabilities for the three capability segments associated jointly with the process and the procedure are in the custody of the process, since they are embedded - or rather pointers to them are embedded - in enter capabilities held in its PRL.

The only way that the process can make use of these capabilities, however, is by executing an ENTER instruction for the protected procedure with which they are associated.

The address of a word in memory must specify the identity of the capability segment containing the capability for the segment concerned, the offset of the slot in the capability segment containing that capability, and finally the offset in the segment of the word required. Thus an instruction to add into the accumulator word 26 of the segment defined by a capability standing in the P capability segment - that is, capability segment number 4 - at offset 3 could be written:

XADDS 4/3/26

The number of the capability segment and the offset in it - 4/3 in this instance - are together referred to as the capability specifier.

Setting Up a Subprocess

In order to activate a subprocess the coordinator must possess or acquire a segment which will become the PRL of the subprocess. This segment will contain the capabilities that the subprocess will need, together with capabilities for segments that will form its process base and capability segments. The coordinator then executes an ENTER SUBPROCESS (ESP) instruction having as an argument a capability for the segment just mentioned. The microprogram which executes the ENTER SUBPROCESS instruction dumps the status of the coordinator's process in its own process base and activates the subprocess. Return to the coordinator is brought about in due course by the execution of an ENTER COORDINATOR (EC) instruction. This dumps into the process base of the subprocess the status of that subprocess, and reloads the status of the superior process from the process base of the coordinator.

The CAP microprogram was designed in such a way that a subprocess, for example a user process, can itself act as a coordinator for subprocesses running below it. For each subprocess it must first of all prepare a segment which will act as the PRL. It can then enter or re-enter the subprocess as necessary by executing an ENTER SUBPROCESS instruction with a capability for that segment as argument. The subprocess can act in a similar way, so that a hierarchy of coordinators of indefinite depth can be established, the top coordinator being known as the master coordinator. Although this feature was implemented and is described in what follows, it has not, in fact, been found to be useful for the purposes for which it was intended. The reasons for this will be discussed in Chapter 5.

Note on Nomenclature

Two extreme points of view may be taken about the status of the entries in capability segments and in PRLs. They may be regarded as mere pointers constituting a chain of reference which finally ends on the absolute capability in the MRL. The term capability is then reserved for what has been termed above as an absolute capability. The other extreme is to refer to all the entries as capabilities whether they occur in the MRL, PRL, or a capability segment, it being understood that the actual format is not necessarily the same in all cases. The latter point of view has been adopted in the present book. In some publications relating to the CAP, entries in capability segments have been referred to as capabilities, but in the case of PRLs the term PRL entry has been used. The same was done for the MRL. In earlier publications the capability segments were referred to as indirection tables or indirectories; this nomenclature emphasised the fact that the capability segments serve to provide access to a selection only of the capabilities in the PRL.

The Capability Loading Cycle

In the CAP there are 64 capability registers contained in a 64 word store known as the capability store. No segment can be accessed by the main memory access circuits unless an (evaluated) capability for it exists in this store. If a capability is referred to in the address of an instruction and it is not already to be found in the capability store, the microprogram automatically enters the capability loading cycle. This cycle evaluates and loads the missing capability, if necessary overwriting, according to appropriate rules, a capability already loaded. In addition to holding evaluated capabilities, the capability store also contains a data structure which (1) provides a chain of reference from each evaluated capability back to the absolute capability in the MRL on which it is ultimately based, and (2) provides a backward chain of reference for nested ENTER SUBPROCESS instructions.

To evaluate and load a segment capability, the microprogram first reads the capability from the capability segment in which it resides. Like all segment capabilities, it will consist of a relative base, a size, and an access code, along with a pointer to the capability in the PRL of the process. This latter capability will next be read from the PRL and will again consist of a relative base, a size, and an access code, together with the address of a capability in the address space of the next superior process. The bases, sizes, and access codes are combined according to obvious algorithms, and then the cycle proceeds with the reading of the capability in the superior's address space, just

as was done for the capability in the junior's address space. The cycle continues in this way and finally terminates when the MRL is reached.

When a capability loading cycle has terminated and the evaluated capability has been loaded, the original instruction in the program which gave rise to the initiation of the capability loading cycle is retried and, if all has gone well, will now be executed successfully. It may, however, happen that at some stage in a loading cycle a capability needed by the microprogram - possibly for a capability segment - is found not to be loaded. It would be awkward to make the microprogram work recursively, and instead the following procedure is adopted. If a capability needed by the microprogram is found to be missing, the current capability loading cycle is abandoned and a new one initiated to load the missing capability. This cycle may itself have to be abandoned and another initiated. The process of initiating capability loading cycles and abandoning them will be continued as long as necessary, but eventually a cycle will run to completion and an evaluated capability will be successfully loaded. The original instruction will then be retried and, since the capability that was missing initially will still be missing, the whole procedure will be repeated. Quite possibly the original instruction may have to be retried a number of times, but eventually all the required capabilities will be loaded and the instruction will be successfully executed.

The overwriting rules for the capability store are designed in such a way that the chains of reference referred to on the previous page are preserved intact. In particular, the evaluated capabilities for the MRL and the process base of the top level process are never overwritten; neither are those for the PRL and the process base of the current process, nor those for capability segments that have evaluated capabilities in the capability store. During the capability loading cycle a wide variety of errors may be detected. For example, a capability segment may contain a pointer beyond the end of the PRL, or a PRL may contain an address that is invalid in the environment of the superior process. Altogether there are 16 distinct types of capability loading cycle trap. Some of these traps may be caused by the user writing an erroneous virtual address, others can reflect a corruption of the system or a subprocess set up incorrectly, while some may be deliberately caused to indicate, for example, that a segment asked for is not loaded in the main memory. In all cases the microprogram action is the same: it causes the coordinator of the offending process to be resumed and supplied with detailed information about what has occurred. It is then entirely up to the software to classify and interpret the

various causes of failure. It may be noted that, since the error action is to resume the next senior process, there is no means of handling faults of this, or indeed of any other, kind in the most senior process. Reliance, therefore, must be placed on the master coordinator being written correctly and all the material that it requires must be kept permanently in main memory.

Capability Handling Instructions

In addition to the ENTER, RETURN, MAKEIND, ENTER SUBPROCESS, and ENTER COORDINATOR instructions whose actions have been indicated above, there are three other instructions which concern capabilities. The MOVECAP instruction enables a capability to be copied from a capability segment to another segment for which write capability access is available. The REFINE instruction copies a capability in the same way as MOVECAP but, in addition, is able to alter the base, size, and access code in the capability so that the access it affords is restricted. The FLUSH instruction is used to ensure that the capability store does not contain capabilities that have become obsolete as a result of changes made to one or more of the capability segments or PRLs which were used by the capability loading cycle when it loaded them. The operation of flushing is done automatically in the case of MOVECAP and REFINE; an explicit FLUSH instruction is only required on the rare occasions when changes to a capability segment or a PRL are made by a (highly privileged) procedure that has D-type access to it.

Capability Formats

All capabilities occupy two consecutive words in the CAP memory. The leading bits of the second word always indicate the type of the capability. In the case of a segment capability, the first word contains either an absolute memory address or, in the case of a relative capability, a reference to the capability with respect to which it is defined. In the case of a capability in the PRL, the reference consists of a segment specifier; in the case of a capability in a capability segment, the reference is to the PRL and consists simply of an offset. The second word in a segment capability contains a statement of the length of the segment. It also contains five access bits known as WC, RC, W, R, E, corresponding respectively to write capability, read capability, write data, read data, execute instruction.

The format of an enter capability depends on whether it occurs in a PRL (or in the MRL) or in a capability segment. In the former case, the first word contains three 10 bit numbers. These are offsets, in the

same PRL (or in the MRL), for the slots where capabilities for the P, I, and R capability segments are to be found. The second word contains 14 bits, which are known by analogy with the bits in a segment capability as access bits. All that the microprogram does with them, however, is to hand them over to the programmer to use as he thinks fit. In the CAP filing system, five of these bits are used in enter capabilities for controlling access to file directories. Enter capabilities in capability segments are all defined in terms of a corresponding enter capability in the PRL, and their first word simply contains the offset of this capability. The second word contains the 14 access bits.

Peripheral Capabilities

Access to peripherals is controlled by capabilities. In order to avoid having a special form of capability for this purpose, a device has been adopted which involves sacrificing a small amount of main memory, namely that in the segment delimited by the addresses 0 and 31. Each word in this segment corresponds to one of the peripherals and any capability that would allow it to be accessed allows the corresponding peripheral to be activated. Instructions are provided for transferring either a single character or a block of words to or from a peripheral.

In the CAP all memory accesses for peripheral transfers take place in the protection environment of the process doing the transfer and thus use the capability unit in a completely normal way. This approach avoids the problems that arise when autonomous channels bypass the normal protection system. It is made possible by the fact that all input and output is buffered in an auxiliary computer. A practical consequence is that block transfers take place in a series of bursts during which all memory cycles are used, instead of by the more usual cycle stealing procedure.

Software Capabilities

The protected procedure mechanism is capable, in principle, of providing as finely structured a protection system as is required. There are, however, certain overheads associated with the use of protected procedures and in practice it has been found both inconvenient and inefficient to have to establish a separate protected procedure for every distinguishable action that it is desired to protect. Software capabilities enable this to be avoided.

Advantage is taken of the fact that information from capabilities can be read into arithmetic registers. A software capability is held in a capability segment (and therefore is just as unforgeable as a regular

capability) but it is interpreted by a program rather than by the hardware. It enables a single protected procedure to be used to perform a variety of related functions with separate protection for each of those functions. The user calls the protected procedure in the ordinary way, but quotes a software capability as an additional argument. This is inspected within the procedure and the call rejected if it is not in order. An example of a software capability is one giving authority for a particular message channel to be set up by the appropriate protected procedure. The first two bits in the second word of a software capability indicate that it is a capability of software type. Otherwise the way that the bits are used is a software convention and varies from one software capability to another.

CHAPTER 2

THE CAP HARDWARE AND MICROPROGRAM

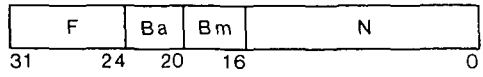
The CAP is a 32 bit computer. Those parts of the hardware that are particularly interesting from the point of view of protection are the microprogram unit and the capability unit, and these are described in detail in this chapter. There is also a floating point arithmetic unit which runs autonomously under its own control. At present the computer has 192K words of main memory, although the addressing scheme would allow much more to be connected. A slave store transparent to the program is provided for the purpose of reducing the effective memory access time.

Since the construction of the CAP computer was undertaken for the sole purpose of providing a vehicle for research in memory protection, an approach was taken to the provision of channels and input-output facilities which would minimise the amount of effort that would have to be spent in design and construction. There are only two peripheral devices attached to the CAP itself; they are a teletype and a paper tape reader whose sole use is to load the microprogram memory from a paper tape. All other peripheral transfers from the CAP are done via a link to an auxiliary computer (a CTL Modular One) to which the main peripherals are attached. These include a fixed head disc unit, a movable head disc unit, a line printer, a paper tape punch, a paper tape reader, and a multiplexer for users' terminals.

The microprogram is contained in a memory of 4K words each of 16 bits. The memory is writable, except for a few words at the top end which contain a read-only bootstrap routine. The CAP is, however, used in such a way that the microprogram memory is loaded once for all when the system is initialised, and no dynamic reloading takes place during operation. The microprogram implements a fairly conventional instruction set and it also implements the special instructions that deal with process and protection domain changing.

The central register unit contains 16 registers - known as B0 to B15 - which are accessible to the program and are used for such purposes as fixed point arithmetic and address modification. There are 16 other registers, A0 to A15, which are not visible to the program, but which are used by the microprogram for working space.

Instructions are 32 bits long, the bits being denoted by d0 to d31 starting at the least significant end. The layout of an instruction is as follows:



where:

F denotes a function code given by d24-d31
Ba denotes a register identified by d20-d23
Bm denotes a register identified by d16-d19
N denotes the signed integer formed by d0-d15

Ba and Bm are registers in the blocks of 16 already mentioned. Their contents will be referred to as ba and bm respectively. It appears to the programmer that B0 always contains 0, although it can be used as an ordinary register by the microprogrammer. B15 holds the address of the next instruction in sequence. The effective address (32 bits) which is presented to the capability unit is n = N + bm.

Representative examples of the conventional instructions are given below with the following additional notation:

[x] denotes the contents of the memory location addressed by the 32 bit quantity x,

& is the bitwise logical AND operation.

Primes are used to denote the contents of a register or memory location after the instruction has been executed.

Function	Explanation
BN	ba' = n
BBPS	ba' = ba + [n]
BBAN	ba' = ba & n
BBAS	ba' = ba & [n]
ESB	ba' = [n]; [n]' = ba
ESBPS	[n]' = ba + [n]; ba' = [n]
SLRN	shift ba logically n places right
MODNN	add n to address in next instruction
MODNS	add [n] to address in next instruction
JNLT	if ba < 0 then b15' = n; that is jump to n
JSLT	if ba < 0 then b15' = [n]
SREN	ba' = b15; b15' =n (a subroutine jump)
SREFN	[ba]' = b14; [ba+1]' = b15; [ba+2]' = b13; b14' = ba; b15' = n, (ALGOL68C procedure call and BCPL function application)
SREFS	as SREFN but b15' = [n]
CASE	if 0 < ba < n, b15' = b15 + ba (for the ALGOL68C CASE statement)

MOVE [bm+r]' = [ba+r] where r = 0,1,2,....bn-1
GETBYTE ba' = byte bn of string of bytes starting in [bm]

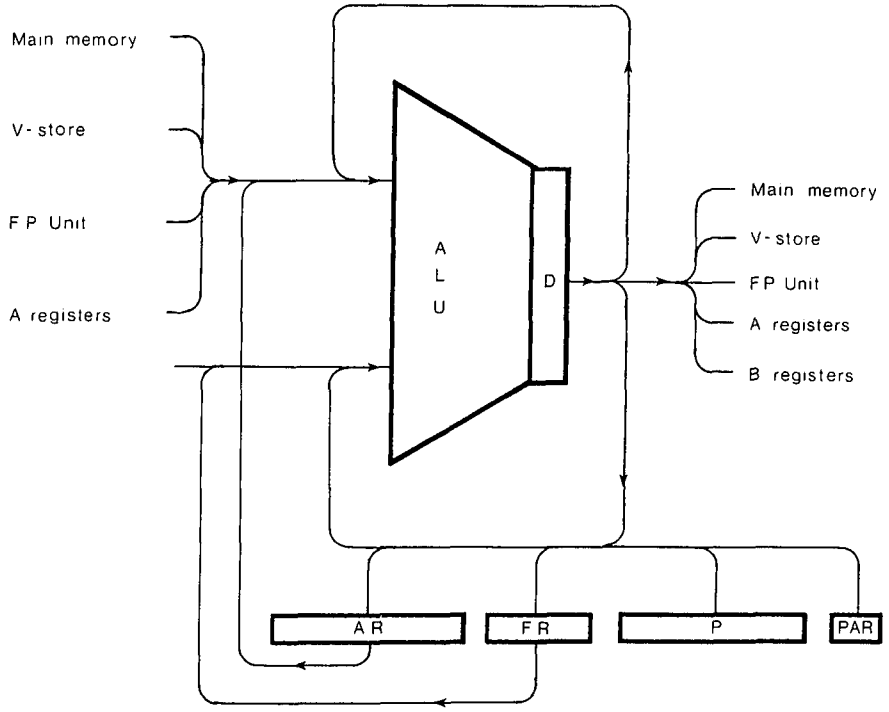


Figure 2.1 The Central Register Unit

The Microprogram Unit

Figure 2.1 is a block diagram of the central register unit showing the data paths that exist between the various registers. The execution of a machine instruction begins with the simultaneous loading from memory of its 16 most significant bits into FR and its 16 least significant bits (with the first bit extended leftward to make 32 bits in all) into AR.

The microprogram must be able to read certain input signals and certain internal flip flops, and must similarly be able to supply control signals to the central register unit. For convenience all these inputs and outputs to the microprogram are grouped together so as to be accessible in a single address space. This is referred to as the V-store, although it is not a store in any usual sense of the term. The words in the V-store are nominally 32 bits wide, but in most cases only a

few bits in each word have any significance. Some locations are read-write, some are read-only, and others are write-only; some flip flops are represented by two separate bits, one in a read-only word and one in a write-only word. Register D (32 bits) plays a central role in the operation of the microprogram. The results of all arithmetic and logical operations performed by micro-instructions pass into it, as well as going to any other destination that may be specified for them. The same applies to information that may be read from or to the main memory, the V-store, and the floating point arithmetic unit.

Micro-instructions fall into three groups, each with its own format. Micro-instructions in the first group take two operands, perform a logical or arithmetic operation using them, and send the result to a selected destination. Micro-instructions in the second group contain a 12 bit address referring to a word in the microprogram memory and are used for accessing that memory. Two jump instructions are included, one of which saves a link and is used for entering micro-subroutines. The third group contains micro-instructions for accessing the V-store and performing tests on information in it. This group also contains micro-instructions for shifting the number in the central register D either arithmetically or logically.

The microprogram addresses the main memory via the capability unit. It does this by placing the full address - that is, capability specifier and offset - of the required word in the 32 bit register P and at the same time placing in the 8 bit register PAR information about the type of memory access that is required. The capability unit is then responsible for checking the validity of the request and adding the segment base to the offset supplied.

The Capability Unit

The principal constituent of the capability unit is the capability store which contains 64 words of 84 bits, 13 of which are used as parity bits. The locations in this store are, as was explained in Chapter 1, known as capability registers and contain the evaluated forms of those capabilities that are in current use. The microprogram is capable of accessing the capability store and is responsible for evaluating and loading capabilities as required. Special circuits are provided within the capability unit for performing address bound checking, for checking access rights, and for adding the segment base to the offset of the required word. Some of the fields in the capability registers are used to contain pointers from one capability to another; in this way the microprogram can establish a data structure that models

dynamically the process structure of the program that is running. Except when engineering tests are being done, all accesses to the main memory in the CAP are made via the capability unit, and the checks mentioned above are always performed.

A typical access request to the memory is for a single read or write operation of C-type or D-type as the case may be. The memory access cycle usually consists of two steps. The first is the execution of a micro-instruction which loads an address into the register P and access request bits into register PAR. As soon as this is done, the capability unit proceeds autonomously to check that the requested access is permitted. Step 2 is the execution of a second micro-instruction which reads the required word from memory or writes a word to memory, whichever is called for by the access request bits. There is an interlock to prevent step 2 being performed before the capability unit and the memory are ready. Other micro-instructions, unconnected with memory access, may be placed between steps 1 and 2 and will be executed in parallel with the operation of the capability unit. In the case of some instructions, the access bits indicate that two accesses to memory are required, the first being for reading and the second for writing. In this case step 2 will consist of the execution of a read micro-instruction and there will be an eventual third step consisting of the execution of a write micro-instruction.

Between steps 1 and 2 the capability unit searches the capability store for the capability specified in the address that has been placed in register P. If this capability is not found, a trap, or rather a jump, occurs at microprogram level when step 2 is attempted and the microprogram routine for executing the capability loading cycle is entered. This loads the required capability, overwriting if necessary one of those already there. A microprogram trap of a different sort will occur if the access demanded by the program is an illegal one, for example, if it is to a word lying outside the limit of the segment concerned. In this case the error is reported, via the microprogram, to the program, where it is to be presumed there will be a routine for dealing with the error.

In addition to the fields containing the base, limit, and access code of a capability, each word in the capability store has a tag field (divided into two parts), a count field, and an ancillary field. The contents of the two parts of the tag field taken together constitute a short term unique identifier for each capability held in the capability store, and are used by the hardware when ascertaining whether or not a particular capability is present. The entries in the tag field serve the

additional purpose of giving a chain of reference back from a capability associated with the running process to the absolute capability in the MRL with respect to which it is ultimately defined. The second part of the tag field of a capability associated with the running process contains the address in the capability store of the evaluated capability for the relevant capability segment, and the first part contains the offset in that segment of the capability in question. The tag field of the capability for the capability segment similarly contains the address in the capability store of the capability for the PRL and the appropriate offset within it. Entries in the ancillary field provide a chain of backward reference for capability segments and PRLs. In the case of a PRL, the entry gives the address in the capability unit of the capability for the coordinator's process base. In the case of capability segments, it contains the address of the capability for the appropriate PRL. There is no entry in the case of other segments. The purpose of the entries for capability segments is to enable a procedure to use a capability segment containing capabilities defined at a higher level in the process hierarchy than that of the PRL of the process in which it runs. For reasons explained in Chapter 5 on page 59 this facility has been little used. Figure 2.2 illustrates the way in which the two parts of the tag field and the ancillary field are used.

It is the function of the microprogram to write information into all the fields of the capability store and to update it as the execution of the program proceeds. The tag field together with the fields containing the base, limit, and access code of a capability, are referenced during the memory access cycle by the hardware of the capability unit which, as already explained, acts independently of the microprogram. The count and the ancillary fields are not accessed by the hardware and the way they are used is entirely a function of the way the microprogram is written. The count field, as its name implies, records the number of times the capability concerned is referenced from within the capability unit or from one of the registers in the microprogram unit. The information in the count field is used by the microprogram when it is necessary to overwrite a capability in order to find room for a new one.

Memory Access Validation

When the microprogram generates a memory request by loading P, it simultaneously loads 8 access request bits into PAR. The capability unit uses 5 of these bits to check the validity of the request and, if it is valid, the bits are then used by the memory control unit which makes the

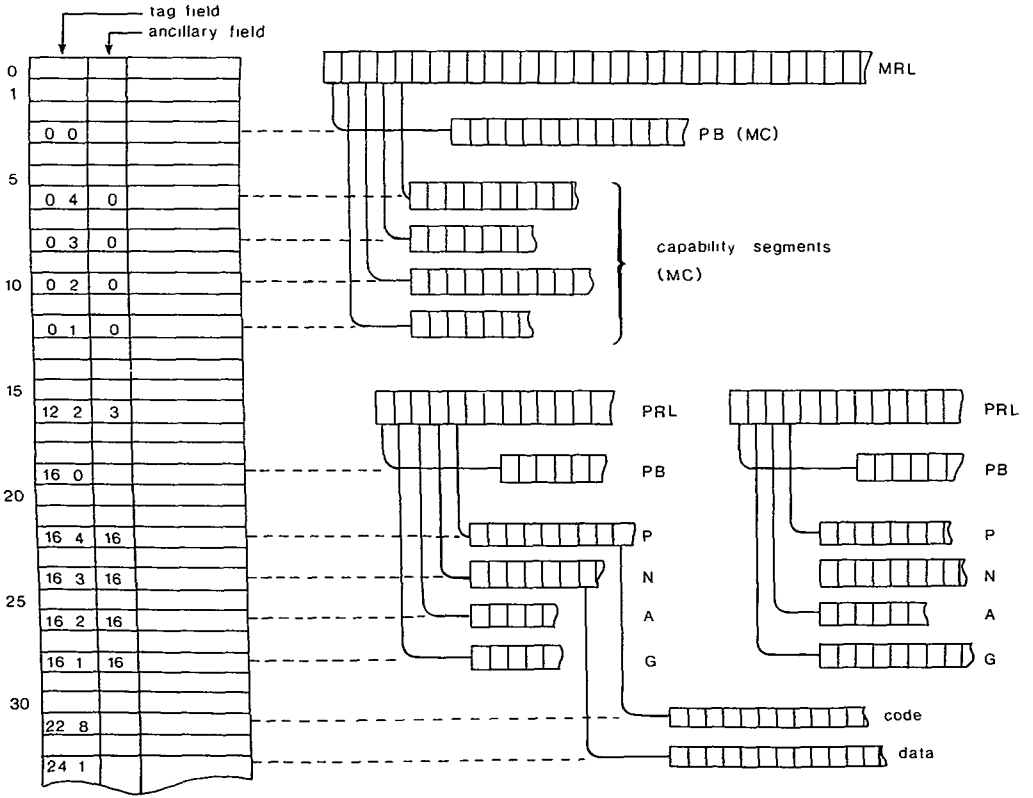


Figure 2.2 Use of the Tag and Ancillary Fields in the Capability Store

required location in the main memory available to the microprogram. The 5 bits correspond to the following operations: write capability; read capability; write data; read data; execute instruction. If one or more read type bits are 1s and none of the write type bits are 1s, then the memory control initiates a memory read cycle. The microprogram is expected to complete this cycle by accepting the word from the memory (step 1 in the above description). If one or two write type bits are 1s, then the memory control initiates a write cycle which the microprogram is similarly expected to complete in due course. If a combination of both read and write type bits are 1s, then the memory control first initiates a read cycle. When the microprogram has completed this cycle by accepting the word (step 2), the memory control automatically initiates a write cycle to the same address, and this cycle is in due course also completed by the microprogram (step 3).

In order to check the validity of a memory access request, the capability unit first searches the capability store to see whether it contains a capability for the segment in which the requested word lies. In the description that follows of the manner in which the search is made, it will be assumed that the capability unit is in normal mode. There are three other modes which will be described later.

During the matching operation the second part of the tag field is compared with the segment number standing in the P register. The first part of the field is matched with a word from a small store known as the TGM store. This contains 16 words and the one used for the match is selected by the 4 most significant digits in the P register; these are the digits of the segment specifier that indicate which capability segment belonging to the current process contains the capability for the segment being accessed. The matching operation is shown schematically in Figure 2.3. It follows from the above description that the TGM store must contain at the appropriate place the address in the capability store of the evaluated form of the capability for that capability segment, and that a copy of that address must appear in the first part of the tag field of the evaluated capability. It is the duty of the microprogram to write this address into the tag field and into the TGM store.

The content addressing of the capability store on its tag field is performed partly by association and partly by searching. As seen by the microprogram the capability store appears as a straightforward store of 71 bits plus some parity bits, but its internal structure is more complex. It consists in fact of two separate stores. One is a 64 word store 14 bits wide (plus some parity bits). The second is a 16 word store, each word containing 228 bits (plus some parity bits). The first

store holds the count and ancillary fields of the evaluated capabilities and the second holds the tag, base, limit, and access fields, 4 sets to a word. When the capability store is accessed by the microprogram, the appropriate set of 57 bits from the second store is selected at the output.

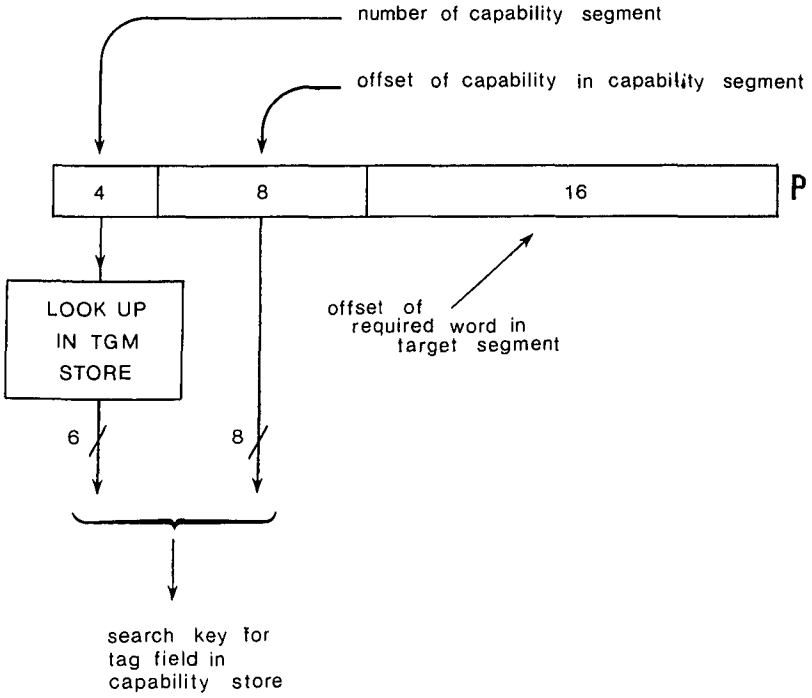


Figure 2.3

It is only the second store that is accessed autonomously by the capability unit when searching for a capability. The search starts by one of the locations in that store being read. The tag fields in the 4 quarter words thus obtained are compared simultaneously by the hardware with the digits from the TGM store and the segment field of the P register. If a match is obtained the required capability has been found and the contents of the selected base, limit, and access fields are passed to the request validation circuits. If a match is not obtained, the next word in the store is read and the process repeated. This goes on until either a match is obtained or failure reported. There is obviously a premium on placing capabilities in the store where they will

be found most rapidly and the microprogram is designed to do as good a job in this respect as is reasonably possible.

The starting address for the search is derived from the 4 most significant digits of the word in the TGM store used to match the second part of the tag field by inverting the most significant digit. The practical effect of this is to start the search approximately 32 places on in the capability store from the address initially contained in the TGM store. A 4 bit register, which can be read by the microprogram via the V-store, contains a count of the number of words (if this is greater than 2) that had to be examined before one was found that would match. If no match is obtained, the register holds the number 15. Each word in the TGM store contains a bit additional to those used in the match. If this bit is a 0, the equivalence logic is disabled and a search will always fail.

The mode in which the capability unit operates is determined by two bits in a register accessible to the microprogram through the V-store. Of the four possible modes, the normal mode has already been described. In absolute mode, which is only used for the initial bootstrap, for hardware testing, and for maintenance, all capability checks are bypassed and the 20 least significant bits of P are routed directly to the main memory access circuits. In this mode, parity errors in main memory do not produce a microprogram trap during the extraction and decoding of an instruction. In direct mode and last mode no search of the capability store takes place. In direct mode the capability whose tag, base, and limit fields are passed to the address validation circuits is the one standing at the address in the capability store given by the 6 least significant digits in the segment specifier field of P. The microprogram is thus enabled to address the capability store directly. Last mode is used when it is required to make a series of main memory accesses using the same evaluated capability. The microprogram must first write the address of the capability in the capability store to the proper place in the V-store.

Access and limit checks are made for all modes of the capability unit other than absolute mode. The limit checking circuits will indicate an error if the segment offset number in P is equal to or greater than that specified in the limit field of the capability. Circuits for performing this comparison are permanently connected and will yield a decision in the time they take to settle down. Similarly, the permanently connected access validation circuits will indicate an error if any bit in the access field in P is a 1 when the corresponding bit in the access field of the capability is a 0. As already explained, the

corresponding traps do not take place until the microprogram attempts to complete the memory access cycle.

The Microprogram

The CAP microprogram falls into three main parts: that which implements the regular instruction set of the machine, that which performs operations upon capabilities, and that which handles input and output. The common starting point for all these is a sequence, known as stage 1, which extracts and decodes instructions. Stage 1 includes instruction fetch, incrementation of the program counter, computation of the effective address and its dispatch if neccessary to the capability unit (with the requisite access bits, if any), and branching on the function code. These operations can be accomplished in four or five micro-instructions, depending on the instruction being executed. For a simple instruction, such as one which adds a number from the memory into a register, these micro-instructions can also perform the operations required to complete the execution of the instruction. In more complex cases a jump occurs and the implementation is completed by a further sequence of micro-instructions; for example, a test-and-count instruction takes a further three micro-instructions. All instructions are, as a matter of policy, implemented so as to be restartable from the beginning if a trap occurs on any memory access. The MOVE instruction and related instructions are arranged so that, when they are restarted after an interruption they will be resumed at the point at which they left off. Adherence to this policy imposed a practical upper limit on the complexity of the special instructions that could be implemented.

The part of the microprogram concerned with capability operations may itself be subdivided into three main parts. First, there is the capability loading cycle to which reference has already been made and which is initiated whenever a search for a particular evaluated capability in the capability store fails. Secondly, there is microcode to implement a group of instructions for changing the environment and passing capabilities as arguments. These instructions are: ENTER, RETURN, MOVECAP, REFINE, MAKEIND, and FLUSH. The third part implements the ENTER SUBPROCESS and ENTER COORDINATOR instructions. All the above instructions make use of capabilities for memory access and may thus cause capability loading cycles to be initiated. They are subject to the same restartability discipline as are regular instructions.

The ENTER, RETURN, and MAKEIND instructions perform operations on the C-stack. Since the C-stack is not required to be resident in memory, it cannot be guaranteed that a capability for it will be loaded. The

microcode for the above instructions, therefore, starts by ensuring that this is so.

In order to make the microprogram as compact as possible, certain common operations on the C-stack - and also on the process base - are performed by micro-subroutines. The ENTER and RETURN instructions can give rise to traps and there is even the possibility that the C-stack will be malformed; care was taken, therefore, when writing the microcode for the ENTER and RETURN instructions, to ensure that at all times one domain or the other is sufficiently well established to have a trap reported to it.

The MOVECAP and REFINE instructions both perform operations on capability segments. As a result of their actions some of the evaluated capabilities in the capability store may be rendered invalid and the microcode must mark them as such. It must also mark as invalid any evaluated capabilities whose original loading cycles made use of them. The microcode for doing this is also used by the FLUSH instruction which explicitly invalidates capabilities in the capability store.

The ENTER SUBPROCESS instruction establishes a new current PRL and a new current process base. The microcode loads the appropriate capabilities and checks that the segment about to become the new current process base is adequate for the purpose; it checks, for example, that the segment is long enough and that the capability for it gives read and write access. One of the functions of the ENTER SUBPROCESS instruction is to save the contents of the processor registers in the old process base before copying new values into them from the new process base. As an aid to handling interrupts and interprocess communication, a bit [known, following Saltzer (1966), as the wake-up waiting switch] in the old process base is tested and, if it is found to be set, the ENTER SUBPROCESS instruction is abandoned. The ENTER COORDINATOR instruction performs the converse operation to ENTER SUBPROCESS. In addition to being entered explicitly, the microcode for this instruction is also entered whenever any trap occurs as a result of an error detected either by the microprogram or by the hardware.

A separate section of the microprogram deals with input and output which, as already explained, are handled through a peripheral computer. There are instructions for transferring both single characters and blocks of words, and for issuing commands to peripheral devices. All such instructions must be supported by an appropriate capability. Information is transferred by the microprogram from a buffer in the CAP computer to a buffer in the peripheral computer and vice versa. These buffers are in areas set aside for the purpose in the main memories of

the two machines and the microprogram is responsible for the manipulation of the necessary pointers. For efficiency reasons the microprogram stores some information relating to input and output in the microprogram memory. Input and output is handled within the current process so that all accesses to the main memory of the CAP use the capability unit in the regular manner and may, in particular, give rise to capability loading cycles and virtual memory traps.

The modular nature of the CAP microprogram makes it possible to introduce changes in one part of the system without affecting others. For example, it would be possible to experiment with different protection structures by making changes to the part of the microprogram that handles capabilities and implements the capability loading cycle; other parts would be very little affected.

CHAPTER 3

THE CAP OPERATING SYSTEM

The CAP operating system is intended to provide an environment in which users' programs may run and in which they may take full advantage of the hardware-supported capability system of the machine. It is also intended to constitute in itself an example of the exploitation of those facilities.

The operating system, like all other programs in the CAP, consists of a collection of protected procedures with linkage between them provided by the microprogram. In particular, the microprogram supports the ENTER instruction, by which protected procedures are entered and left, and the C-stack on which arguments of capability type are passed. By convention, numerical arguments are passed in the processor registers B1 to B5.

Each protected procedure is a self-contained object and no assumptions are made by the microprogram about the language in which the various protected procedures are written. Provided that they conform to the interface outlined above, the protected procedures making up a complete program may be written in a variety of languages. It was at one time thought that some of the procedures constituting the operating system would be written in ALGOL68C and some in BCPL; in fact, as it has turned out, they are all written in ALGOL68C, although a number of other protected procedures closely associated with the operating system - such as a paginator - are written in BCPL. From the point of view of the microprogram all protected procedures have the same status, those constituting the operating system being in no way distinguishable.

Since the protected procedures are connected together by ENTER and RETURN instructions and not by the procedure calling mechanism of the language in which they are written, each protected procedure is compiled as a complete program. Means must be found - for example by the inclusion of machine code sections - to effect the connection. The way this is done in the case of ALGOL68C is described in Appendix 2. An important consequence of the above statement that the protected procedures are all separate programs is that any one of them may be recompiled without affecting the others. This contributes greatly to the modularity of the resulting system.

The fact that there is no requirement that all the protected procedures forming a program should be written in the same high level language means that the compile time type checking facilities normally available in ALGOL68C (for example) are not available for calls from one

protected procedure to another. Since the interfaces between the protected procedures are simple and clearly defined, this lack was not felt to any noticeable extent when the operating system was being debugged. In fact, the compile time checks performed by the ALGOL68C compiler within the individual protected procedures and the run time checks performed by the capability hardware together provide a much more comprehensive set of checks than a programmer normally enjoys when developing a complicated program.

Processes and Their Management

The CAP operating system supports a moderate and fixed number of processes, some of which provide operating system services, and some of which are assigned to the users. The processes are all managed by the master coordinator which performs scheduling and despatching functions, performs some functions in relation to trap handling, and supports an interprocess message system. Some operating system services are provided by protected procedures entered by the user process. Others are provided by system processes activated by messages. Since in order to send a message a user process enters a protected procedure, the interface is the same in both cases, and the programmer usually does not need to know by which technique a particular service is provided. All the user processes and most of the system processes run in a virtual memory in which the unit of swapping is the segment, except that very large segments are handled by a windowing technique. The system processes that do not run in the virtual memory are those which support the virtual memory itself. There is a filing system very intimately related to the virtual memory. Segments are swapped to and from their place of residence in the filing system, rather than from a copy on a logically separate disc. It is thus the case that the notions of file and segment are equivalent in the same sense that they are in MULTICS.

It has been stated earlier that the facilities available to a process are all represented in its process resource list (PRL). The state of the process, both in a hardware and in a software sense, is recorded in the segment called the process base for which there is a capability placed by convention in the first position in the PRL. Alterations to the resources of the process or in its state involve changes either to its PRL, or to its process base, or to both. Every process possesses an enter capability for its own instance of a protected procedure called ECPROC which has the privilege required to make these changes. ECPROC also provides an interface to the facilities of the master coordinator. The client of ECPROC does not need to know whether a

particular service is executed within ECPROC or whether ECPROC calls the master coordinator. The policy has been to provide services entirely in ECPROC whenever possible, since the master coordinator itself is serialised and may only execute services for one process at a time. Furthermore, the coordinator is incapable of taking virtual memory traps and may therefore only touch material which is known to be in the main memory. ECPROC is not subject to this restriction.

Some coordinator services (this expression will be used indifferently to refer to those provided by ECPROC or by the coordinator strictly so called) are available to all callers. Others, such as the creation of new capabilities, are highly privileged and are only available in one or two protection domains. In general, all calls to ECPROC have to be supported by the presentation of a software capability that gives authority to request a particular service. In accordance with common usage, calls on ECPROC will sometimes be referred to as primitives.

Capability Management. The CAP architecture is such that various operations on capabilities are most conveniently performed by the coordinator. ECPROC provides a suitable interface to these. The presenting process must possess the appropriate software capability and it is convenient to refer to such capabilities as permission capabilities. The operations on capabilities performed by the coordinator are the creation of capabilities, the alteration of existing capabilities, and the extraction from existing capabilities of information beyond that which is ordinarily available to users. In addition, there is an operation which ECPROC can perform on behalf of the swapping system which has the effect of disabling all instances of a capability for a segment that is about to be swapped out, or conversely, to reinstate them after it is swapped in. This service is a store management feature rather than a capability management feature.

Interprocess Communication. The interprocess communication system is concerned with information transfer and synchronisation between processes. There are WAIT and WAKE-UP primitives of the usual kind and also provision for passing messages from one process to another. Messages are of four kinds: null messages, data messages, segment messages, and full messages. A null message consists simply of a synchronisation signal. A data message carries four words of data. A segment message carries a capability for a memory segment. A full message carries both four words of data and also a capability for a memory segment. Messages other than null messages may demand a reply.

If a process is in a waiting state, it will be woken up when a message is addressed to it.

The sending and receiving of messages is mediated by a data structure belonging to the coordinator and called a message channel. Each channel contains a statement of the type of message for which it may be used and other types of message may not be sent or received on that particular channel. ECPROC is responsible for setting up and deleting channels. However, the user does not normally use ECPROC directly for these purposes, but goes instead via another protected procedure known as SETUP. The advantage of proceeding in this way is that SETUP is responsible for the administration of an external scheme of message channel names unknown to the master coordinator.

Messages in transit are recorded in a single buffer segment which is common to all processes. It is possible for ECPROC to make use of this segment on behalf of all processes, either to insert a message into it or to remove a message from it, without requiring any special interlock. This has been achieved by careful use of indivisible machine instructions. At the receiving end of the channel an enquiry facility is provided to enable a process to determine how many incoming messages are waiting to be accepted. An error is signaled if an attempt is made to accept a message when there is not one there; it is the responsibility of the receiving program to make use of the WAIT primitive where this is appropriate. No action is taken to ensure that a recipient process is woken up only if the message being sent is one for which it is known to have a receiving capability in its current protection domain; processes must, therefore, be prepared to be woken up in circumstances in which there is no action for them to take.

In the case of messages requiring a reply, it depends on the ECPROC entry used whether the sending process is halted immediately to await the reply, or whether it continues to run; in the latter case it will presumably make an enquiry later and possibly go into a waiting state. In the case of nonreply messages this choice does not exist. Nonreply messages are used extensively in circumstances in which some cleaning up action or action to preserve consistency is required, but in which the further progress of the sender does not depend on the result.

Segment Reservation and Interlocks. Some segments are available to a considerable number of processes, usually by means of instances of the same protected procedure. For example, in the filing system a segment containing a file directory is handled in this way. The master coordinator provides a reservation facility for controlling access to shared segments. This is based on a locking mechanism that allows

multiple readers of a segment but only one writer. The standard reservation primitive causes the calling process to be held up if the segment required is already reserved by some other process, but an alternative call is available which will reserve the segment if possible and if not, give a return code without halting the process. It is a programming rule that shared segments should be reserved before being accessed, but this rule is not enforced by the software or hardware. This has not been found to present a serious hazard in practice because the processes competing for a shared segment typically run in the same simple piece of shared code.

The Handling of Peripheral Interrupts. The coordinator is responsible for waking up the correct process when a peripheral interrupt occurs. The use of peripherals is controlled by capabilities but, in order to claim the interrupts of a particular peripheral, the holder of the appropriate capability must inform the master coordinator that he has it; otherwise the master coordinator does not know where to send the interrupts. If a capability for accessing a particular peripheral were to be possessed by more than one process then confusion would result. This is avoided by placing capabilities for peripherals in the custody of suitable protected procedures.

The Handling of Traps, Errors, and Attentions. Traps are essentially internal interrupts and are generated within the microprogram. Errors are generated within a program. In the CAP system both traps and errors - collectively referred to as faults - are handled in a very similar manner by a routine that forms part of ECPROC. In the case of an error, the program enters ECPROC by executing an ENTER instruction with an appropriate parameter. In the case of a trap, an entry to the master coordinator is made directly by the microprogram which then enters ECPROC, but the code is written so as to simulate the effect of an entry in the normal way by means of an ENTER instruction.

ECPROC first looks to see whether a virtual memory trap or a trap resulting from an attempt to enter a protected procedure just retrieved from the filing system and requiring generation (see page 48) has occurred. These traps are of routine occurrence and must be dealt with efficiently. ECPROC initiates the necessary action and in due course returns control directly to the program in such a manner that the offending instruction is retried. In the case of other faults ECPROC puts a mark in the frame on the C-stack belonging to the procedure in which the fault occurred, and deposits a description of the fault in a standard place. It then executes a RETURN instruction, having first

modified the link in the C-stack frame so that control passes to a specific place in the first program segment of the faulting procedure. It is assumed that the programmer will have placed there a routine capable of dealing appropriately with all the faults that can occur. It is the further responsibility of this routine to remove the mark placed on the stack and this it does by making a suitable call on ECPROC.

The above description is incomplete since it can happen that, when ECPROC comes to mark the frame belonging to the faulting procedure, it finds that it is already marked. This will be the case if the fault occurred when an earlier fault was being dealt with. In these circumstances, ECPROC goes down the stack until it finds a frame that is unmarked. It marks this and then modifies the links on the stack so that, on the execution of the RETURN instruction, control passes to the procedure corresponding to the frame just marked, bypassing the intermediate procedures. Thus, if a procedure cannot handle a fault that has occurred within it, a fault is raised in the procedure that called it; if that procedure cannot handle the fault, then a fault is raised in its own caller, and so on.

It will be remembered that virtual memory traps cannot be dealt with when the master coordinator is running (see page 34). In order to avoid having the C-stack permanently resident, it is arranged that, when entry to ECPROC takes place from the master coordinator, space provided for the purpose in the process base is used for the C-stack frame. Room for three such frames is provided in every process base and it can be shown that this is as many as will ever be required.

Attentions are generated by a user typing a specific character on a terminal, this character being recognised for what it is by the process responsible for that terminal. Attentions cause compulsory transfer of control to a specified place in the procedure running when the attention is generated. The implementation is very similar to that used for faults.

Attentions have varying degrees of severity. They may be reset by a call to ECPROC. This call has to be supported by a permission capability of sufficient power to allow the resetting of the attention in question. The procedure in which user processes run initially, namely STARTOP (see page 42), has a permission capability which enables any attention to be reset. Command programs have a slightly less powerful permission capability and programs running under them less powerful capabilities still. If a procedure attempts to reset an attention with an insufficiently powerful permission capability, control returns to the caller of that procedure which can then in turn attempt to reset the

attention. Eventually, a procedure will be reached with a sufficiently powerful permission capability to be successful. In the case of an attention of maximum severity this will be STARTOP and the effect will be to terminate the user's session.

The Virtual Memory System

Real Store Manager. A process called the real store manager is responsible for bringing segments into the main memory when they are required and for deciding which segments should be swapped out in order to make room for them. There is a capability for every swappable segment in one of the master coordinator's capability segments. In the case of a segment that is loaded in the main memory the capability has its normal form. If the segment is not in the main memory, the capability has a special form and is then known as an outform capability. If a process attempts to use the segment, a trap will occur during the capability loading cycle when the outform capability is encountered. The trap will be reported to the master coordinator which will refer the matter to ECPROC. ECPROC will discover that the trap is due to a segment being out of the main memory and will send a message to the real store manager requesting that it be loaded. The sending process waits in ECPROC until a reply is received and then retries the instruction that failed. Since the real store manager will by now have loaded the missing segment and converted the outform capability into a normal one, the instruction will succeed. Conversely, when the real store manager causes a segment to be swapped out in order to make room for another segment, it converts the capability for the first segment into an outform capability, and in doing so invalidates any dependent entries that there may be in the capability store.

The disc address of a segment which is not in the main memory is recorded in the outform capability that represents it. There is in addition a data structure in which is recorded where each segment currently in main memory starts, where it finishes, and what its disc address is. On receipt of a message indicating that a virtual memory trap has occurred, the real store manager inspects this data structure in order to ascertain whether there is room to bring the missing segment in without swapping any other segment out; if so, it does this and updates the data structure accordingly. If there is not room, the real store manager consults an algorithm for instructions as to which segment or segments to swap out. Information is available about whether a segment has been written to while in main memory; if not, copying back to the disc is unnecessary and the operation of swapping out amounts to no more

than making the space occupied by the segment available for reuse. The operations of writing a segment to the disc and reading the segment from the disc are performed by a protected procedure called by the real store manager. This procedure has available to it a disc map whose management will be described below. The operation of the real store manager is held up until the disc transfer is complete.

The largest single segment that the real store manager is prepared to handle is 32K words long. It is possible to access larger segments by windowing. Windowing essentially provides a facility whereby a segment may be equated with part of another segment or, to put it more naturally, with part of a file. Sequential access to long files is obtained by opening a succession of windows on them. This is usually done by having two windows in existence at a time so as to provide the effect of double buffering.

The real store manager provides a number of miscellaneous facilities. One of these makes it possible for the programmer to make sure that at a particular moment the disc copy of a specified segment is up to date; if the segment is not in main memory, this implies no action on the part of the real store manager, but if it is in main memory and has been written to, a new copy must be made. This facility is primarily used in connection with file directories and similar objects. Another facility enables the programmer to cause a segment to be swapped out, for example, when he knows that the segment will not be required again in the near future.

As mentioned above, the real store manager maintains a data structure in which is recorded the disc address, size, and access particulars of every segment with which it is concerned. This table is updated from information received from the virtual store manager described in the next section.

Virtual Store Manager. The real store manager has the duty just described of managing the swapping of segments which are currently in use to and from the disc. Segments may be, and frequently are, shared between different processes. The condition for withdrawing a segment from the real store manager's regime is that no current process has a capability for it. This level of administration is the province of the virtual store manager.

Whenever it is required to issue a process with a capability for a segment, a message containing the long term name for the segment as used by the filing system is sent to the virtual store manager. The virtual store manager finds out whether the segment is in the currently active virtual memory, that is, whether a capability for it already

exists in one of the coordinator's capability segments. If not, the virtual store manager calls the real store manager to cause an (outform) capability to be constructed and inserted. In either case the virtual store manager returns to its caller the specifier of the capability in question.

The virtual store manager associates with each segment for which it is responsible a reference count which is simply a count of the number of processes which possess a capability for the segment. The virtual store manager increments the reference count by one whenever a new process is issued with a capability for the segment and decrements it by one whenever it receives a message to the effect that a capability is no longer required by one of the processes that was using it. This message does not require a reply, but if the result is that the reference count becomes zero, then the long term management system (see below) is communicated with to ascertain whether the segment is known to it. If not, then steps are taken to eliminate all knowledge of the segment from the system.

The real store manager must, of necessity, be permanently resident in main memory. The virtual store manager does not require to be permanently resident. In the original implementation of the CAP operating system, the functions of the virtual store manager and the real store manager were combined. Separating them reduced significantly the amount of material that had to be permanently resident.

Long Term Management

System Internal Names. None of the information held by the real store manager or the virtual store manager persists from one run of the system to the next. The management of persistent material is in the hands of a protected procedure known as the system internal names manager (SINMAN). System internal names are integers which remain associated with the objects they represent as long as those objects exist. For convenience, the system internal name of an object is taken to be the address at which the object begins on the disc.

In order to discuss the operation of SINMAN it is first necessary to consider the types of permanent object which the system supports. There are three of these as follows: segments, which are simply at this level blocks of uninterpreted information; directory segments, which relate text names to system internal names; procedure description blocks, which are templates for the construction of protected procedures and which contain the system internal names of the objects that will go to make them up.

Directories and procedure description blocks both have the property that the occurrence of a system internal name in any one of them is a sufficient reason for maintaining the corresponding object in existence. SINMAN's primary duty is to maintain a list of such names, together with the names of other objects that should also be kept in existence because they are in the PRL of some process in the system. Along with this goes the duty of deleting from the list objects that cease to be in one of the above categories. The list is known as the SIN directory. It is indexed by the system internal name and contains reference counts (showing how many references to each system internal name there are in the directories or procedure description blocks) together with information about the type and size of the object concerned. The SIN directory is updated when new objects are created, when capabilities for objects are inserted into directories or procedure description blocks, or when they are removed.

SINMAN has a secondary function which is logically independent of the primary function just described and which might perhaps have been better performed by a separate procedure. This is to issue capabilities for particular objects on the list when requested by a suitably authorised client to do so. Such requests are made by quoting the system internal name. Some capabilities, for example, those for segments of the ALGOL68C run time system, are used by many, in not all, protected procedures. It is quite possible, therefore, that when a process requests a capability for an object it already has one in its PRL. For each process an index is maintained relating entries in the PRL to system internal names. When presented with a system internal name and asked for a capability for the corresponding object, the first thing that SINMAN does is to consult the index; if the name of the object is there, it delivers the appropriate capability. If it is not, SINMAN sends a message to the virtual store manager asking for information with which to construct the capability. If the virtual store manager has no knowledge of the object, then it will ask the real store manager to generate the required information. In any case, the virtual store manager returns the information to SINMAN which proceeds to construct the capability, consulting the SIN directory to find out whether the object is a segment, a directory, or a procedure description block.

SINMAN deletes at once from the SIN directory without further formality any object of type segment whose reference count falls to zero and which is not in the currently active virtual memory. Objects of type directory or of type procedure description block may contain within themselves references to other objects. Before deleting them, it is

therefore necessary to scan them and decrement the reference counts of any other objects referred to. Even so, the effect of the deletion may be to leave one or more detached loops. The only way in which the existence of such detached loops may be detected and the space they take up recovered is by making use of some form of scanning garbage collector. Accordingly, a separate process has been provided which implements a garbage collecting algorithm and which runs slowly in the background. Its detailed action will be described on page 55.

As a process runs, it is likely to request from time to time the creation of new segments and its PRL will tend to fill up. A process can, however, lose access to segments for which it has capabilities in its PRL. This will happen, for example, if all capabilities held in capability segments belonging to the process and defined in terms of a particular capability in the PRL are overwritten by MOVECAP instructions. In this case, the slot in the PRL can be reused. However, the only way to find out that this is possible is to perform a garbage collecting operation. This is quite distinct from the garbage collection described in the last paragraph and is performed by a procedure known as PRLGARB. PRLGARB is entered automatically if the PRL becomes full, but its services can be asked for explicitly. It identifies all the slots in the PRL which are not referred to in any of the protection domains represented on the C-stack. PRLGARB has access to the index which relates entries in the PRL to system internal names and deletes entries from it as required; it also communicates by message with the virtual store manager to indicate that certain capabilities are no longer in use within the process.

Command Program and User Interface

It was stated on page 33 that there are a fixed number of user processes. These are created during system generation and are equipped with all the standard facilities required for users' processes to run. Initially each process runs in its own instance of a protected procedure called STARTOP. STARTOP has access to various message channels and a capability that enables it to access the master file directory. When the system is initialised, each instance of STARTOP sends a message to a process known as Initiator requesting work to do and then halts waiting for a reply. Initiator is in communication with the set of processes which manage the interactive terminals and, when a user types a particular character on an idle terminal, Initiator recognises the action as a request to log in. It is then in a position to reply to one of the messages from an instance of STARTOP, assuming that there is one

outstanding. The reply includes the number of the terminal concerned and STARTOP secures an enter capability for a protected procedure through which it can communicate with that terminal. It then issues a prompt to the user asking for his identifier. Armed with this, it enters a protected procedure called LOGIN, which requests the user to type his password.

LOGIN then checks that the user's identifier and the password just typed are to be found in the file of authorised users and retrieves from the same file the name of the command program to be entered. This will normally be the standard system command program. LOGIN returns control to STARTOP which retrieves from the master file directory a capability for the command program and also one for the user's file directory. STARTOP enters the command program, passing the latter capability as a parameter. The user is now in command status and anything that he types is treated as a command.

Some system processes run exactly as though they were user processes. The despooling process is one of these; it has a special program called DESPOOL that stands in the same relation to it as does the command program to an ordinary user. The despooling process is, however, activated automatically by Initiator.

There is available to everyone a protected procedure called RUN which permits a user process to activate Initiator directly. It is thus possible for a user process to start up an independent process which will run initially in a protected procedure for which the user process provides a capability.

The standard command program is based upon a particular view of the functions that such a program should perform. It will call commands whose names are typed by the user and pass to them the residue of the command line as an argument string. It is also capable of resetting any attentions that the terminal user may have created, thus enabling him to force a return to command status. Normally, when searching for the program to execute a command, the command program goes first to the user's file directory and then to the command library, but it is possible to alter this and include other directories in the sequence.

It was felt that, if the command program were always to decode the argument of a command, an undesirable rigidity would be imposed upon the format of command lines. It is, nevertheless, desirable to have some de facto standardisation and the following compromise was adopted. The command program enters the procedure that will execute the command that has been called, passing as arguments a capability for a segment containing the complete command line and an enter capability for a

protected procedure known as PARMS. PARMS provides a variety of facilities for command line decoding; it can search for positional and keyword parameters, perform decimal to binary conversion, and open files. It is entirely at the discretion of the writer of a program for executing a particular command whether or not he makes use of PARMS, but in practice it is rare for him not to do so.

It is often convenient for programs that execute commands to have their arguments passed to them in machine registers instead of in the form of character strings. Most such programs are, in fact, able to accept parameters in either way. A convention has been established whereby the content of a particular register indicates which method is being used.

On completion of a command, control returns to the command program and the user is once again in command status. It will be remembered that the command program is a protected procedure that was originally entered from STARTOP. When the user types a LOGOUT command, the command program is caused to execute a RETURN instruction which sends control back to STARTOP. STARTOP issues a suitable message to the terminal recording the fact that the session is ended. It then sends a message to Initiator requesting further work and, after performing a garbage collection of its own PRL, becomes dormant once more until it receives a reply.

CHAPTER 4

THE CAP FILING SYSTEM

From the point of view of a user at the console, the CAP filing system resembles other filing systems in that it contains the names of objects which the user wishes to retain for future use. Each user has his own user file directory and in it are entered the names of objects which are of three types: segments, protected procedures, and other directories. A system programmer, working on other parts of the operating system, sees the matter slightly differently. From his point of view the function of the filing system is to preserve capabilities for objects in such a way that they can later be retrieved. The implementer of the filing system has still another point of view. What are actually preserved are not capabilities themselves, but information from which capabilities can be reconstructed. This information is held in directory segments which are ordinary data segments. It consists of the text name of the object, its system internal name, and information concerning access rights. Thus the act of retrieving a capability is not, as the term might suggest, a simple extraction of the capability from a directory segment.

Many of the underlying mechanisms on which the filing system depends have already been described. In particular, procedure description blocks were introduced on page 40 in connection with the management of system internal names. A procedure description block is a recipe or template for the generation of an instance of a protected procedure. Generation takes place partly when the protected procedure is retrieved from the filing system and partly when it is first entered.

A procedure description block contains information about the number and size of the capability segments and workspace segments that the protected procedure requires. It also contains a list of system internal names of objects for which capabilities must be placed in the appropriate capability segments when the protected procedure is generated.

Protected procedures are created by a system protected procedure called MAKEPACK. It is necessary first to procure from SINMAN a capability for an empty segment of type procedure description block and to hand it to MAKEPACK together with the necessary capabilities. MAKEPACK then returns an enter capability - in fact, it is always an outform enter capability - for the protected procedure. There is no special privilege required for the use of MAKEPACK since it is impossible to cause a capability to be incorporated in a new procedure description

block without passing it to MAKEPACK as an argument, and this implies the legitimate possession of the capability.

DIRMAN

Directory segments are not accessed directly from outside the filing system, but are invariably accessed via instances of a protected procedure known as DIRMAN (directory manager). An instance of DIRMAN consists of the code for the management of directory segments, some local work space, and the relevant directory segment itself. If a user process has access to a number of directory segments then it will have an equal number of instances of DIRMAN; similarly, if a particular directory segment is available to a number of processes, then each process will have an instance of DIRMAN corresponding to it. From the point of view of the user, the function of DIRMAN is to retrieve capabilities requested in terms of textual names and to perform the converse operation of preserving, along with the textual name, capabilities passed to it. However, as explained above, what is actually preserved is not the capability itself but information from which the capability may be reconstructed. So that it may perform the above functions, DIRMAN is equipped with an enter capability for SINMAN, which performs the underlying operations. DIRMAN acts as a gatekeeper, checking the validity of the request, performing text name to system internal name translation, and verifying that the user is entitled to the access status that he requests.

The operation of preserving a capability for a segment proceeds as follows. DIRMAN is called with a text name and a capability as arguments. It calls SINMAN, giving the capability as an argument, and SINMAN consults the virtual store manager to ascertain the corresponding system internal name. DIRMAN then performs the operation of preservation by placing the text name alongside the corresponding system internal name in the directory segment, at the same time requesting SINMAN to increment by one the reference count for the system internal name. The need to maintain the integrity of the system against failure - in so far as this is possible - imposes certain constraints on the order in which these and consequential operations are performed. This subject is discussed later in the chapter.

For the converse operation of retrieval, DIRMAN is called with the text name for the capability and the desired access status as parameters. It searches the directory segment to obtain the system internal name corresponding to the given capability and checks that the requested access status is available. If all is well, DIRMAN then calls

SINMAN, this time with the system internal name as a numerical argument. After suitable transactions with the virtual store manager, SINMAN constructs the capability and returns it to DIRMAN for passing back to the original caller.

When DIRMAN is asked to preserve an enter capability for a protected procedure, it proceeds in a similar manner, except that what it puts in the directory segment alongside the text name is the system internal name of the procedure description block. When DIRMAN is asked to retrieve the object by being given the text name as a parameter, it discovers, on the basis of information that it receives from SINMAN, that the system internal name refers to a procedure control block and it delivers an enter capability for an instance of the protected procedure.

Objects of type directory are seen by a user process as instances of DIRMAN. When a process wishes to preserve a directory in another directory, it passes an enter capability for the instance of DIRMAN correponding to the first directory to the instance of DIRMAN corresponding to the second. DIRMAN discovers, again on the basis of information that it receives from SINMAN, that the enter capability is for an instance of itself. Whereas, in the case of an ordinary protected procedure, it would put in the directory segment the system internal name of the procedure description block, in this case it puts there the system internal name of the the relevant directory segment, along with the text name by which it is known. When, at a later time, DIRMAN is asked to retrieve the directory, it will deliver an instance of itself with the appropriate directory segment bound in.

Just as segment capabilities have access bits determining their access status, so do enter capabilities for particular instances of DIRMAN. There are five such bits, of which one gives permission to create entries in the directory. The interpretation of the other four will be given on page 49.

There are no restrictions on the way in which the system internal name for one directory segment may be preserved in another. The structures formed are, therefore, properly described as directed graphs rather than as hierarchies of any form. Consequently a user may have more than one naming path available to him for reaching the same object. If he wishes to follow one of these paths, the text that he presents to DIRMAN must consist of several components. DIRMAN will take the first component of the name and access its own directory segment to find the system internal name that corresponds. It will then go to SINMAN for the capability for the object concerned. This will turn out to be a directory segment and DIRMAN will repeat the procedure for the second component of

the name, using this directory segment instead of its own. This process will continue until all the components of the name have been used up. Retrieval of the object from the final directory reached will then take place in the usual way.

The Retrieval of Procedure Description Blocks

It would have been possible to arrange that, on retrieval of a procedure description block, a complete instance of the protected procedure that it describes would be immediately generated. This would have been analogous to the way in which an instance of DIRMAN is generated when an object of type directory is retrieved. For the following practical reasons this approach was not taken. A protected procedure is likely to contain capabilities for other protected procedures, some of which may only be entered if the course of control in the primary procedure goes in a certain way, for example, if an error of a particular kind occurs. To avoid waste of work, the construction of the complete instance of the protected procedure is deferred until it is known that it will actually be required. Accordingly, when a protected procedure is retrieved from the filing system, DIRMAN causes a correct enter capability to be placed in the capability segment of the process, but in the PRL it places a capability of segment type for the segment containing the procedure description block. When an attempt is made to use the enter capability as the argument of an ENTER instruction, the microprogram detects the inconsistency and a trap occurs. The trap handling routine (part of ECPROC) calls a system protected procedure known as LINKER. LINKER has a capability for SINMAN and it can also read the information in the procedure description block using the segment type capability for it in the PRL. It can therefore construct the three capabilities required to convert the capability in the PRL into a legal enter capability (see page 11). When it has done this, it returns control to the trap handling routine which completes the task. The work is divided in this way because the trap handling routine already has those privileges necessary to complete the construction of the enter capability and to place it in the PRL, and it is unnecessary to give them to LINKER as well. When the trap handling routine has finished its work, the ENTER instruction that originally gave rise to the trap is retried.

The capability for the segment containing the procedure description block placed in the PRL is in every way the same as a regular segment capability. The access bits are, however, used in a special way and to emphasize this are denoted by L, I, M instead of R, W, E. L gives authority for the LINKER to perform its task, I gives authority for the

contents of the procedure description block to be inspected, and M gives authority for them to be modified. Inspection or modification of a procedure description block is effected through the agency of MAKEPACK.

User File Directories and Master File Directory

It will be convenient from now on to refer to directory segments simply as directories or file directories. A user, of course, does not have direct access to these, but accesses them through instances of DIRMAN. Each user has his own user file directory and when he logs in to the system the process which is established for him is automatically equipped with an enter capability for an instance of DIRMAN through which he may access it. Capabilities for all user file directories are preserved in the master file directory. A user file directory will contain preserved capabilities and associated text names, and may include capabilities for other directories. If a particular user wishes to group certain of his files together in a way which has logical significance for him, he can ask the system to provide him with a new, empty, directory and preserve in it capabilities for those files. In the ordinary way he will preserve a capability for this new directory in his user file directory, but there is no obligation upon him to do this. It may be that the grouping he wishes to establish is strictly temporary and that he will wish the directory to go away when the current phase of his work is finished. Indeed, it is possible for him to create for short term purposes a complex network of directories which will vanish when he logs out, just as a scratch segment would do.

Since capabilities may be passed from one user to another, it is unnecessary to arrange that DIRMAN should accept instructions of the form "allow access to .BIN if user equals RMN." If a user decides that another user should be able to make use of one of his segments, all he needs to do is to place a capability for that segment in a directory - created for the purpose, if necessary - to which the second user also has access. The latter can then, if he wishes, preserve a copy of the capability in his user file directory or in some other directory. He does not need to give it the same text name. Capabilities for objects other than segments can be handed over in a similar way and so can software capabilities.

Access Control

Segment capabilities preserved in file directories contain three data access bits, R, W, E. It has already been mentioned that enter capabilities for particular instances of DIRMAN contain five access bits.

These will be denoted by C, V, X, Y, Z. The first gives permission to create entries in the directory; that is, if a process executes an ENTER instruction with an enter capability for DIRMAN as a parameter and with another parameter indicating that a new entry is to be created, an error will be signalled unless C = 1. The remaining four bits form the elements of an access vector which will be denoted by $\underline{a}$.

Associated with every entry in the directory are two matrices, the permission matrix denoted by P and the access control matrix denoted by A. The permission matrix has four rows and three columns. The access control matrix has four rows and three columns if the entry refers to a segment or a procedure description block, and four rows and five columns if it refers to a directory. The elements of the matrices are either 1s or 0s. As will be seen shortly, the matrices determine the access status that may be associated with a retrieved capability. The user specifies, when he preserves a capability, what he wishes the matrices to be. DIRMAN will prevent him from setting up the matrices (or from altering them after they have been set up) in such a way that the capability can be retrieved with greater access status than was associated with it when it was preserved.

When DIRMAN is used to perform an operation on an entry in a directory, a permission vector, $\underline{p}$, is computed by multiplying the access vector into the permission matrix; thus

$$\underline{p} = \underline{a}P$$

where Boolean arithmetic is implied. The elements of $\underline{p}$ each give permission for a particular operation to be performed. If the element is a 1, permission is accorded; otherwise it is not. The actions associated with the elements of $\underline{p}$ are as follows:

$\underline{p}_1$ gives permission to delete the entry

$\underline{p}_2$ gives permission to update the entry (that is, to make it contain a different preserved capability)

$\underline{p}_3$ gives permission to alter one or both of the matrices P and A

When DIRMAN is used to retrieve a capability for an object represented in a directory, an access code vector $\underline{v}$ is computed as follows:

$$\underline{v} = \underline{a}A$$

where again Boolean arithmetic is implied. If the capability being retrieved is a capability for a segment, $\underline{v}$ will have three elements and their values will be copied into the retrieved capability to form the access code; that is, they will become the values of the bits R, W, E. Similarly, if the capability is a capability for a procedure description

block, there will also be three elements in v and their values will be copied into the retrieved capability to form the access digits L, I, M. In the case of a capability for a directory, v will have five elements whose values will be copied into the retrieved enter capability for DIRMAN as the values of C, V, X, Y, Z.

In the above it has been assumed that the process calling DIRMAN wishes to receive a capability which has the maximum access status allowed by the matrices P and A. There are, however, occasions - for example, when the capability is to be passed on to another process or procedure - when maximum access status is not required. In this case DIRMAN may be called with a parameter specifying what the access bits should be. DIRMAN will place these digits in the retrieved capability, having first checked that the access they imply is not more privileged than the caller would have obtained if he had asked for maximum privilege; that is, it checks that no digit in the requested access code is a 1 when the corresponding digit in the maximum permitted code is a 0.

The mechanism by which the access bits in a directory capability are determined has now been explained and it remains to describe how the system is used in practice. The status associated with each bit in the access code are given by the rows of P and A. In the case of general matrices, therefore, no direct significance is to be attached to the bits V, X, Y, Z as far as the access bits in the retrieved capability are concerned. In some cases, significance can be attached to certain combinations of digits. One such case is that of directory capabilities preserved for the use of the ultimate human user of the filing system. In this case the following default matrices are used for segment type entries (data segments) and directory type entries respectively:

SEGMENT TYPE						DIRECTORY TYPE							
P			A			P			A				
1	1	1	1	1	0	1	0	1	1	1	0	0	0
0	1	0	1	1	0	0	0	1	1	0	1	0	0
0	0	0	1	1	0	0	0	0	1	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	1

A capability for a file directory preserved in the user file directory of its owner, in that of one of his close friends, in that of one of his friends, and in that of a member of the general public might, given the above matrix, have access codes as follows:

owner	1 1 1 1 1
close friend	0 0 1 1 1
friend	0 0 0 1 1
general public	0 0 0 0 1

The owner could, in fact, give himself the same privileges by using the access code 11000, but he would not then be able to refine the capability - by replacing 1s by 0s in the access code - in order to pass a lower degree of privilege to someone else.

When a user requests the system to create for him a new directory, he receives a capability for it with access code 11111. It is then left to the user to set up the access matrices, using the above default option or not as he sees fit. There is a similar arrangement for new procedure description blocks.

The use of the access matrices may be further illustrated with reference to a file directory known as .LIBRARY which contains procedure description blocks for protected procedures which either form part of the operating system or are provided as utility programs for the benefit of users. An example of the first group is MAKEPACK and an example of the second group is the standard text editor. There are three groups of users who require different degrees of privilege in relation to .LIBRARY. These are the program librarian who is responsible for overseeing the contents as a whole, the utility programmers who are responsible for maintaining the utility programs, and the system programmers who are responsible for maintaining the system programs. In addition, there is the command program through which the protected procedures in .LIBRARY are used.

The access matrices for system procedures and utility procedures respectively are set up as follows:

SYSTEM PROCEDURES

P			A		
0	0	1	0	0	0
0	0	0	0	0	0
0	1	0	1	1	1
0	0	0	1	0	0

UTILITY PROCEDURES

P			A		
0	0	1	0	0	0
0	1	0	1	1	1
0	0	0	0	0	0
0	0	0	1	0	0

Capabilities preserved in the user file directories belonging to the program librarian, the utility programmers, and the system programmers, and in the master file directory (for the command program) have access codes as follows:

program librarian	1 1 0 0 0
utility programmers	0 0 1 0 0
system programmers	0 0 0 1 0
master file directory	0 0 0 0 1

It may be verified that the privileges accorded in the several categories are as follows. The program librarian can make new entries in .LIBRARY and he can make changes to any of the access matrices. He has not the immediate power of deleting entries, but he can alter a particular access matrix in order to give himself this power and then delete the corresponding entry. This is an example of a device commonly used to prevent accidental deletion of an entry; the filing system will prevent a situation arising in which there is an entry in existence with no way of deleting it. Utility programmers and system programmers have the power to modify and test protected procedures in their respective categories and to install new versions. They cannot interfere with procedures that are not in their own categories, nor can they create new entries in the library. The command program can cause protected procedures to be generated and entered, but cannot access them or make use of them in any other way.

Disc Management

Practical aspects of disc management include space allocation, integrity control, provision for restart, and garbage collection.

A system process called Module is responsible for allocating disc space requested for new objects, for de-allocating that space when it is no longer required, and for altering the allocation of space to existing objects. It is also capable of writing its allocation map to disc when requested to do so by SINMAN; two separate areas of disc, used alternately, are set aside for this purpose. To avoid the danger of inconsistency, no allocation or de-allocation of space takes place while writing operations are in progress. Module is also responsible for checking the consistency of its allocation maps during system restart.

It is not possible to arrange that new versions of all file directories, procedure description blocks, the SIN directory, and the allocation map are written to disc simultaneously. There will thus inevitably be a period during which the versions on the disc are inconsistent and complications will ensue if a system crash occurs during this period. Care has been taken in the implementation that this inconsistency should be on the safe side so that, when the system is restarted, the information can be brought to a state of consistency with little effort and not too much loss of material. The disc versions of all

directories and procedure description blocks, the SIN directory, and the disc allocation map are required to satisfy the following rules:

1. Nothing must be recorded in the SIN directory as permanently existing that has not already been recorded in the disc allocation map as having disc space allocated to it.

2. Nothing must have its system internal name recorded in a directory or procedure description block unless that name has already been marked as permanently existing in the SIN directory.

It is the responsibility of the directory and system internal name management system to ensure that the above rules are observed. When the system stops abruptly, it may be found on restart that there is space allocated on the disc which is not recorded in the SIN directory, or that there is space allocated in the SIN directory which is not used by any directory or procedure description block. A process known as Restart is responsible for making the records consistent at the cost of possibly destroying some information. Restart automatically recovers any disc space that is rendered inaccessible from the master directory as a result of the forcing of consistency.

Observance of the above rules implies a certain degree of serialisation of the various disc transfers involved. Considerable efforts have been made in the design of the CAP system to reduce, by optimisations of various sorts, the degradation of the response given to users that such serialisation is liable to cause. For example, when disc space is first allocated for a segment, the version of the disc allocation map held in the main memory is updated, but no steps are taken at that point to ensure that the version on disc is up to date. This need not be done until a capability for the new object is preserved. It may be that, by this time, the disc version of the map has been updated for some other purpose, and a system of version numbers is provided whereby it may be ascertained whether or not this is so.

When a capability is preserved, it is necessary to ensure that the SIN directory is up to date on disc before the directory or procedure description block is updated. There will be no violation of the rules, however, if the updating of the file directory is deferred. Accordingly, control is returned to the user as soon as the version in main memory has been updated, and a message, calling for no reply, is sent to a separate process known as Ensurer, requesting that the necessary disc writing operation should be performed in due course. Similarly, when a user deletes an entry from a file directory, control may be returned at once

to his program, leaving the disc versions of the SIN directory and the disc allocation map to be updated later.

Occasionally RESTART detects an inconsistency in the data structures held on the disc that can only be the consequence of software error or of serious hardware malfunction. In this case RESTART will report the error and request the services of an expert to examine the state of the filing system. A consequence of the careful attention given to ensuring that the rules set out above are observed is that it is possible to be quite precise as to when an expert should be called. Since the hardware capability protection system makes it extremely unlikely that trouble occurring outside the filing system itself will interfere with the integrity of the filing system data structures, a call for an expert is very rare. When one is called, it is usually because hardware malfunction of the disc system has caused information to be written to the wrong place.

It was mentioned on page 41 that disc space can be lost during normal running of the system through directories becoming detached. It would not be very satisfactory to rely upon periodic restarts to recover this space. Accordingly, there is an asynchronous garbage collector that runs at regular intervals. This need deal only with directories and procedure description blocks, since ordinary segments are dealt with by the reference count system.

When the garbage collector begins a scan of the disc, it first makes a copy of the SIN directory as it exists at that moment. References in what follows to the SIN directory will be to this copy. The garbage collector goes through the SIN directory and notes all those system internal names which refer to objects of type directory, a term which will for the purposes of this discussion be taken to cover also procedure description blocks. It then puts a mark against the item corresponding to the master directory. It scans that directory for system internal names and puts a similar mark in the SIN directory against all names that it finds. It then ticks off the master directory as dealt with. The garbage collector proceeds to scan, in the same way as it did the master directory, each object of type directory whose system internal name is marked in the SIN directory, ticking it off when dealt with. It scans the SIN directory repeatedly until all objects of type directory have either been marked and subsequently ticked, or have not been marked at all. It is the latter that can be collected as garbage.

Since the garbage collector runs under time sharing while the system is operating in the normal way, it is necessary that it should be

informed when a new capability for a directory is created or when one is retrieved from another directory. SINMAN sends a nonreply message to the garbage collector every time this happens and the directories in question are marked as needing inspection. The end of the garbage collecting operation is reached when there are no directories in the SIN directory with marks against them and no messages waiting to be dealt with. At this point the garbage collector will have identified as garbage whatever was garbage at the beginning of its run, and may or may not have identified material that became garbage during the run. It is necessary that any directories that have been declared garbage should be scanned in order to decrement the reference counts of all objects referred to in them.

At the end of a garbage collecting operation, a valuable check on consistency may be made by verifying that the reference counts of the garbage directories themselves are all zero. If they are not, it indicates that there is something seriously wrong with the garbage collector or that a hardware error has occurred. This check was particularly useful during the debugging of the garbage collector. Since the garbage collector gives rise to a substantial amount of disc traffic, it is normally arranged to run in short bursts so that users are not disturbed. Two parameters determine how often the garbage collector is woken up and how many directories it will deal with each time. Typically these are set so that it is woken up once a minute and scans 10 directories. The practical result is that garbage is unlikely to persist for more than one or two hours before being collected.

CHAPTER 5

DISCUSSION AND CONCLUSIONS

Planning of the CAP project started in 1970. The research objectives formulated then and set out in Chapter 1 have been followed throughout with only minor changes in emphasis. The CAP computer and its operating system have been in use since 1976 and have survived the test of having a number of users from outside the CAP research group as well as from within it. An attempt will be made in the present chapter to summarize some of the lessons that have been learned.

Two fundamental decisions were taken at the beginning of the project. One was that segments would be swapped as a whole instead of in pages and the other was that a local rather than a global naming system would be used. Some of the consequences of these decisions will be discussed in the sections that follow. Later sections will deal with a number of specific problems. Some of these will be recognised as problems that face the designer of any operating system, irrespective of the computer on which it is intended to run; they are seen here in particular relation to the protection features of the CAP.

Segment Swapping and Its Consequences

In a conventionally paged and segmented system the unit of protection is the segment and the entries in the segment table correspond in a rough way to the contents of the capability store in the CAP. They are, however, software entities and the protection is enforced by the kernel of the operating system, which also plays an important role in the transfer of pages to and from the main memory. At the time the CAP project was being planned it did not appear straightforward to graft on to such a system a hardware capability mechanism of the type described in Chapter 1. Moreover, it was desired to be able to protect segments of any length from one word upwards, whereas in a conventional paging system the grain of protection is a page. It was accordingly decided that a hardware paging system would not be built. Instead, a system would be designed in which segments of normal size would be swapped as a whole and very large segments would be brought partially into main memory by a windowing process under the control of the programmer.

It is not easy to design a satisfactory algorithm for the space management of segments varying in size from a few words up to 32K words in length in the presence of swapping. Any evaluation of the CAP project must take into account the fact that it was intended to be an experimental model for a very large multipurpose time sharing system

serving many users. Such a system would have vastly more main memory than the CAP. As things stand the physical limitations of the CAP memory have influenced the design of the operating system and caused it to depart to some extent from what would otherwise have been desirable. This is inevitable in any experimental system intended to be put to practical use. When the project was planned it appeared probable that very large mass core memory would become available at a cost that would be within the project's budget. If this had happened, and if one had been fitted to the CAP as its main memory, the result would have been a computer functionally similar to, although slower than, the large scale systems that we had in mind. The present prospect is, of course, that very large semiconductor main memories will become economically possible. The effect of this will perhaps be to favor the system used in the CAP, that is, the transfer of segments as a whole to and from the filing system, as compared with a system in which individual pages are swapped to and from from a fixed head disc.

The problem of managing space in the main memory becomes particularly acute if a large number of small segments must be handled. Such problems are by no means confined to the CAP or to systems in which space is managed in a similar way. Analagous problems arise with paging systems in which it is found that excessive memory fragmentation and a high paging rate occur if a lot of segments in frequent use are of less than one page in length. Inevitably, there will be conflict between the desire of the system designer to structure his program in the most logical and elegant manner, and the need to control the proliferation of small independent segments. In the CAP small segments are sometimes made by subsegmenting a longer segment which is treated as a whole for swapping purposes. An example is to be found in the program for STOREMAN, given in Appendix 3, where new capability segments are created by the subsegmentation of a large segment, one such segment being given to each process for the purpose. This leads to complications in the PRL garbage collector and in one or two other places. A more fundamental consequence of the pressure to avoid small segments is seen in connection with directory segments and procedure description blocks. In each case using two segments instead of one would have led to a better structure. One of the segments would have contained the system internal names and would have been the same whether it was associated with a file directory or with a procedure description block. The second segment would have contained the data structure appropriate either to a file directory or to a procedure description block as the case might be. The fact that the first segment would have been the same in both cases would have resulted

in a simplification of programs such as Restart and Discgarb which, at present, must be able to access both directory segments and procedure description blocks with their different formats.

Problems associated with swapping and the shortage of space in the main memory influenced the decisions taken about procedure capability segments. It had been intended originally that all instances of a particular protected procedure, in whatever process they occurred, would share the same P capability segment. This is clearly not possible if the capabilities in that segment are defined relatively to those in the PRL of a particular process. Instead, they must be defined in terms of those in the PRL of the process next above in the hierarchy. At the top level this will be the MRL. It was proposed that, when an enter capability for a shared protected procedure occurred in the PRL of a process, it would contain a special pointer indicating that the P capability segment was to be obtained by going to a higher level in the hierarchy; when the capability was evaluated a corresponding pointer would be inserted in the ancillary field in the capability store. This feature is, in fact, implemented in the CAP microprogram and is used in the particular case of ECPROC. It is not used in other cases because it proved too complicated to manage virtual memory traps properly. In particular, complicated investigations in the various PRLs and capability segments would have been necessary to distinguish between traps from various causes. The system as designed suffered from the further disadvantage that it would have been necessary to designate in advance those protected procedures that could be shared in the above sense.

Implementation of shared P capability segments would have been made much easier if there had been enough main memory for the code segments of a procedure to remain resident as long as it was in the active virtual memory. Capability management would then have been in the hands of LINKER and complications associated with swapping would have been avoided.

The Consequences of Local Naming

In the CAP system the segments associated with a protected procedure are referred to in terms of the offsets of their capabilities in the various capability segments belonging to the process. Thus every instance of a protected procedure has its own address space or, in other words, names are local to an instance of a protected procedure. One consequence of this is that it is not possible to pass directly from one protected procedure to another a data structure that extends over several segments, since the intersegment pointers embedded in the segments would

not be valid in the second procedure. This has not proved to be a limitation in practice. The majority of the data structures that it has been necessary to pass have all been accommodated in a single segment which could be passed without difficulty by passing a capability for it. If an occasion does arise to pass a large multisegment data structure, the natural thing to do is to bind it into a protected procedure through the code of which it can be accessed. This is in accordance with the principle that any structure which is complex enough to need several data segments is also worth protecting in order to hide its details from its users. Thus, what might at one time have been seen as implying a limitation in the CAP system, can now be seen as being in accordance with modern programming doctrine.

The fact that the same capability is known by different names in different protected procedures means that, given the way in which the capability unit is organised, the capability store may contain a number of copies of the evaluated capability for the same segment. Since some segments, such as those containing parts of the ALGOL68C run time system, are used by almost every protected procedure, this is a serious matter as far as utilisation of the space in the capability store is concerned. The provision made for 64 locations has been found to be adequate rather than generous as had been expected. If it had been possible to bring into full use the facility mentioned in the last section whereby instances of protected procedures could share the same P capability segment, the pressure on space in the capability store would have been somewhat reduced.

In the design of all capability systems a tension is felt between the efficiency - in the short term - of allowing capabilities to be copied and passed around freely, and the convenience from the managerial point of view of keeping track of them. Efficiency demands that there should be no overheads attached to the operations of moving or copying capabilities, while good management - especially memory management - is facilitated if a record is made when a capability is moved or copied. Indeed, if no records at all are kept, the resulting gain in efficiency is likely to be negatived by the cost of performing extensive garbage collecting operations. The records usually take the form of reference counts indicating how often a capability is referred to. When the reference count falls to zero, the slot containing the capability can be reused. The use of reference counts does not, however, necessarily allow garbage collection to be dispensed with altogether since, in certain circumstances, circular chains of references may be formed.

In the design of the CAP system a compromise was made. It was realised that the passage of capabilities between protected procedures in the same process would be very much more frequent than the passage of capabilities between processes and that it would be worthwhile making the former very efficient. This is why such instructions as MOVECAP and REFINE simply overwrite the previous content of the relevant capability slot without preserving any record of the operation. On the other hand, the virtual store manager maintains reference counts of the number of PRLs containing capabilities for segments that are in active use, and with their aid it can perform its managerial function without garbage collection being necessary. From the point of view of the total amount of garbage collection to be done, the compromise adopted is very satisfactory. The scale of each PRL garbage collection performed is small and it is reasonable to hold up the work of a process while it is being done.

While the system as implemented certainly works, it is possible that an alternative scheme in which reference counts were maintained on a system wide basis would be preferable. It would be necessary to accept the complication in MOVECAP and similar instructions that this would entail, together with the fact that the residual garbage collection necessary would be on a system wide scale and therefore somewhat time consuming. In return it would be possible to design the system so that addresses had a global rather than a merely local significance. The problem about the passage of enter capabilities between processes mentioned in the next section would not arise. A. J. Herbert has made a proposal for a new CAP protection system of this type. It would be based on the use of a central capability list and would run on the same hardware as the present system, but with a new microprogram. Appendix 1 contains a reprint of the paper by Herbert in which the proposals are set out in detail. It is hoped to implement this new system and it will then be possible to evaluate its practical advantages and disadvantages compared with the present system.

The Control of Subprocesses

At the time the CAP project was being planned, it was a requirement, generally accepted by all those concerned with the design of very large time sharing systems, that it should be possible for the management of such a system to allocate resources to a user who would be able to reallocate them among his own clients. For example, the teacher of a course on programming should be able to reallocate to his students the resources allocated to him, and have the computer control the use

that they make of them without the system management being involved in any way. There are two ways in which this may be done. One is to make it possible for a subordinate operating system - or rather a subordinate coordinator, since the central filing system would be available at the lower level - to run under the main operating system. The other is for all the processes to be controlled by the same coordinator, but for the coordinator to be so designed that it can recognise a hierarchical relationship between the processes and act accordingly.

In the case of the CAP the first approach was taken. Provision was made in the implementation of the ENTER SUBPROCESS and ENTER COORDINATOR instructions for a hierarchy of coordinators to be established under the master coordinator. Considerations of elegance and generality dictated that the hierarchy should, in principle, be of indefinite depth, although it was realised that in fact one, or at very most two, levels below the main system would be all that would ever be required. When activating a subprocess, the coordinator at a particular level would pass to it the capabilities that it needed. Whenever an interrupt took place the master coordinator would be entered. It would consult its scheduling algorithm and decide which process to activate; this might or might not be the process that had been interrupted. Similarly, subordinate coordinators when entered would consult their own scheduling algorithm and decide which of their subprocesses to activate. In this way they could share among their subprocesses the processor time that had been allocated to them.

Whenever an attempt is made to run one operating system or coordinator under another, a major difficulty encountered is that of giving users at the lower level access to the main filing system. The difficulty lies in providing sufficiently powerful, but at the same time efficient, facilities for passing messages from one level to another. It was first encountered in the early days of time sharing when attempts were made to run time sharing systems under conventional batch type operating systems. Some of the most successful time sharing systems operating in this way provided a filing system of their own entirely separate from that associated with the batch operating system.

That the difficulty just described was particularly acute in the case of the CAP became apparent when it was realised that, for reasons explained in detail in the next section, the passing of an enter capability from a process to a subordinate process would result in a loss of protection. It was thought at first that the difficulties could be circumvented if the superior process were to pass, not the enter capability, but capabilities for code and data segments with the the aid

of which a new enter capability could be constructed at subordinate level. However, this is not really practicable since the subordinate user cannot have passed to him any system procedures and data structures that will help him with the work and he is in almost as exposed a position as a user presented with a bare machine. It must be concluded, therefore, that the inability to pass enter capabilities is fatal to the successful operation of a hierarchy of coordinators in the manner described above.

At one time a plan was drawn up for making a more modest use of the facility provided in the microprogram for one coordinator to run under another. It was proposed that the major processes of the system, such as those for peripheral handling, real store management, and virtual store management would be directly responsible to the master coordinator, whereas user programs would be responsible to a subordinate coordinator, itself responsible to the master coordinator. It was felt that this would be a practical way of providing separate scheduling for system processes and user processes. It turned out, however, on closer examination that, despite the apparent elegance of the notion, all that would be achieved would be an increase in the amount of trusted software and a lengthening of the chains of communication between user processes and system services. The problem of the filing system was also encountered. If the filing system were defined at the upper level it would be difficult to make it available to users in an efficient manner. On the other hand, if it were defined at user level, it would be unavailable to the system processes. These would then require a rudimentary filing system of their own, especially if they were to be able to bootstrap themselves into operation at the beginning of a run. In the face of these considerations the decision was taken to have one level of coordination only.

In the next section the consequences of passing enter capabilities from one process level to another are examined in detail and it is shown how the loss of protection arises.

Subprocesses and Enter Capabilities

In Figure 5.1 coordinators are indicated by square boxes and other procedures by round boxes. Lines connecting boxes indicate a route that a process may traverse (in either direction) as it runs, first in one procedure and then in another. Code in certain of the boxes may be identical and in practice would be shared. Processes running in boxes enclosed by one of the ovals come under the immediate jurisdiction of the same coordinator and may be said to run in the same domain of

coordination. The reason why a loss of protection results if enter capabilities are allowed to be passed across the boundaries of domains of coordination is connected with the existence of an external interrupt system and the fact that when an interrupt takes place a coordinator at any level stores in space available to itself the information necessary for restarting the interrupted process.

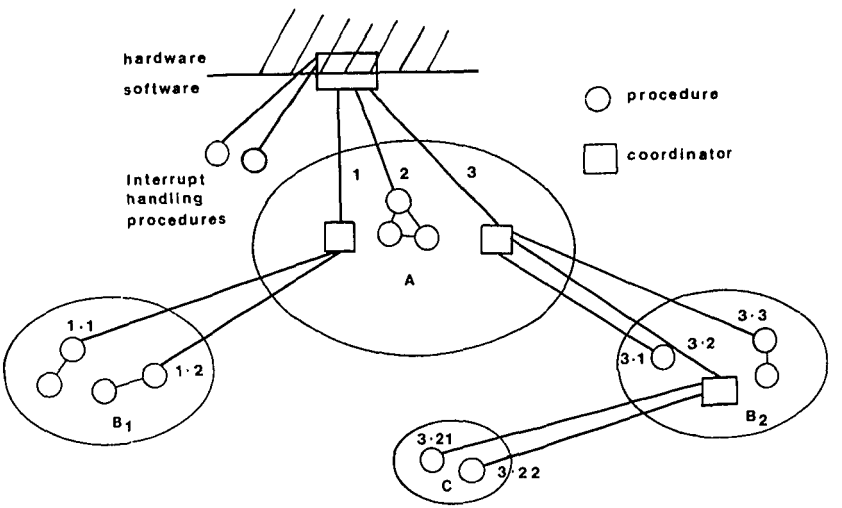


Figure 5.1 Domains of Coordination and Domains of Protection

Suppose the writer of a routine intended to be run in a given domain of coordination has available to him an enter capability for a procedure associated with which there is information that must be protected. He may use that enter capability for the purpose of entering the procedure, but this will not enable him to do anything with the protected information that the writer of the procedure has not provided for. Eventually control will come back to the original routine with the authorised action completed, or the authorised information passed. Everything happens within one domain of coordination and no breach of protection is possible. Suppose, however, the writer of the routine that we are considering decides to make it a coordinator and causes it to pass to a subordinate process, operating in another domain of coordination, the enter capability in question. This enter capability is used within

the second domain to enter the protected procedure. While running in the procedure, the subordinate process is open, as all processes are, to the hazard of arbitrary interruption as a result of the occurrence of an external interrupt. When this happens, the contents of the processor registers and all necessary information for restarting the process are placed on the associated process base by the coordinator to which it is responsible. Since, at the moment of interruption, the process was running in the protected procedure, some of the information placed there may be confidential to that procedure. It has, however, been placed on a process base accessible to a coordinator in a higher domain of coordination and has, therefore, become freely available to the superior process.

The above may be made clearer if a special case is considered. Suppose there is a procedure which has access to a table containing detailed information that is not supposed to be generally available. The procedure is designed to produce a statistical report based on information in the table and pass this report back to a process that has entered it in the proper way by means of an enter capability. Suppose that a user who has access to this enter capability wishes to attempt to get access to the entire table. He would proceed as follows. He would first write a routine to run as part of a process in his primary domain of coordination. He would design this routine to act as a coordinator; this he is entitled to do since any user can write a coordinator and set up a subsystem. He would then write another routine which would run in a subordinate process under the coordinator and would repeatedly enter the protected procedure, requesting information in a legitimate manner. He would expect the subordinate process to be interrupted frequently, as all processes are; on some occasions it would be interrupted while running in the protected procedure and information concerning the latter would be placed in a segment accessible to the coordinator. Since he himself is responsible for writing the coordinator, he can arrange that, whenever it is re-entered after an interrupt, the information relating to the subordinate process is examined in the hope that it will contain means of accessing the confidential information that he is seeking. His progress will not, of course, be straightforward since he will be groping in the dark and may not be able to recognise the significance of the information that he uncovers. Nor does one wish to stress the breaching of protection in the context of deliberate wrongdoing. We are more concerned with the way things may go wrong as a result of a software error or hardware failure. Anything could happen if pointers and other items became available in illegitimate ways.

The difficulty just explained arises because the status of a process is stored in space accessible to the coordinator. A way out of the difficulty would be to arrange for the status to be stored centrally. This, of course, is what happens in the alternative approach to subsystem implementation in which all processes are handled by a unique system coordinator.

The Address Argument Validation Problem

This problem is well known to those concerned with protection in operating systems. Suppose a process enters a procedure and in doing so passes from one protection environment to another. Suppose further that it carries with it the address of an object as a parameter. It is a requirement for proper protection that the object shall be accessible in the original protection environment, namely, that of the caller. It is necessary to guard against the possibility that the address may be valid in the protection environment of the called procedure (and hence give rise to no trap when it is used) and yet be invalid in that of the caller. Obviously the check of validity cannot be made before the change in environment takes place, since the calling procedure is not to be trusted. It cannot be made by the operating system during the calling process, since there is no way in which the system can know whether what is being passed is a numerical argument or an address. The validation of the address must, therefore, be performed in the called procedure and sufficient information to make this possible must be preserved about the protection environment of the caller.

Validation of addresses is complex and lengthy if performed by software and designers of large computer systems now usually provide hardware support. In the Honeywell 6180 computer on which MULTICS runs there is a convention that addresses should be passed from one ring of protection to another by making indirect reference to words placed by the calling procedure on the stack. When performing the indirection, the hardware automatically checks that the address is not being used in a ring with a lower degree of privilege than the ring of the caller. The ICL 2900 series of computers also uses rings of protection. Here the check is made by means of a special validating instruction and there are conventions regarding its use which the writer of a procedure designed to be called from other rings must follow. In either of these systems, provided that the called procedure is correctly written, no violation of protection can occur as a result of error or malice on the part of the writer of the calling procedure.

In a capability system no threat to protection is raised by the passage of capabilities from one procedure to another, since a calling procedure can pass a capability only if it has access to it itself. A threat does, however, arise if an attempt is made to pass from one procedure to another a data structure stretching across more than one segment. It has already been pointed out that, in the CAP system, it is not possible to do this for reasons that have nothing to do with protection, but which are a consequence of the use of a local naming scheme. It follows that in the CAP there is nothing corresponding to the address argument validation problem.

The reason why multisegment data structures are dangerous is that they contain pointers from words in one segment to words in another. A pointer will consist of a bit pattern specifying a capability for the second segment and an offset in it. If the capability is specified uniquely by a global name, then no problem arises. It is more likely, however, that, in the interests of efficiency, the bit pattern will identify the capability by pointing to where it can be found. For example, in the case of a computer in which capability registers are referred to explicitly, a capability may be identified by giving the number of the register in which it stands. A loss of protection will occur if the calling procedure specifies the wrong register. In the Plessey System 250 this cannot happen, since the capability registers are referred to by means of tags in the instructions (cf. index register tags) and these tags are never changed. In other systems a validity check would be necessary. The price paid for not needing such a check in the Plessey System 250 is that in addition to it being impracticable to pass multisegment data structures from one protected procedure to another, it is also extremely difficult to use them within a single protected procedure.

Modularity

A consequence of the CAP's protection system which was not fully appreciated at the outset is the high degree of modularity which it enforces on the software. The unit of modularity is the protected procedure. The procedure description block which defines the protected procedure contains a complete and unambiguous specification, not only of the code and data belonging to the procedure, but also of the environment required for it to operate correctly.

The nature of the modularity that this achieves is indicated by the statement that protected procedures may, in principle, be written in any language that takes the programmer's fancy. In practice, all the

procedures of the operating system proper have been written in ALGOL68C, but there are one or two procedures used for text editing and such purposes that are written in BCPL. The interface between the procedures has quite deliberately been defined at a very low level, namely at machine code level. In consequence no type checking is provided across the boundary. This has not, in practice, been a serious drawback, no doubt because of the simplicity of the interface.

The fact that the modularity extends to the environment as well as the code makes it unusually easy to make structural alterations to the operating system. In particular, it is very easy to substitute a service process for a service procedure. One first sets up a new process with the same environment as that associated in the original process with the protected procedure providing the service. One then needs to make only a small change to the calling sequence. The details of the way this is done for procedures written in ALGOL68C are explained in Appendix 2. The ease of transfer of programs from the procedural to the message mode of call exemplifies the duality of operating system structures drawn attention to by Lauer and Needham (1978). This duality is often obscured by an excess of implicit assumptions about the environments in which programs may run.

The Minimum Privilege Principle

The minimum privilege principle asserts that, at any time, a running program shall be given those, and only those, privileges of whatever kind that are logically necessary for the correct performance of its task. In many computer systems the principle is either inapplicable or the architecture imposes severe limits on the extent to which it can be observed. In the case of the CAP and other systems that emphasize protection, there are no such limits and it is a matter for debate how far one should go. It is clear that, in the case of material that is sensitive in the sense that its corruption or leakage would have serious consequences, a strong case can be made out for following the principle very strictly, even to the extent of subdividing data structures to a fine degree. At the same time it must be recognised that in any system, however effective the hardware support, there are bound to be overheads involved in changing the domain of protection and this consideration will set a limit to what a practical designer is prepared to do. The case for following the principle strictly in the case of data whose corruption or leakage would lead merely to minor inconvenience must be made on the grounds of facilitating the development of the programs and of making the system more rugged, in the sense that the term was used in Chapter 1. In

a tightly protected system, an error of any kind is likely soon to lead to a protection trap and, even if it does not, the source of any observed corruption of a part of the system can readily be located by examining all protected procedures that have capabilities for it. The system is in consequence more resilient to bugs whose effects do not show up until it has been running for some time, to hardware incidents, to accidents of all kinds, and to unexpected and possibly dangerous side effects of any modifications that may be made to the system while it is in service.

A further reason that can be advanced for adopting the minimum privilege approach in a strict form is that it makes it easy to lay down guide lines for members of the team working on system development. It is not obvious what other guidance could be given to designers who must decide, among other things, which capabilities a certain procedure should have. Associated with minimum privilege is the principle of data abstraction; in CAP terms this means that, if a data structure is required to be accessible in a number of different domains of protection, then it should be managed by and only accessible through a protected procedure defined for the purpose.

In order that the minimum privilege principle may be correctly applied, it is necessary to ensure that any tract of code which requires special privileges should be as small as possible. This consideration will be the major one in deciding on the scope and function of the modules into which the system is divided. An example in the CAP is furnished by the protected procedure MAKEPACK, which has important privileges in relation to SINMAN. MAKEPACK was kept small by omitting everything concerned with the user interface. The omitted material was included in another protected procedure (MAKEPDB) which requires no special privilege of any kind. By the same token, it is desirable to arrange that individual capabilities do not give a wider range of privilege than is likely to be needed on any particular occasion. This is particularly the case in relation to enter capabilities. If a protected procedure is capable of performing a variety of functions, not all of which will normally be required by a particular caller, some means must be found whereby they can be separately authorised. In the case of the CAP this is done by the use of software capabilities or by the use of access bits in enter capabilities, which provide what Saltzer and Schroeder (1975) refer to as separation of privilege. In the basic parts of the CAP system the separation of privilege was taken a long way. For example, the privilege to create a capability is distinct from the privilege to modify an existing one. It could be argued that this is going too far, because the consequences of abuse of either privilege

would not appear to differ materially. It was difficult, however, to be quite sure that this was really the case, and separation was thought to be a wise precaution.

A static audit of the protection structure of the CAP was carried out by D. J. Cook (Cook, 1978) and, perhaps not surprisingly, a good deal of overprivilege was found to exist. Some of this was connected with SINMAN and it may be worth while explaining how this came about. The early design of SINMAN was quite simple, but as time went on additional facilities were added without anyone having a proper appreciation of what was happening. The consequence was that the privileges were not properly separated. When the full significance of this finally became apparent, all the programs that used SINMAN had been successfully debugged and there was consequently little incentive for the design team to put things right. Tnis was, perhaps, a pity because the additional ruggedness that would have been imparted to the system if privilege had been properly separated would have been worth having. The experience just related emphasizes the fact that the designers' understanding of a program grows as the program is developed. It is rarely possible to produce a really well structured program on the first attempt.

Another source of overprivilege arose from the existence of the G (global) capability segment. None of the capabilities in that segment are such that their misuse could do the system any harm, but it is nevertheless true that some of them are surplus to the requirements of many programs. The purist would say that these should not be put in the G capability segment and he would perhaps be right.

It is interesting to note that if the aim of the CAP project had been to study security rather than protection, the presence of some of the G capabilities would have been more serious. For example, one of them enables the current process to ascertain the name of the user on whose behalf it is running. It would thus be possible for a dishonest system programmer to write a program that would discriminate against a particular user. This would be less easy if the capability giving access to the user's name were only accessible to code that really required it. G capabilities have something in common with global variables which it is now considered good programming practice to avoid as far as possible. If we were to produce a new version of the CAP operating system, we would certainly make less use of the G capability segment.

Restricting the Use of Capabilities

It would sometimes be useful if restrictions could be placed on the use of capabilities in a way that is not possible in the existing CAP system. Consider, for example, the capability needed by the command program for resetting certain classes of attentions. It will be recalled that software capabilities for resetting certain attentions are freely available; with their aid the user is able to build into any subsystem that he writes means whereby an operator at a console can intervene to divert the flow of control from one part of the subsystem to another. Certain attentions, however, need more powerful software capabilities to reset them. The command program possesses one of these and the effect of its use is to return the user to command status; that is, it aborts his program without terminating his session. An even more powerful software capability for resetting attentions is possessed by STARTOP and this, when used, terminates the user's session.

At first sight it might appear that no harm would result if the two powerful capabilities just referred to were made generally available, since by misusing them a user could do harm only to himself. However, they might become incorporated into utility routines or into other routines passed from one user to another and this could be embarrassing. For example, by using the software capability normally used only by the command program, it would be possible to write a routine containing a loop from which the user could only escape by terminating his session.

As the CAP system stands at the moment, the software capability for resetting attentions used by the command program is placed in its P capability segment when it is generated. If instead the capability were passed to it as a parameter when it is entered from STARTOP, it would be possible for a user to write a command program of his own that would take the place of the regular one. All he would have to do would be to ensure that the entry in the file containing users' particulars (see page 43) was changed accordingly. However, this would enable a user to obtain a copy of the capability for resetting attentions which he could then preserve in the filing system and incorporate in other programs. It would be possible to prevent him doing this by passing, not the capability itself, but an enter capability constructed by MAKEENTER for a two line protected procedure equipped with the capability which would reset the attention. Since such an enter capability cannot be preserved in the filing system this would provide a way round the difficulty. A better solution, but one involving a radical change to the system, would be to have a bit in a capability which indicated whether it could be preserved or not.

Further thought, however, suggests that a more generalised implementation might be considered. All capabilities of whatever type would have a set of bits in them whose integrity would be maintained by the microprogram in the same way as other bits in the capability; the interpretation of these particular bits would, however, be entirely a matter of software convention. It would be rather as though all capabilities had a software capability incorporated within them. There is a parallel here with the system used in connection with the entries in a file directory and described in Chapter 4. What is there called the access control matrix is concerned with the uses that may be made of the object for which a capability has been preserved and the permission matrix is concerned with operations that may be performed on the entry itself.

The remarks that have just been made have a bearing on the problem of the revocation of capabilities. There are issues here that go deeper than mere questions of implementation and depend on the view taken of the nature of capabilities. In the CAP, a user who has been given or has otherwise acquired a capability is free to preserve it in the filing system. No record is kept centrally. The original owner of the capability has, therefore, no way of revoking it. Nor can he destroy the object to which it refers since the filing system will maintain it in existence as long as there is a capability for it anywhere in the system. This is consistent with the view that a capability is a ticket which is unforgeable and which will not lose its validity.

If the system were to keep a central record of the whereabouts of all capabilities it would, in theory, be possible for them to be revoked by being marked as invalid. The overheads of having such a record would, however, be prohibitive. If the destruction of objects were to be permitted without all capabilities for them being marked as invalid, then means would have to be found to prevent the capabilities being reused for new objects; in practice, this means giving them unique identifiers which are never re-used. This can be done, but it adds substantial overheads to the management of capabilities.

There is room for debate about the circumstances in which the revocation of capabilities is necessary, or indeed whether it is necessary at all. The provision of capabilities which could be handled by a user during his current session but not preserved would, perhaps, meet some requirements without imposing the substantial overheads mentioned above. A user wishing to use a revocable object would need to request a capability for it on each occasion he logged in. The owner of the capability would be free to determine from time to time the

conditions on which the request could be granted. Once having granted the request, he could not go back on his action, but the capability would lapse at the end of the user's session.

Tagged Capability Architecture

In the CAP, capabilities and data words are strictly segregated into capability segments and data segments. Various proposals have, however, been made for computer designs in which each word is stored in memory with an extra tag bit to indicate whether it is a data word or a capability. The advantage of doing this is that by mixing data words and capabilities in the same segment it is possible to reduce the number of small segments that have to be handled; for example, the capabilities required by a procedure can be put in the same segment as the code of the procedure. The capabilities themselves would still be capabilities for segments and would carry access bits indicating read, write, and execute access.

In such a computer, enforcement by the hardware of the rule that capabilities cannot be counterfeited is effected in the following manner. When one of the arithmetic registers is loaded from the memory, the tag bit associated with the word in the memory is ignored; when the content of an arithmetic register is copied into memory, the tag bit is forced to be a 0. When an attempt is made to load a capability register from the memory, a check is made by the hardware and an error signalled if the tag bit is not a 1. As in other capability systems, it is necessary that a sufficiently trusted system procedure should have the power of creating capabilities; in this case it would be the power to execute a machine instruction that would convert a tag bit from a 0 into a 1. The execution of this instruction would be controlled by a special kind of capability in much the way that the execution of input and output instructions is controlled in the CAP computer. The possession of this special capability would be confined to the highly trusted procedure just mentioned.

Experience obtained in designing the CAP operating system has shown very clearly the importance of being able to relax in a controlled manner the rules under which protection is enforced; for example, the procedure responsible for performing housekeeping tasks necessary after the occurrence of an interrupt is given a D-type capability for a segment containing capabilities that elsewhere is accessed by means of a C-type capability (see page 3). The tag system in the form just described would not provide the necessary degree of flexibility. The reader will perhaps be able to devise ways in which it could be provided without destroying

the purity of the tagged architecture concept. We will not pursue the subject here since we feel that there is a separate objection to the use of tagged capabilities that in itself is sufficiently strong to make their use unattractive in a system of the CAP type.

The methods for the management of capabilities that are used in the CAP operating system depend on the fact that capabilities are confined to capability segments. In a tagged architecture they would be scattered throughout the memory and, as far as we can see, their management would involve extensive scans through memory, in particular, when they were being preserved in the filing system and when the segments to which they refer were about to be destroyed or removed from main memory. It is possible that this is not an insuperable objection and that fresh insights will lead to proposals for a tagged architecture that would be both elegant and efficient. It should, however, be added that any such architecture would have to meet an objection that can be raised against all systems of tagging for whatever purpose the tags are used, namely that they imply an increase in the length of the memory word above what would otherwise be necessary. Since the cost of memory is likely in the future to dominate the cost of processors, this means a proportionate increase in the total cost. A designer who has to consider the overall economy of his design and who has competing requirements to reconcile might not be prepared to incur this cost. He would be more likely to do so if it could be shown that reducing the number of small segments that have to be handled would lead to a compensating economy in memory usage.

We may perhaps add that, in designing the CAP computer and its operating system, we and our colleagues were anxious to demonstrate the possibility of producing a tightly protected general purpose operating system that would run on a computer with no privileged mode. The CAP philosophy was to avoid anything in the nature of a kernel running in a privileged mode, except in so far as the microprogram might be said to play this role. It is apparent however, that in a different context - for example, that of a system intended specifically for transaction processing, with perhaps more emphasis on privacy than on system ruggedness - the same ideas might well find a different implementation, both in terms of hardware and in terms of software. One lesson in particular has been learned. That is the high importance that is now to be attached to protected procedures and their generalisations, by whatever names they may be known. It is clear that future designers of integrated hardware and software computer systems will need to consider very carefully what form of support to give to such objects.

APPENDIX 1

A HARDWARE-SUPPORTED PROTECTION ARCHITECTURE*

A. J. HERBERT
Computer Laboratory
University of Cambridge, England

Protection Kernels

In both HYDRA [1] and CAL-TSS [2] a large body of code, the kernel, is responsible for protection mechanisms. The kernel provides software emulation of a machine which includes high level protection functions in its repertoire of operations. Software emulation of simple protection operations is slow and cumbersome; a complex function, such as moving between protection domains, is considerably slower, perhaps by a factor of 1000, than a simple hardware procedure call. Software kernels are characteristically large and written in machine code. The inefficiency of kernel primitives encourages the implementation of complex compound functions within the kernel to cut down on the number of calls to it. All of the above factors increase the complexity of the kernel and lead to a corresponding rise in the probability that it contains errors.

The complexity of the kernel can be reduced by breaking it down into a basic kernel which provides the bare bones of the kernel mechanisms and a high level kernel which implements more complex functions in terms of the primitives of the basic kernel. The high level kernel has the use of the protection mechanisms of the basic kernel, increasing its ruggedness. There are two major considerations in this approach: firstly, identifying primitives that belong to the basic kernel and, secondly, reducing overheads to a minimum within it.

The first problem is a matter of careful design and philosophy; the second may be alleviated by making the basic kernel part of the hardware of the machine. Hard-wired protection mechanisms are traditionally simple in nature - the inspection of access control bits and addressing limits. In a microprogrammed machine with an adequate supply of microprogram memory, it is possible to consider using

---

*This appendix is a copy, with minor revisions, of a paper first presented at "2ème Colloque Internationale sur les Systèmes d'Exploitation," IRIA-Rocquencourt and published in "Operating Systems," edited by D. Lanciaux, North-Holland Publishing Co., 1979.

microprogram control of protection hardware to implement a sophisticated protection system.

If individual facilities in a microprogrammed kernel are long in terms of numbers of instructions obeyed in providing them, there will be only small gains in efficiency. As the range of facilities provided by the kernel increases so does the size and complexity of its code, increasing the risk of errors and poor reliability. The kernel should be compact and its operations should be simple if it is to be successful. It should be designed independently of considerations of the detailed design of higher levels of the system and should be envisaged as a self-contained protection system in its own right so that the protection system may maintain its integrity even if the system it supports should fail.

The Cambridge CAP Computer

CAP is a microprogrammable processor, designed and built at the University of Cambridge, England, for use in the investigation of hardware-supported capability-based protection systems [3]. The microprogrammer has access to the hardware of the machine, in particular, to a capability unit. The unit consists of 64 capability registers each of which has base, size, access code and tag fields. The base, size and access fields describe a contiguous set of locations of store and the access conferred by the capability register for that region of store. A memory transfer is carried out in two parts: firstly a capability register is selected and then the address of the word to access is specified as an offset within the region delimited by the register. If the offset exceeds the size field, or the nature of the access is not commensurate with the access code, a protection violation is signalled to the microprogram. Provided that there is no protection violation, the word of store accessed is the one whose absolute address is the summation of the base field and the offset. Capability registers can be selected by performing an associative search of all the registers in the capability unit for one having a particular value in its tag field. This mode of operation can be used to make the unit function as a capability cache.

As well as protection, the CAP microprogram is responsible for the basic instruction set of the machine, interrupt handling and peripheral device transfers over a fast link to a peripheral processor.

A memory protection system has been constructed for the machine and is described elsewhere [4]. The system has been successful [5] primarily because of its fine-grainedness and the efficiency of its

mechanisms. The subject of this paper is a design (CAP-3) that extends the scope of the memory protection scheme (CAP-1) to offer a wider range of facilities, including the protection of abstract or typed objects.

Naming Structure

Unlike CAP-1 which has a nested naming structure, where the name of an object is its address in a higher name space, CAP-3 has a global naming structure. Every capability includes a global name which selects an entry in a central table, the map, describing the object referenced by the capability. All capabilities for the same object, although possibly conferring differing access rights, lead to the same map slot. Protection operations upon objects are done by altering the contents of map entries.

In a global naming scheme such that employed by HYDRA, global names are unique for the entire lifetime of the system. Such a name space is vast, with long identifiers (64 bits) and is sparse because many names will belong to deleted objects. To handle the identifiers, it is usual to organise the map as a hash table, keyed on unique identifiers. The HYDRA kernel is obliged to ensure that it always keeps the map in a consistent state, even over a system break, so that capabilities preserved on backing store will be interpreted correctly when the system is restarted. Hashing is a slow and complex process compared to direct indexing; assembling an object such as a domain that contains many capabilities will consume considerable effort. As the map holds an entry for every object currently known to the system its size, even if the map is represented compactly, is too great to hold totally in memory: some swapping mechanism is needed to page the large overall structure from disc or drum into a smaller resident table. The ideal swapping system would be the virtual memory mechanisms of the operating system, but these cannot be used as the virtual memory relies on the kernel for protection!

CAP-3 leaves unique name management in the above sense to software; the kernel provides no support for long term names nor for preserving naming integrity over system breaks. These functions must be carried out by software running above the kernel and it is the duty of this software to map its naming system onto that of the kernel. The central map is resident in store and addressed directly as a vector by simple indexing. The map may have a maximum of 16383 entries, although its size in practice is expected to be much smaller (say 2000-4000 entries). Objects only remain in the map if there is an active capability for them. An active capability is one that is addressible by an active process running in the machine. The microprogram will detect a

capability becoming inactive and reclaim the corresponding map slot for subsequent reuse.

The advantages that the global naming scheme of CAP-3 has over the nested scheme of CAP-1 are that the global nature of names makes it possible to transfer capabilities between protection domains without having to translate names between spaces, and that evaluation of capabilities can be faster than in CAP-1 as it is not necessary to climb a hierarchy of name spaces in the course of the computation.

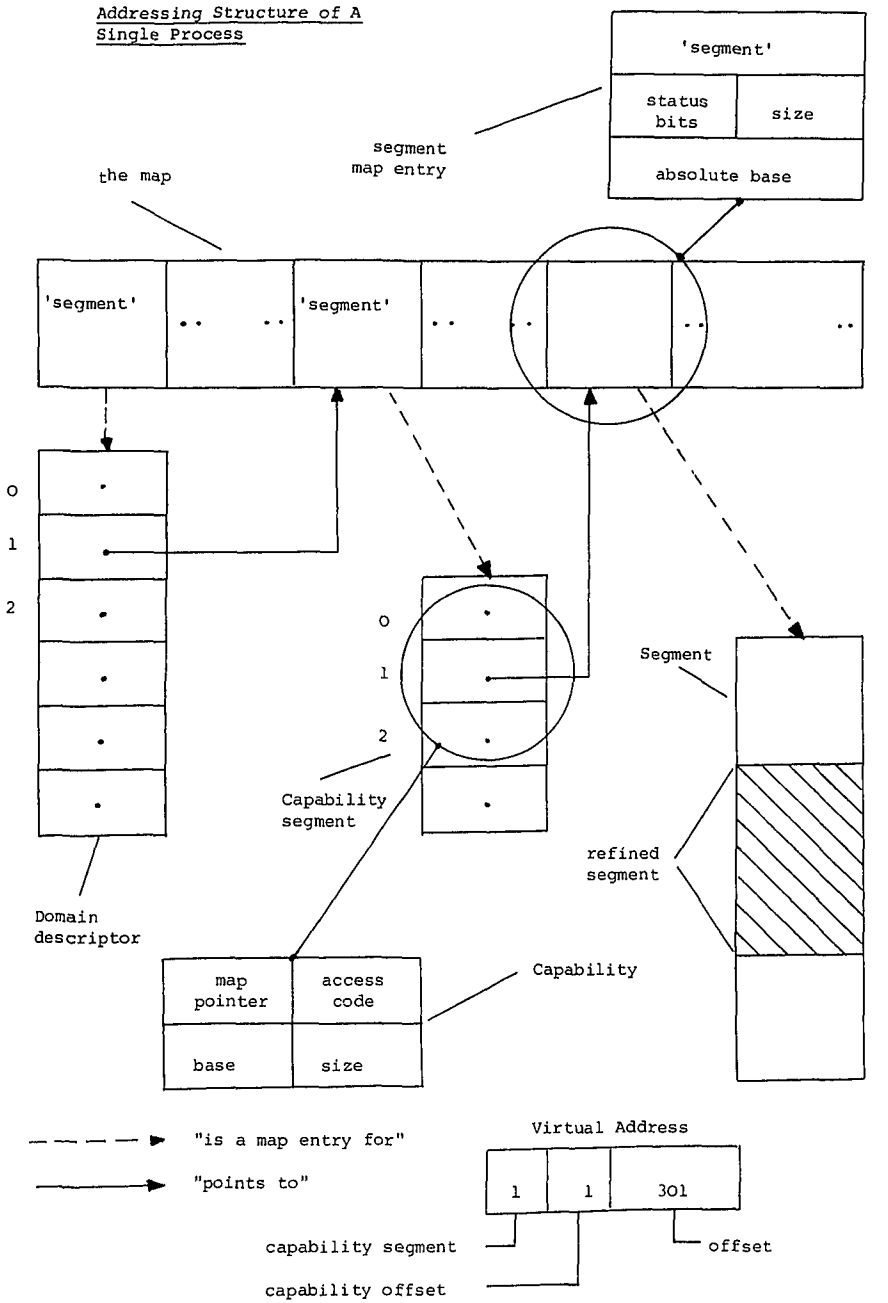
Segments

The basic unit of memory protection is the segment, a contiguous set of words of memory up to 65535 words in all, with a granularity of one word. Segment capabilities may have data access, in which case the access code is a selection of read data, write data, and execute, or capability access, with accesses read capability and write capability. The access codes can be used to ensure that capabilities are neither forged nor corrupted.

Addressing

Addresses in CAP-3 are interpreted relative to a domain of protection. The structure of a protection domain is rooted in a capability segment called the domain descriptor. There are entries in the domain descriptor for up to 16 capability segments that form the address space of the domain. Each of these segments can hold up to 256 capabilities. A virtual address (32 bits) divides into two major parts: a capability specifier (16 bits) and an offset (16 bits). The capability specifier further breaks into two: capability segment selector (4 bits) selecting one of the 16 segments in the domain descriptor and a capability offset (8 bits) indexing a capability from within the selected segment. In addition there are four unused bits. The capability specifier indicates the location of the capability for the object being addressed. If the object is a segment then the offset part of the virtual address refers to a particular word within the segment.

The CAP hardware provides support for virtual address translation. The capability specifier of a virtual address is used to compute the key for searching the capability unit during a memory transfer. If a capability register in the unit with the required key is not found, the microprogram evaluates the capability from store and loads it into the unit. The hardware mechanism reduces microprogram involvement in addressing to exception handling only, giving a considerable saving in overheads. The binding of capabilities to


Addressing Structure of A Single Process
'segment'
status bits
size
absolute base
segment map entry
the map
'segment'
'segment'
0
1
2
0
1
2
Segment
Capability segment
refined segment
Domain descriptor
map pointer
access code
base
size
Capability
"is a map entry for"
"points to"
Virtual Address
1
1
301
capability segment
offset
capability offset

addresses simplifies protection for users: they need not concern themselves with capability register allocation, as would be necessary in an explicit capability register machine such as the Plessey System 250 [6].

Memory Protection

As was remarked upon previously, the primary unit of memory protection is the segment. The map entry for a segment consists of a segment type mark, the absolute address of the segment in memory and its size in words. There are also various status bits for the benefit of memory management in the operating system. A capability for a segment holds, as well as a map pointer and access right bits, a refining base and limit. The capability denotes that part of the overall segment starting from the refining base such that its size exceeds neither the overall size, nor the refining size. The refinement mechanism makes it possible to create capabilities giving access to only a small part of a larger segment - a facility frequently required to satisfy the "minimum privilege" principle.

Peripheral Protection

To use a particular peripheral device it is necessary to quote a capability for a word of store associated with the device - these words are known collectively as the P-store. The P-store should not be confused with the peripheral control stores found in many machines. CAP-3 has explicit instructions for starting I/O transfers which pass their arguments directly to the microprogram. Before proceeding with an I/O order the microprogram checks that a capability for the correct P-store is quoted, although the actual contents of the location are neither read nor altered. This contrivance is used for efficiency. P-store capabilities are cached on the capability unit just as other segment capabilities are, saving on the number of capability evaluations that would occur if a "device" type object mechanism were to be used instead. The operating system can control access to peripheral devices by restricting the availability of P-store capabilities.

Capability Transfer

There are machine instructions for copying capabilities between capability segments. MOVECAP will copy a capability without alteration. A more potent instruction, REFINE, performs the copying operation and at the same time removes selected bits in the access code of the copy. For segment capabilities, REFINE can also modify the refining base and size

of the capability to make the copy a subsegment of the original.

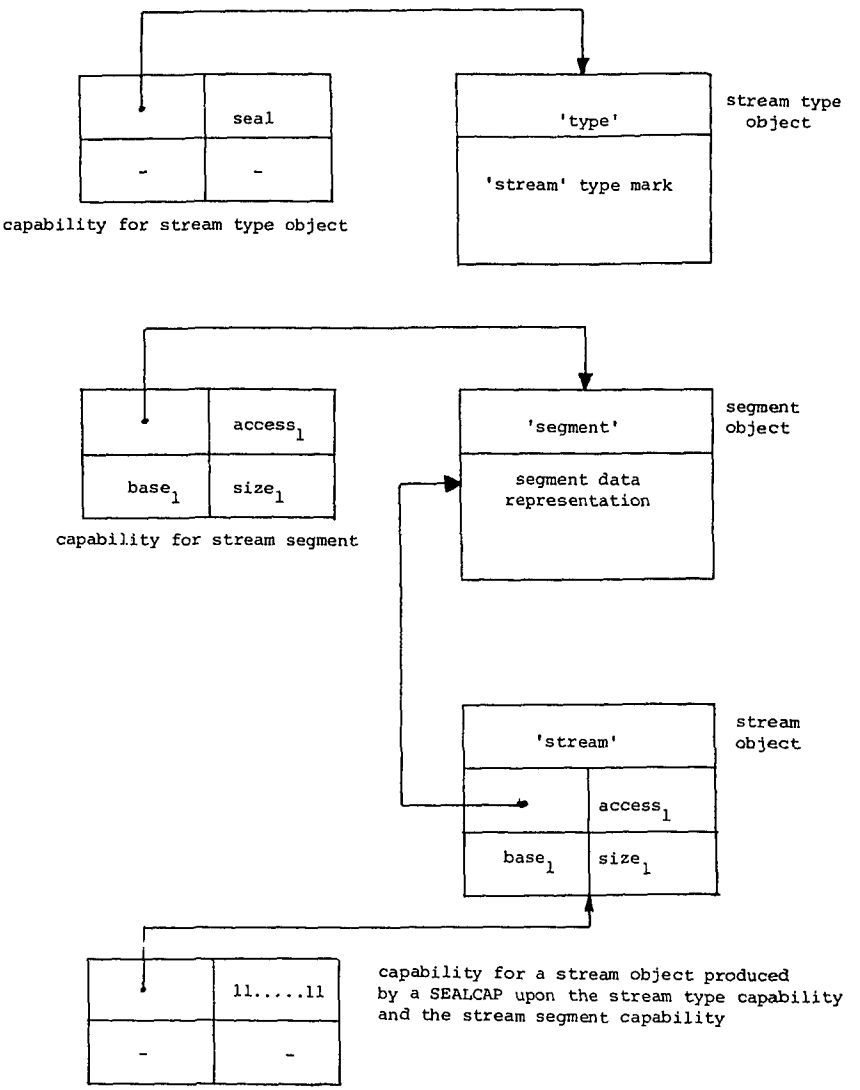
Type Objects and Type Extension

There are many types of object in an operating system; some such as segments, processes, and message channels can be interpreted by the microprogram; others such as file directories, I/O streams, etc. are the currency of the operating system software. The microprogram does not know in advance the range of types of object needed by the operating system and must be able to cope with a large variety of them.

In CAP-1, protected objects are composed of a protected procedure (a protection domain within a process) encapsulating the representation of the object. For example, a CAP-1 file directory manager is a protected procedure holding a directory segment containing names, disc addresses, and access information for files in the directory it manages [7]. There are unfortunately some flaws in this contrivance; as all abstract objects have the same type (protected procedure) it is difficult to distinguish between them unless some marking convention is used. The mechanism is cumbersome, especially for simple objects; binary operations such as "merge sorted files" cannot be carried out and, because the nested name space structure of CAP-1 makes the cost of moving a protected procedure prohibitively expensive, it is not possible to pass protected objects freely between processes.

CAP-3 uses a type extension mechanism based upon sealing information within map entries. The representation of an object is placed, together with a type mark, in a map entry which serves to define the object. A capability for an object does not confer any privileges for manipulating the contents of the map entry describing the object. Thus the integrity of the contents of the entry are ensured as they are sealed within the map.

The purpose of the type mark is to distinguish between different sorts of objects. The representation may be either a capability representation or a data representation. A data representation consists of a protected bit pattern. A segment map entry, for example, has the absolute base, size and status of the segment sealed within it. Capability representations are used to describe objects that are implemented in terms of other objects; for example, a stream object may be implemented as a segment of memory; concealing the segment in a stream object prevents holders of a stream capability from accessing the segment and corrupting the stream structure. The ability to seal capabilities furnishes an extensible protection architecture; any new object can be defined in terms of pre-existing objects, reflecting the general



An Example of Capability sealing

principles of layering or levels of abstraction [8].

Certain operations such as object creation are similar for all types of objects in terms of changes to the map; to perform such a generic type operation it is mandatory to present a capability for a type object representing all objects of the class being exercised. A type object is a data object having a characteristic type field, recognisable by the microprogram. The representation of the type object is a bit pattern that is the type mark of all objects of that type. The type object has no other significance than as a means of controlling generic type operations.

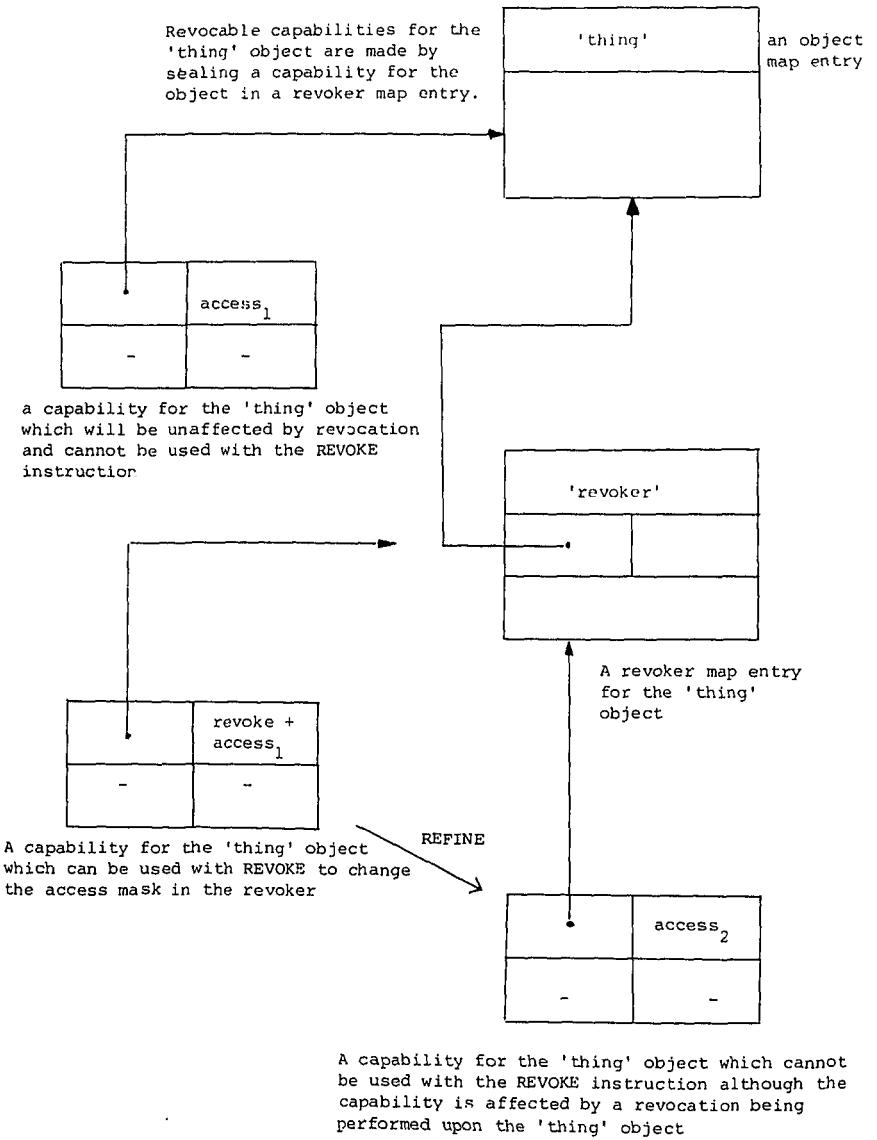
The SEALDATA instruction will create a capability for a data object. The arguments of the instruction are a capability with seal access for a type object and the data to seal. The microprogram finds an unused map slot and primes the type field with the type mark from the representation of the type object. The data argument is loaded into the representation part of the entry. The outcome is a fully privileged capability for the new object. Type objects can be made by sealing data specifying a new type mark with a type object whose representation is the "type object" type.

If sealed data is to be interrogated, in the case of a password or some such thing, the UNSEALDATA order is employed. The arguments of UNSEALDATA are a capability for the object to unseal and a capability, holding unseal access, for a type object matching the type of the object to be unsealed. The map entry to be unsealed is checked to ensure that it possesses a data representation and the representation is extracted.

An object, such as a segment, can have a data representation that needs to be modified if, for example, the segment is relocated in memory. There is an instruction, ALTERDATA, which supports this operation. It takes a capability for a data object with alter access together with a capability with seal access for an appropriate type object and loads a new bit pattern into the representation of the data object. Any capabilities in the capability unit derived from this object are flushed out so that they will subsequently be re-evaluated in the light of the nascent representation.

Objects with a capability representation are sealed and unsealed by SEALCAP and UNSEALCAP respectively. These instructions are analogous to their counterparts for data sealing except that SEALCAP copies a capability into the representation of the new object and UNSEALCAP returns a capability as its result. There is also an ALTERCAP instruction.

An Example of Revocation



As objects are represented by map entries identified by global names, they can freely migrate between processes without difficulty. The type object mechanism for protecting objects enables generic operations to be carried out by the microprogram kernel, even though it has no knowledge of the meaning of the objects themselves. This type extension scheme is based on a design proposed by D. Redell [9].

Revocation

A further feature of Redell's design is that by the inclusion of a special variety of map entry it is possible to implement a revocation scheme that fits into the overall type extension architecture. In CAP-1 any object that is to be revocable has to be concealed in a protected procedure which performs all operations on the protected objects, subject to the revocation conditions. This is a very expensive technique for frequently accessed objects.

In CAP-3 an object is made revocable by sealing a capability for it in a revoker map entry. The revoker contains an access mask which controls access to the sealed object. If, in the course of the evaluation of a capability, the microprogram encounters a revoker, it calculates the access to the object to be the intersection of the access mask in the revoker and the access code in the capability. Several revokers can be scanned in this way until the root object is found. Apart from its access mask a revoker is transparent to all other operations; it does not conceal either the type mark or the representation of the root object.

The REVOKE instruction may be used to modify the access mask of a revoker. REVOKE works upon a capability holding the generic access revoke for a revocable object and will load a new bit pattern into the mask of the revoker pointed at by the capability.

By careful application of the revoke bit (similar in role to Redell's lock bit) and multiple chains of revokers it is possible to implement a considerable range of revocation policies.

To make revocation immediately effective it is necessary to remove from the capability unit any capability whose evaluation involves an indirection through a revoker changed by REVOKE. Stored in every evaluated capability is the map slot of the root object from which the capability register was set up. When a REVOKE order is executed, the root object of the chain of map entries following the revoker is determined and any entries in the capability unit for this object are flushed out. This guarantees the removal of all capabilities evaluated through the revoker, although upon occasion it may remove more

capabilities than is strictly necessary.

Revoke access and alter access are the only two generic access codes and have the same significance in all capabilities. The remainder of the access code in a capability is type dependent and is not the concern of the kernel unless, of course, the type is hardware-supported.

Map Management

The microprogram kernel keeps a pool of free map slots and can detect objects becoming inactive so that their entries can be reclaimed. For this purpose there is a reference count associated with each map slot recording how many capabilities and other map entries point to it. Whenever a capability or map pointer is destroyed the reference count of the object held up by the pointer is decremented. Should the reference count fall to zero, there is no need to retain the object in the map and its slot is returned to the free pool.

If a capability segment is sealed in an object and a capability for this object is subsequently copied into the capability segment, a circular structure is formed. The reference count mechanism cannot detect circular structures becoming inactive. An asynchronous capability segment garbage collector modelled on the CAP-1 filing system garbage collector [10] can be used to dispose of inactive circular structures. The microprogram does not flush out a capability segment when its reference count reaches zero as the task can be deeply recursive and time-consuming. The microprogram is responsible only for the map management; capability segment management is in the hands of the operating system.

When an object is sealed, its creator can label the new map entry with an identifier of his own choosing so that the object can be recognised subsequently without having to keep tables relating objects to map slots.

Process Structure

CAP-1 has a hierarchical process structure and a non-hierarchical protection domain structure within processes. CAP-3 attempts to capture the advantages of CAP-1's domain structure by having a non-hierarchical process structure where each process is in a single protection domain. Capabilities are passed between processes by a microprogrammed message system. Each process is defined by its domain descriptor capability segment. A particular segment capability in the descriptor is for the process base. In this segment there is a register dump area and information describing the state of the process.

A process may send a message to a channel. A channel capability leads to a channel object whose representation is a capability segment. The segment contains capabilities for the process to be woken up on the arrival of a message at the channel and for the queue of messages queued upon to the channel.

In a process' domain descriptor there is a capability for a capability segment known as the message pool. Messages are constructed as fixed size subsegments allocated out of the pool. The allocation is carried out by the MAKEBLOK instruction which delivers a capability for a message object. The message object has a capability representation, that is, a capability for the portion of the message pool - the message block - allocated for this message. An argument of MAKEBLOK specifies an identifier that may be used to recognise messages when they are replied to. There are instructions for transferring capabilities in and out of message blocks.

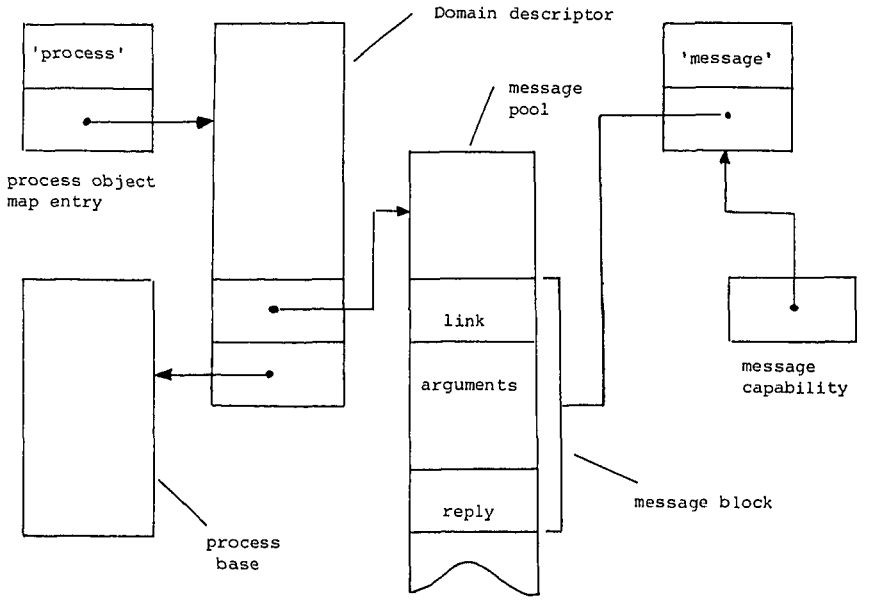
Messages are despatched by the SEND instruction which is presented with a capability for a message object, a capability for a channel object and an optional capability for a reply channel. The reply capability is dumped in the message block but it cannot be extracted by the receiver of the message. The message block is attached to the possibly empty chain of blocks waiting at the destination channel. The process owning the destination channel is then marked as active. Finally, the representation capability of the message object is made invalid so that the sending process loses access to the message block. A process may elect to wait after sending a message.

Channels are polled by the RECEIVE instruction. RECEIVE takes a message from a channel's queue and makes it available as a capability for a message object freshly created in the map with a capability for the extracted message block as its representation. If there are no messages on the queue RECEIVE yields a numerical return code indicating the absence of a message.

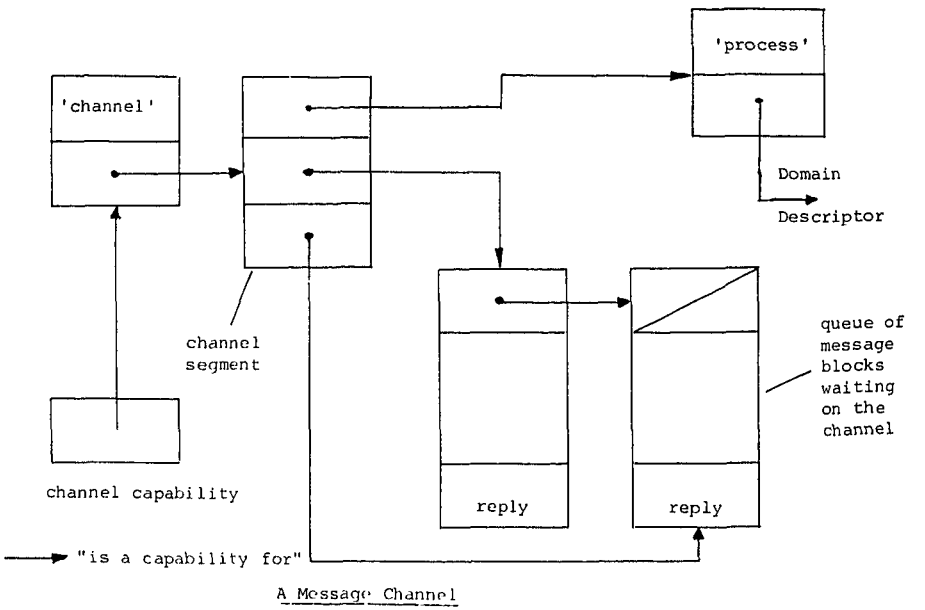
When the receiver of a message has finished with it, he executes the REPLY instruction upon the message object capability. REPLY invalidates the representation capability of the message object so that it is useless to the receiver. If the sender of the message included a reply channel capability, the message block is sent back to this channel. The destruction of the message object ensures that the reply capability is only used once. There is a variant of REPLY that causes a wait after despatching the message.

When a message transaction has been completed, the message block may be returned to the message pool from which it was made by obeying the

Message System Structure



A message prior to SEND



A Message Channel

KILLBLOK instruction on a message object capability describing the block. The capabilities in the message block are flushed out and the message block is invalidated to prevent it being used subsequently. It is not possible to kill a message which contains a reply capability until the reply has been exercised.

Various access bits are associated with both message object and channel object capabilities. These bits are used to control the privileges of sending messages, receiving messages and issuing replies and also to ensure the correct sequencing of message transactions.

When a process is woken up by the arrival of a message the microprogram judges whether or not a process switch from the transmitting process is necessary on the basis of priorities held in each process' process base.

If a process wishes to await the arrival of a message it may suspend by executing the WAIT instruction. All holdups, interrupts, and traps are handled by the Interrupt process, known to the kernel. The interrupt process may cause another process to run by obeying the WAKEUP instruction specifying a capability for a process object which has a capability for the new process' domain descriptor as its representation. Control is immediately transferred to the selected process. A "wake up waiting" field in each process base is used by the kernel to prevent a process from losing a message that arrives in the middle of the execution of a RECEIVE and WAIT instruction sequence.

The structure of the capability unit makes it possible to leave a process' capabilities inactive in the unit when control switches to another process and then to make them easily accessible again when the first process resumes, which reduces the overheads of interprocess communication. The CAP-3 message system aims to be as cheap as the protected procedure mechanism of CAP-1. The mechanism removes the gross distinction between processes and protected procedures found in CAP-1.

Implementation and Future Plans

At the time of writing, the microprogram has been completely coded and runs several test programs. It is just under 4000 micro-instructions long. The code is divided up thus: basic instruction set, 1000 orders; memory protection, 500 orders; type extension and revocation, 500 orders; processes and messages, 1000 orders. The remaining orders are accounted for by initialisation, interrupt handling and fault reporting code.

Ad hoc measurements reveal that a message transaction between processes takes between 100 and 200 times as long as a simple "load from

store" instruction. Type extension and revocation operations are at about 20 on this scale. By contrast, CAP-1 can carry out a protected procedure call in 50 units whereas a (software) message transaction takes several thousands of units. It should be noted that the CAP machine has a "vertical" style of microprogram with little scope for parallelism, so one might hope to do better than this in other machine architectures.

The major test of the kernel has yet to come: will it support an operating system? It would be straightforward to re-implement the current CAP operating system to run with the new microprogram. A greater challenge exists in the production of a system utilising the type extension and multiprogramming mechanisms of the kernel both within the operating system and in the facilities offered to users. The main question is whether or not the kernel primitives are too cut down compared to the analogues in, say, HYDRA to be effective. It is in answering questions of this sort that the future of the kernel lies.

An Evaluation

The design described above provides an indication of the level of facilities that it is reasonable to provide in a microprogrammed protection system. The kernel is self-contained and its interfaces to the operating system are clearcut and simple. The kernel tries to avoid forcing a particular structure on the software built around it. The primitives of the kernel are a firm foundation for the construction of a well-protected operating system. The kernel does not trust the software built upon it; all data structures and arguments are checked at every stage. While this may slightly reduce efficiency, it does lead to the production of a rugged and reliable kernel. This approach has been successfully adopted in previous CAP microprograms to good effect. The microprogram kernel, aided by the sophisticated hardware of CAP, has the properties of simplicity, flexibility, and economy of mechanism necessary in a worthwhile rugged, extendible, fine-grained protection system.

Acknowledgments

The CAP project is supported by the Science Research Council. My research supervisor, Dr. R. M. Needham, A.D. Birrell, and other members of the CAP project have greatly helped the design process by offering advice and through constructive criticism of any part of the design that seemed in danger of becoming inelegant, cumbersome, or misguided.

References

1. Cohen, E. and Jefferson, D. "Protection in the HYDRA Operating System", Operating Systems Review, Vol.9, No.5, November 1975.
2. Lampson, B. W. and Sturgis, H. E. "Reflections on an Operating System Design", Communications of the Association for Computing Machinery, Vol. 19, No. 5, May 1976.
3. Herbert, A. J. (Editor), "The CAP Hardware Manual", Computer Laboratory, University of Cambridge, England, 1978.
4. Needham, R. M. and Walker, R. D. H. "The Cambridge CAP Computer and Its Protection System", Operating Systems Review, Vol.11, No.5, November 1977.
5. Needham, R. M. "The CAP Project - an Interim Evaluation", Operating Systems Review, Vol.11, No.5, November 1977.
6. England, D. M. "Architectural Features of System 250", Infotech State of the Art Report on Operating Systems, 1972.
7. Needham, R. M. and Birrell, A. D. "The CAP Filing System", Operating Systems Review, Vol.11, No.5, November 1977.
8. Dijkstra, E. W. "The Structure of THE Multiprogramming System", Communications of the Association for Computing Machinery, Vol.11, No. 5, May 1968.
9. Redell, D. D. "Naming and Protection in Extendible Operating Systems", Ph.D. Thesis, University of California at Berkeley, September 1974.
10. Birrell, A. D. and Needham, R. M. "An Asynchronous Garbage Collector for the CAP Filing System", Operating Systems Review, Vol.12, No.2, April 1978.

APPENDIX 2

ALGOL68C AND ITS RUN TIME SYSTEM

ALGOL68C is described in Bourne et al. (1976). The compiler, which is itself written in ALGOL68C, produces output in an intermediate language known as Zcode. The Zcode is then converted into machine language by means of a translator. Parameters associated with the compiler enable the exact form of the Zcode to be so chosen that it will, within limits, be well adapted to the computer for which the compilation is being done. The ALGOL68C compiler contains a special feature whereby the environment - notably the status of the stack - as it exists at a particular point in compilation may be preserved and subsequently reinstated. This mechanism was provided so that the separate compilation of sections of the program should be possible.

As in the case of other languages, it is necessary, in order that the generated code should run, that there should be present in memory a certain number of standard routines. These constitute the run time system. All but a very small part of the run time system is written in ALGOL68C. However, the lowest level, containing about 220 instructions and known as MC, is written in assembly language. It provides the support services, such as heap and stack allocation, that must be present for any compiled code to run at all. MC is part of a segment known as MIN which also contains a section of code for supporting the operations of moving capabilities and for managing capability segments. This is written in ALGOL68C with a few interpolated machine code instructions, notably instructions for performing operations peculiar to the CAP. For example, it contains the body of the ALGOL68C procedure movecap which has embedded in it the CAP instructions for moving capabilities, namely MOVECAP and MOVECAPA. This is an example of a device sometimes known as a write-around, whereby facilities available in machine code, but unknown to the compiler of a high level language, may be made available through procedure calls in the language. Other machine instructions besides MOVECAP are dealt with in the same way; for example, the ALGOL68C procedure call

```
enter(proc, a, b, c, d)
```

is executed by a write-around consisting of machine instructions which load the numerical parameters a, b, c, d into the registers B1, B2, B3, B4 and then enter the protected procedure PROC. It is assumed that, before calling the enter procedure, the ALGOL68C programmer will have used a call to movecap to place on the C-stack those capabilities that he wishes to pass as arguments.

MIN constitutes one segment of the run time system. Another segment, known as SER, supports message passing and other coordinator services. A third segment, USE, contains a number of higher level write-arounds. One of these makes it possible for the programmer to write a call to DIRMAN as

```
dirman(a, b, c, d)
```

instead of in the less convenient and elegant form

```
enter(dirman a, b, c, d).
```

Others provide similar facilities for STOREMAN. USE also implements the ALGOL68C transput calls.

Each protected procedure in the CAP system is written and compiled as a complete ALGOL68C program and includes its own instance of the run time system, or of as much of it as it needs. Naturally, the code itself is shared. The powerful facilities included in the ALGOL68C system for preserving and reinstating compile time environments were of crucial importance in the design and implementation of the system for compiling protected procedures, and without them it would have been difficult to achieve anything like the same degree of elegance and efficiency.

MAKEPACK is used by the ALGOL68C system to put capabilities for the segments containing the run time system and for a segment containing the compiled code in the P capability segment of the protected procedure. This is the reason why in the programs shown in Appendix 3 the first few slots of the P capability segment are left free.

A typical protected procedure is constructed according to the following plan:

```
BEGIN
   initialisation code
END;
DO # to infinity #
CASE first argument IN
   code for performing services offered by the protected procedure
ESAC;
return(result)
OD;
```

When a protected procedure is entered for the first time after it has been retrieved from the filing system, control enters at the beginning and the initialisation code is executed. This code may be quite lengthy and will include operations for the setting up of message channels. It is followed by an indefinite DO statement containing a CASE

statement switched on the first of the integer parameters with which the protected procedure is called; it is this parameter which specifies the nature of the operation. Control thus passes to the appropriate section of code in the CASE statement and when this has been executed a procedure known as return is entered. This procedure is implemented in the MC section of the run time system and brings about an exit from the protected procedure by means of a RETURN instruction. Before doing this, however, it plants a link in such a way that, when the protected procedure is next entered, control passes to a point immediately after the call to the return procedure by which it left. The effect is that, when a protected procedure is first called, control enters at the beginning, but subsequent calls pick up where the last one left off. This provides what is in effect a coroutine mechanism.

Some services are provided by procedures running in independent processes. The process is normally halted and is woken up by a message when its services are required. Procedures intended to be used in this way are constructed according to the following plan:

```
BEGIN
   initialisation code
END;
DO # to infinity #
WHILE messages (input) = 0 DO waitevent OD;
   receive message with reply(a, b, c, d);
CASE a IN
   code for performing services offered by the protected procedure
ESAC;
return reply(p, q, r, s);
OD;
```

The decision as to whether a given service should be provided by means of an independent process or not is often somewhat arbitrary. The fact that the two implementations have a very similar structure makes it relatively easy to effect a change if one is required.

APPENDIX 3

SPECIMEN PROGRAMS

The programs given below constitute a group concerned with the management of the virtual memory and the filing system. The most central of these programs is SINMAN, which is responsible for the management of system internal names (SINs), for keeping the authoritative records of object types, and for maintaining reference counts. SINMAN communicates with the virtual store manager, VSM, in order to arrange for objects to be brought into the active virtual memory as necessary. In turn VSM communicates with the real store manager. The program for the real store manager is not given here.

System internal names are integers. The possession of an enter capability for SINMAN is, therefore, an important privilege, since it implies a trust not to mishandle these integers. Four programs with this privilege are presented here: DIRMAN, the file directory manager; MAKEPACK, the program responsible for making new PDBs; STOREMAN, which allows a user to create new objects and change their size; and DISCGARB, the asynchronous disc garbage collector. DIRSTRUCT, a piece of source code which defines the layout of directory segments, is also presented. All the programs given here are created during system generation, except MAKEPACK which is retrieved from the filing system in the ordinary way when the system has been started up.

The programs have been slightly simplified in the interests of brevity by omitting the bodies of some procedures and explaining instead their functions in comments. The names of some variables have been altered from those used in the working version to correspond with the terminology adopted in the text and also in some cases to make their significance clearer.

The programs all operate in the environment of the ALGOL68C run time system. This provides services for setting up and using message channels, together with a variety of services connected with capability management. When these services are requested, an appropriate software capability has usually to be presented. In many cases the nature of the service provided by the run time system will be apparent from the name of the procedure call used to request it. For example, getslot and freeslot manage slots in the I capability segment.

Some procedures whose functions may not be clear from their names are described below. Note that the mode SLOT refers to capability segment entries. A typical declaration is:

```
SLOT commcap = getslot;
```

this declaration causes the allocation of a slot which is later used for a (dynamically created) communication capability.

maparray This procedure converts a SLOT into REF[]INT. It provides a means for talking in ALGOL68C about the contents of segments other than the stack and the heap.

enter A write-around which calls the machine ENTER instruction (see page 7). It takes as arguments a SLOT and up to five INTs. The action is to execute an ENTER instruction with reference to the capability at SLOT with the five INTs in the first five B registers.

enter2 The same as enter, but with an additional five REF INT arguments to which the contents of B2 to B5 are assigned on return.

return Takes an INT, sets it in B2, and then executes a RETURN instruction. On a subsequent call of the protected procedure control will pick up in return with the variables first argument to fifth argument set from B0 to B4.

return2 The same as return, but taking four INTs which are assigned to B2 to B5.

run time error This is a procedure variable; the procedure currently assigned to it will be called as the result of any trap or error. There is a default procedure which simply returns the fault to the caller.

The following procedures are, like enter, write-arounds for the corresponding machine instructions: indinf, seginf, segsize, cseginf, movecap, movecapa, refine, makeind.

A number of identifiers are declared and set equal to constants used in the programs for masking and other purposes. Examples are: exec access, read access, write access, rcap access, wcap access, capability permission, store permission, system stop permission. The last three refer to permission bits used in software capabilities. The capability specifier in the master coordinator's address space of the capability for a segment is denoted by mcaddr. Many communications that pass between system processes concerning segments are expressed in terms of their mcaddrs.

There are a number of procedures through which services provided by the master coordinator and ECPROC are made available to users. Such procedures have names that are suggestive of their function. For example, procedures concerned with sending and receiving messages are send data message, wait event. These and similar procedures always take

a SLOT argument giving a send, receive, or reply capability as appropriate. They may take other arguments as well. An example of a procedure for capability management is update prl capability. Procedures in this class take a SLOT argument giving the appropriate permission capability. Another procedure which takes a permission capability in a SLOT argument is: set up send with reply.

Fault Numbers and Return Codes

Generally speaking, if a protected procedure is given improper data or if the external circumstances are such that it cannot carry out its task, it issues a return code or raises a fault with a fault number indicating what has happened. If there is evidence of serious corruption of program or data it may be appropriate to stop the system. Only a few protected procedures are able to do this, the privilege being conferred by a software capability. The faults raised by procedures which are called directly by user programs usually cause a compulsory diversion of control in the caller; other procedures give return codes which it is the responsibility of the caller to inspect. In either case, the return code consists of a conventionally laid out 32 bit word. In the case of software-detected errors, the first hexadecimal digit in the word is always 8. The next digit records the depth of procedure nesting at which the fault occurred; if a procedure passes back to its caller a fault which it has itself received, it increments this digit by one. The next two digits indicate the source of the fault; for example 0A indicates that the fault occurred in DIRMAN, 0B that it occurred in MAKEPACK, and 21 that it occurred in the ALGOL68C run time system. The next two digits are used in connection with traps occurring in the capability loading cycle, and the last two digits give more detailed information about the fault that has occurred.

SINMAN

SINMAN is the program which handles the SIN directory and thus is the basis for the management of all disc objects, permanent or temporary. For each object it maintains a reference count of the number of times its SIN is recorded in directories or PDBs. SINMAN also collaborates with VSM to arrange that objects are kept in existence as long as there is a capability for them in the active virtual memory. Since calls to SINMAN often have SINs as arguments, enter capabilities for SINMAN are only given to programs which must be trusted with SINs for other reasons. They are RESTART, DIRMAN, MAKEPACK, LINKER, STOREMAN, VSM, and DISCGARB. Note that entry requests 14 to 19 are associated with DISCGARB; in particular requests 16 to 19 provide efficient facilities for creation, alteration, and deletion of capabilities which avoid generation of garbage in the PRL but rely on the good behavior of DISCGARB.

```
SLOT stop slot          = P 4,
     create cap         = P 5,
     info cap           = P 6,
     channel modstart   = P 7, # message channel for system start-up #
     channel module     = P 8,
     channel rsm        = P 9,
     channel module2    = P 10,
     channel vsm        = P 11,
     mcaddr slot        = P 12,
     dirman p capseg    = P 13,
     dirman i capseg    = P 14,
     channel vsmdel     = P 15,
     channel disc garb  = P 16,
     dg running         = P 17,
     # R capability segment is shared with PRLGARB #
     pslslot            = R 0,
     vsm delete         = R 1,
     make enter         = G 5,
     arg slot           = A 0;
     # I capability segment slots all managed by runtime system #

runtime error := (STRING s, INT i) VOID: stop system (stop slot, i);

SLOT inst wk1           = getslot,
     inst wk2           = getslot,
     preserve wk        = getslot,
     new segment wk     = getslot,
     from wk            = getslot,
     to wk              = getslot,

PROC null = (SLOT s) VOID: movecap (null capability, s);

INT blksize             = 7 * 128,
    cap access          = ABS 16r3f0000,
    rwaccess            = read access ! write access,
    rcwcaccess          = rcapaccess ! wcapaccess;
```

```
# Process SIN list (PSL) #

# The process SIN list is shared with PRLGARB. It relates PRL slots to
  SINs and is used to determine whether a capability for an object of
  given SIN does not already exist. It is also used to ascertain the SIN
  of a capability collected by PRLGARB in order to report it to VSM. #

# The PSL is organized as a hash table.  Entries are:

    ---------------------------
    | PRL |d |   acc   |  SIN  |
    ---------------------------
    31  23 22 21    16 15     0

  The following declarations concern the size of the PSL and the various
  masks for its fields. #


INT psl size           = 499,
    psl incr           = 13;
REF [] INT psl         = maparray (pslslot);
INT pslunset           = -1;
INT pslprlshift        = 23,
    psldeletemk        = ABS 16r400000,
    pslkeymask         = ABS 16r3fffff,
    pslsinmask         = ABS 16rffff,
    fdmpslacc          = exec access;


PROC psladd = (INT sin, acc, prl) VOID:
     BEGIN
         SKIP # the detail of adding an entry to the PSL #
     END;

PROC pslfind = (INT sin, acc) INT:
     BEGIN
         SKIP # the detail of looking up in the PSL given a SIN #
     END;

PROC keyfromprl = (INT prl) INT:
     BEGIN
         SKIP # the inverse search, given PRL offset #
     END;


# SIN directory layout and access routines #

# The SIN directory is on 18 blocks, whose SINs are read initially from
  the restart block. #

SLOT restartslot = getslot;
REF [] INT restartseg = maparray (restartslot),
INT restartsinblk = 2; # offset of SIN of first SIN directory block #
INT sinmax = 7999;
INT sinblkmax = (sinmax + 1) * 2 % blksize; # number of blocks #
[0: sinblkmax] REF [] INT sinblk,
[0: sinblkmax] SLOT sinslot;

FOR i FROM 0 TO sinblkmax
DO sinblk [i] := maparray (sinslot [i] := getslot) OD;
```

```
# Each entry in the SIN directory consists of two words:

    ------------------------    ------------------------
    |SAFETY|  PUC     | TUC |    |   TYPE |  SIZE       |
    ------------------------    ------------------------
     31   24 23       8 7   0     31     24 23          0 #

INT tuc                 = 1, # temporary use count #
    puc                 = ABS 16r100, # permanent use count #
    pucsafety           = ABS 16r1000000,
    tucmask             = ABS 16rff,
    pucmask             = ABS 16rffff00,
    safetymask          = ABS 16rff000000,
    puctucmask          = ABS 16rffffff,
    pucshift            = 8,
    unusedsin           = -1,
    sizemask            = ABS 16rffffff, # 24 bits #
    sintypeshift        = 24;

# require: tuc < 256, puc < 65536, number of simultaneous preservations
  < 128 #

INT sdmtype             = 0,
    segtype             = 1,
    fdmtype             = 2,
    pdbtype             = 3,
    swctype             = 4;

# Each type has size restrictions as follows #
[] INT max size = [] INT (sizemask, sizemask, 2 * 896, 896, 0)
                         [: AT 0];

PROC sindirentry = (INT sin, count, size, type) VOID:
    # place appropriate data in SIN directory; SIN assumed ok #
    BEGIN
        INT blk = sin * 2 % blksize,
            off = sin * 2 %* blksize;
        sinblk [blk] [off] := count;
        sinblk [blk] [off + 1] := (type SHL sintypeshift) ! size
    END;

# variables set by 'current' to details of current SIN #
REF INT count of current, word 1 of current,
INT sin of current,
    type of current,
    size of current,
    block of current;
```

```
PROC current = (INT sin) VOID:
    # assumes 'sin' is within range and not deleted #
    BEGIN
        block of current := sin * 2 % blksize;
        INT off = sin * 2 %* blksize;
        count of current := sinblk [block of current] [off];
        word 1 of current := sinblk [block of current] [off + 1];
        sin of current := sin;
        type of current := word 1 of current SHR 24;
        size of current := word 1 of current & sizemask
    END; # current #


PROC goodsin = (INT sin) BOOL:
    # ensures 'sin' is in range and not deleted; may call 'current' #
    sin >= 0
    ANDF sin <= sinmax
    ANDF (current (sin); count of current = unusedsin);

PROC curr use count = INT:
    (count of current & tucmask) +
    IF count of current < 0 THEN 0 ELSE count of current & pucmask FI;


# Message handling #

SLOT mod send, mod reply, rsm send, rsm reply, vsm send, vsm reply,
    module2 send, modstart send, modstart receive;

setup send with reply (channel module, mod send, data reply message,
    mod reply, data message);

setup send with reply (channel modstart, modstart send,
    full reply message, modstart receive, data message);

setup send with reply (channel rsm, rsm send, data reply message,
    rsm reply, data message);

setup send with reply (channel vsm, vsm send, data reply message,
    vsm reply, data message);

setup send (channel module2, module2 send, data message);

BEGIN
    SLOT temp;
    setup send (channel vsmdel, temp, data message);
    movecap (temp, vsmdelete);
    freeslot (temp)
END;

SLOT discgarb send;
set up send (channel discgarb, discgarb send, data message);

INT mod create          = 1,
    mod extend          = 2,
    mod contract        = 3,
    mod ensure          = 7,
    mod2 del            = 13,
    mod2 copy           = 14,
```

```
INT rsm ensure          = 1,
    rsm outform         = 2,
    rsm chsize          = 3;

INT vsm sin             = 1,
    vsm mcinf           = 2,
    vsm disc            = 4,
    vsm kill            = 6;


# The following six  procedures are concerned with common message
  transactions #

INT m1, m2, m3, m4;

PROC module = (INT a, b, c, d) INT:
    BEGIN
        send data message wait event (mod send, a, b, c, d);
        UNTIL messages (mod reply) = 0 DO wait event OD;
        receive data message (mod reply, m1, m2, m3, m4);
        m1
    END;

PROC modstart = (INT i, SLOT seg) INT:
    BEGIN
        send full message wait event (modstart send, i, ?, ?, ?, seg);
        UNTIL messages (modstart receive) > 0 DO wait event OD;
        receive data message (modstart receive, m1, m2, m3, m4);
        m1
    END;

INT rsm1, rsm2, rsm3, rsm4;

PROC rsm = (INT a, SLOT cap, INT c, d) INT:
    BEGIN
        read prl capability (infocap, cap, rsm1, rsm2);
        send data message wait event (rsm send, a, rsm1, c, d);
        UNTIL messages (rsm reply) = 0 DO wait event OD;
        receive data message (rsm reply, rsm1, rsm2, rsm3, rsm4);
        rsm1
    END;

PROC ensure = (SLOT cap) VOID:
    IF INT n;
        (n := rsm (rsm ensure, cap, ?, ?)) = 0
    THEN runtime error ("rsm rej", n)
    FI;


PROC vsm = (INT a, b, c, d) INT:
    BEGIN
        send data message wait event (vsm send, a, b, c, d);
        UNTIL messages (vsm reply) = 0 DO wait event OD;
        INT p, q, r, s;
        receive data message (vsm reply, p, q, r, s);
        p
    END;
```

```
# Interface with DISCGARB #

# While DISCGARB is running, it is necessary for SINMAN to send it a
  message whenever a directory or PDB capability is retrieved.  The
  following code looks after this #

INT dg running bit      = 1 SHL 31,
    dg serial mask      = ABS 16r7fffffff;
REF INT discgarb word   = maparray (dg running) [0];

IF first argument = 0 THEN discgarb word := 0 FI;

PROC start discgarb = (SLOT b slot, c slot) INT:
    # Initialise bit maps 'b' and 'c' for DISCGARB #
    BEGIN
        # set 'dg running bit' before looking at SIN directory #
        discgarb word := (discgarb word + 1) ! dg running bit;
        REF [] INT b = maparray (b slot),
            c = maparray (c slot);
        INT c count := 0;
        FOR i FROM 0 BY bitswidth TO sinmax
        DO INT word = i % bitswidth;
            b [word] := c [word] := 0;
            FOR j FROM 0 TO bitswidth - 1
            DO IF goodsin (i + j)
                THEF type of current = fdmtype
                    ORF type of current = pdbtype
                THEF (count of current & tucmask) = 0
                THEN b [word] !:= 1 SHL j
                ELSE c [word] !:= 1 SHL j;
                    c count +:= 1
                FI
            OD
        OD;
        c count
    END; # start discgarb #

PROC notify discgarb = (INT sin, type) VOID:
    IF (discgarb word & dg running bit) = 0
    THEF type = fdmtype
        ORF type = pdbtype
    THEN send data message (discgarb send, sin,
            discgarb word & dg serial mask, ?, ?)
    FI;


# Declarations of return codes for various errors of use of SINMAN #

INT unknown request     = ABS 16r800d0001,
    wrong sin           = ABS 16r800d0002,
    unknown cap         = ABS 16r800d0003,
    wrong access        = ABS 16r800d0004,
    small size          = ABS 16r800d0005,
    wrong type          = ABS 16r800d0006,
    illegal cap         = ABS 16r800d0007,
    refined cap         = ABS 16r800d0008,
    bad makeswc         = ABS 16r800d0009,
    bad destroy         = ABS 16r800d000a,
    big size            = ABS 16r800d000b;
```

```
# Six procedures are concerned with capability creation #

PROC newcap = (SLOT cap, INT sin, acc, size) INT:
    # create a store capability at PRL level #
    BEGIN
        # create a null PRL-level capability to ensure that a PRL slot
          will be made available after VSM has been asked to allocate an
          MC address; otherwise difficulties would ensue if the PRL were
          subsequently found to be full.  The actual creation is done by
          using the ECPROC facility 'create prl capability' #
        create prl capability (create cap, cap, -1, 0);
        INT word0 = vsm (vsm sin, size, 0, sin);
        movecap (null capability, cap); # frees a PRL slot #
        INT prlacc =
            IF (acc & rewcaccess) = 0
            THEN rewcaccess
            ELSE rwaccess ! execaccess
            FI;
        INT prl =
            create prl capability (createcap, cap, word0,
                storecapability ! prlacc ! 65535 ! hardware bit) &
            65535;
        # get correct access at capability segment level #
        refine (cap, acc ! 65535, cap);
        prl
    END; # newcap #

PROC capsegcap = (SLOT cap, INT a, b) VOID:
    create capability (create cap, cap, a, b);

PROC segment from current = (SLOT cap, INT acc) VOID:
    # create capability corresponding to current SIN #
    IF INT psl = pslfind (sin of current, 0);
        (psl & psldeletemk) = 0
    THEN INT prl = newcap (cap, sin of current, acc, size of current);
        psladd (sin of current, 0, prl)
    ELSE INT prl = psl SHR pslprlshift;
        capsegcap (cap, prl SHL 16,
            hardwarebit ! storecapability ! acc ! 65535)
    FI; # segfromsin #

PROC new segment = (SLOT cap, INT acc, size, type) INT:
    # allocate work segment, create capability, deliver new SIN #
    IF size <= 0
    THEN small size
    ELIF size > max size [type]
    THEN big size
    ELIF INT sin = module (modcreate, size, ?, ?);
        sin < 0
    THEN sin # disc full, probably #
    ELSE # The SIN directory entry must be made before calling VSM;  SIN
            directory entries must have correct tuc's in case DISCGARB
            initialises itself from them and treats them as garbage
            before VSM increments the TUC #
        sindirentry (sin, safetymask + ((m3 SHL 8) & pucmask) + tuc,
                     size, type);
        INT prl = newcap (cap, sin, acc, size);
        psladd (sin, 0, prl);
        current (sin);
        count of current -:= tuc; # correct for above #
```

```
        notify disc garb (sin, type);
        sin
    FI; # new segment #

PROC new instance = (SLOT cap, seg, INT sin, acc) INT:
    IF INT rc = new segment (inst wk1, rcwcaccess, 4, segtype);
        rc < 0
    THEN null (inst wk1);
        rc # disc full #
    ELSE capsegcap (inst wk2, -1, sin);
        movecapa (inst wk2, inst wk1 + 2);
        null (inst wk2);
        movecapa (seg, inst wk1 + 0);
        makeind (0);
        makeind (3);
        movecap (dirman p capseg, n0);
        movecap (dirman i capseg, n1);
        refine (inst wk1, rcapaccess ! 65535, n2);
        enter (makeenter, 2, acc, ?, ?, ?);
        movecap (n0, cap);
        null (inst wk1);
        INT prl =
            (INT w0, w1;
                read capability (info cap, cap, w0, w1);
                w0 SHR 16);
        psladd (sin, fdm psl acc, prl);
        fdmtype
    FI; # new instance #


# The next two sections are concerned with system startup #

# Set up SIN directory blocks from restart block #

INT restartblk = 1, # disc address and SIN of restart block #
    restartmfd = 1; # offset of sin of MFD in restart block #

# Some code is omitted here. It is used only during system startup, and
  carries out various initialisation and scavenging operations #

# Facilities for common SINMAN operations are built up in the following
  group of procedures #

INT capinf0, capinf1; # words of a software capability #

PROC capinf = (SLOT cap) INT:
    # provide information about a capability #
    IF read capability (info cap, cap, capinf0, capinf1);
        (capinf1 & (hardware bit ! enter bit)) = 0
    THEN (swctype SHL 24) ! exec access
    ELIF (capinf1 & (hardware bit ! enter bit)) =
            (hardware bit ! enter bit)
    THEN illegal cap
    ELIF INT prl0, prl1;
        read prl capability (info cap, cap, prl0, prl1);
        (capinf1 & (hardware bit ! enter bit)) = hardware bit
    THEN # some kind of store capability #
        IF (capinf0 & 65535) = 0
            ORF (capinf1 & 65535) < (prl1 & 65535)
            ORF (prl0 & 65535) = 0
```

```
        THEN refined cap
        ELIF INT sin = vsm (vsm mcinf, prl0, ?, ?);
             sin < 0
        THEN sin # error from VSM #
        ELIF NOT goodsin (sin)
        THEN wrong sin # presumably system error #
        ELSE (type of current SHL 24) ! sin of current !
                (capinf1 & prl1 & cap access)
        FI
    ELIF INT psl = keyfromprl (capinf0 SHR 16);
        (psl & psldeletemk) = 0
    THEN illegal cap
    ELIF NOT goodsin (psl & pslsinmask)
    THEN wrong sin
    ELIF # instance of DIRMAN or enter capability for PDB #
        type of current = fdmtype
    THEN (fdmtype SHL 24) ! sin of current ! (capinf1 & cap access)
    ELSE (pdbtype SHL 24) ! sin of current ! (psl & cap access)
    FI; # capinf #

PROC preserve current = VOID:
    # increment puc of current sin and ensure that disc copy is ok #
    BEGIN
        IF INT ser = count of current +:=:= pucsafety;
           # The '+:=:=' compiles as single instruction. This matters
            because there is no interlock on the SIN directory #
            ser < 0
        THEN # first preservation #
            (segment from current (preserve wk, read access);
                ensure (preserve wk);
                null (preserve wk));
            # ensure module map ok #
            module (modensure, (ser SHR 8) & 65535, 65535, 0);
            count of current +:= puc - (ser & pucmask);
            # ensure SIN directory ok #
            ensure (sinslot [sin of current * 2 % blksize])
        ELSE count of current +:= puc - pucsafety
        FI
    END; # preserve current #

PROC change current = (INT ch) INT:
    # change segment size; yields -1 if disc is full.  Care is required
      to allow for the fact that more than one process may be using the
      segment; the situation is particularly tricky if more than one
      process attempts to change the size of the segment;  it is
      important that the system should not be damaged even if one of the
      users does not get the result he expects.  Care is also required to
      allow for crashing during the operation.  If this happens the size
      as given by the map is authoritative during restart. #
    BEGIN
        INT rc =
            IF ch > 0
            THEN IF INT rc = module (modextend, sin of current,
                                     ch, ?);
                    rc < 0
                THEN rc # disc full #
                ELSE rsm (rsmchsize, argslot, ch, ?);
                    (word 1 of current +:= ch) & sizemask
                FI
            ELIF ch = 0
```

```
            THEN size of current
            ELIF INT new = (word 1 of current +:=:= ch) + ch;
                (new - 1) SHR sintypeshift = type of current
            THEN # size would be < or = 0 #
                word 1 of current -:= ch;
                small size
            ELSE rsm (rsmchsize, argslot, ch, ?);
                module (modcontract, sin of current, - ch, ?);
                new
            FI;
        IF count of current > 0
            ANDF (count of current & pucmask) = 0
        THEN # write up map asynchronously #
            send data message (module2 send, mod2 copy, ?, ?, ?)
        FI;
        rc
    END; # change current #

PROC instance = (SLOT cap, INT acc) INT:
    IF INT psl = pslfind (sin of current, fdm psl acc);
        (psl & psldeletemk) = 0
    THEN segment from current (cap, rwaccess);
        new instance (cap, cap, sin of current, acc)
                    # may return failure #
    ELSE capsegcap (cap, (psl SHR pslprlshift) SHL 16,
                    enterbit ! acc);
        fdmtype
    FI;

PROC enter current = (SLOT cap, INT acc) INT:
    # make an enter capability for current SIN which refers to a pdb #
    BEGIN
        INT prl =
            newcap (cap, sin of current, read access,
                    size of current);
        psladd (sin of current, acc, prl);
        capsegcap (cap, prl SHL 16, enter bit);
        pdbtype
    END;

PROC delete current = VOID:
    # delete the current virtual memory object from disc #
    IF type of current = segtype
    THEN # for the correctness of DISCGARB, it is required that any
           object which is deleted from the map should also be marked as
           deleted from the SIN directory; see also request 9 (RESTART) #
        REF INT (count of current) := unused sin;
        # delete from map asynchronously #
        send data message (module2 send, mod2 del, sin of current,
                           ?, ?)
        # otherwise leave for DISCGARB to collect #
    FI;


# The only fault which can validly occur in SINMAN is PRL full.  Anything
  else stops the system since it shows serious error #
```

```
runtime error := (STRING s, INT i) VOID:
    IF (i & ABS 16rf0ff00ff) = ABS 16r80010018 # PRL full #
    THEN clear fault;
        return (i + ABS 16r01000000);
        GOTO restart
    ELSE stop system (stop slot, i)
    FI;


# MAIN LOOP #

restart:

DO CASE first argument

    IN  # 1: DYNAMIC (for VSM): b3 = SIN, b2 = 1,2 #
        IF goodsin (third argument)
        THEN CASE second argument IN
                IF ((count of current +:=:= tuc) & tucmask) = 0
                THEN notify discgarb (sin of current, type of current)
                FI,
                IF INT old = count of current -:=:= tuc;
                    (old < 0 ORF (old & pucmask) = 0)
                    ANDF (old & tucmask) = tuc
                THEN delete current
                FI
            ESAC;
            return (0)
        ELSE return (unknown request)
        FI,

        # 2: NEWSEG  b2 = init, b3 = access
             NEWSEG  b2 = -1, b3 = access, b4 = type, b5 = size #
        IF INT acc = third argument & cap access,
            INT size, type;
            BOOL new = second argument < 0;
            IF new
            THEN type := fourth argument;
                size := fifth argument;
                FALSE
            ELSE  goodsin (second argument)
                ORF (type := segtype; size := size of current; FALSE)
            FI
        THEN return (wrongsin)
        ELIF type < segtype
            ORF type > pdbtype
        THEN return (wrong type)
        ELIF acc = 0
            ORF ( new ANDF (acc & rwcaccess) = 0)
        THEN return (wrongaccess)
        ELSE INT sin =
                new segment (new segment wk,
                    IF type = segtype THEN acc ELSE rwaccess FI, size,
                    type);
            # this SIN was made current by 'new segment' #
            IF sin < 0
            THEN null (new segment wk);
                return (sin)
            ELIF INT rc =
                    CASE type IN
```

```
                (movecap (new segment wk, argslot); segtype),
                IF INT rc2 = new instance (argslot,
                                new segment wk, sin, acc);
                    new
                    ANDF rc2 >= 0
                THEN IF INT rc3 =
                             enter (argslot, 1, ?, ?, ?, ?);
                         rc3 < 0
                    THEN rc3
                    ELSE rc2
                    FI
                ELSE rc2
                FI,
                (current (sin);
                    maparray (new segment wk) [0] := -1;
                    # =>empty PDB #
                    enter current (argslot, acc))
            OUT wrong type
            ESAC;
        null (new segment wk);
        rc < 0
    THEN return (rc)
    ELSE IF NOT new
        THEN current (second argument);
            segment from current (from wk, read access);
            current (sin);
            segment from current (to wk, write access);
            move (maparray
                    (from wk) [0: size of current - 1 AT 0],
                maparray (to wk));
            null (from wk);
            null (to wk)
        FI;
        return (sin)
    FI
FI,

# 3: SEGFROMSIN b2 = sin #
IF INT sin = second argument & 65535;
    goodsin (sin)
THEN segment from current (argslot, second argument
                        & cap access);
    return (type of current)
ELSE return (wrongsin)
FI,

# 4: CAPFROMSIN b2=sin ! access #
IF INT sin = second argument & 65535;
    goodsin (sin)
THEN INT acc = second argument & capaccess;
    IF acc = 0
    THEN return (wrongaccess)
    ELSE INT rc =
            IF type of current <= segtype
            THEN segment from current (argslot, acc);
                type of current
            ELIF type of current = fdmtype
            THEN instance (argslot, acc)
            ELSE # 'pdbtype'? #
                enter current (argslot, acc)
```

```
            FI;
        return (rc)
    FI
ELSE return (wrongsin)
FI,

# 5: PRESERVE A0 = capability #
IF INT info = capinf (argslot);
    info < 0
    ORF info SHR 24 = swctype
THEN return2 (info, capinf0, capinf1, ?)
ELIF type of current = segtype
    ANDF (info & rcwcaccess) = 0
THEN return (wrong access)
     # C-type capabilities cannot be preserved #
ELSE preserve current;
    return2 (info, 0, info, 0)
FI,

# 6: REMOVE b2 = sin, b3 ignored #
IF goodsin (second argument)
THEN IF type of current = sdmtype
        ANDF ((count of current -:=:= puc) & puctucmask) = puc
    THEN delete current
    FI;
    return (0)
ELSE return (wrongsin)
FI,

# 7: SININF b2 = sin #
IF goodsin (second argument)
THEN return (word 1 of current)
ELSE return (wrongsin)
FI,

# 8: SINOFMFD #
return (mfdsin),

# 9: RESTART A0 = PUC table #
# The detail of this request is omitted since it can only be
  understood in the context of the complete restart system #

# 10: USE COUNT b2 = SIN #
return (IF goodsin (second argument) THEN curr use count ELSE
        wrongsin FI),

# 11: CAPINF A0 = capability #
(INT info = capinf (argslot);
    return2 (info, capinf0, capinf1, ?)),

# 12: CHANGESIZE b2 = change A0 = capability #
IF INT info = capinf (argslot);
    info < 0
THEN return (info)
ELIF type of current = segtype
THEN return (wrongtype)
ELIF (info & (write access ! wcap access)) = 0
THEN return (wrong access)
ELSE return (change current (second argument))
FI,
```

```
# 13: MAKESWC b2 = w0, b3 = w1 #
IF (third argument & ABS 16rc0000000) = 0
    ORF (third argument & ABS 16rf0000000) = 0 # for DIRMAN! #
THEN return (bad makeswc)
ELSE capsegcap (argslot, second argument, third argument);
    return (0)
FI,

# 14: START DISC GARB #
BEGIN
    INT c count = start disc garb (argslot, A 1);
    return2 (c count, mfdsin,
             discgarb word & dg serial mask, ?)
END,

# 15: STOP DISC GARB #
(disc garb word &:= NOT dg running bit; return (0)),

# 16: DESTROY (for DISCGARB) #
IF goodsin (second argument)
    ANDF (type of current = fdmtype ORF
          type of current = pdbtype)
THEN IF curr use count = 0
    THEN return (bad destroy)
    ELSE type of current := segtype;
        delete current;
        return (0)
    FI
ELSE return (wrongsin)
FI,

# 17: CHANGE CAP (for DISCGARB and RESTART) #
# A0 must point to a PRL slot created by request 18;  the
  capability must be destroyed by request 19, not left for
  PRLGARB #
IF INT sin = second argument & 65535,
       acc = second argument & rwaccess;
    goodsin (sin)
THEN INT prl0, prl1;
    INT prl =
        read prl capability (info cap, argslot, prl0, prl1) &
        65535;
    vsm (vsm kill, prl0, 0, 0);
    INT word0 = vsm (vsm sin, size of current, 0, sin);
    update prl capability (create cap, prl, word0,
    store capability ! hardware bit ! acc ! size of current);
    return (type of current)
ELSE return (wrong sin)
FI,

# 18: MAKE CAP (see request 17) #
BEGIN
    INT prl =
        create prl capability (create cap, argslot, -1,
        store capability ! hardware bit ! rwaccess ! 65535)
        & 65535;
    update prl capability (create cap, prl, 0, 0);
    return (0)
END,
```

```
        # 19: KILL CAP (see request 17) #
        BEGIN
            INT prl0, prl1;
            INT prl =
                read prl capability (info cap, argslot, prl0, prl1) &
                65535;
            vsm (vsm kill, prl0, 0, 0);
            update prl capability (create cap, prl, 0, 0);
            return (0)
        END


    OUT runtime error ("SINMAN", unknown request)
    ESAC
OD
```

VIRTUAL STORE MANAGER (VSM)

VSM receives requests to issue a 'mcaddr' given a SIN. It sees whether a mcaddr has been allocated already, and if not procures one by asking RSM (by message). A use count is maintained, giving the number of processes which have been issued with each particular mcaddr. This is decremented whenever a (non-reply) message is received indicating that the sending process has deleted its PRL entry containing that mcaddr. RSM is informed about mcaddrs which have gone out of use - that is, their use counts have fallen to zero. VSM calls SINMAN to indicate that capabilities for the object underlying a particular SIN are in issue (SINMAN request 'dynamic'). VSM also performs the functions of opening and closing windows on large segments. Moving windows around over a large segment is done by RSM. STOREMAN provides the user interface to all window functions.

```
SLOT perm stop          = P4,
     chan rsm           = P5,
     chan1              = P6,
     chan2              = P7,
     sinman slot        = P8,
     sinhash slot       = P9,
INT  mcblk slot         = 10; # for mcarray: P10 to P13 #

runtime error := (STRING s, INT i)VOID:
   stop system(perm stop, i);

# Entry requests #
INT new object          = 11, # RSM #
    ensure              = 1,  # RSM #
    delete object       = 5,  # RSM #
    dynamic             = 1,  # SINMAN #
    use count           = 10; # SINMAN #

INT dyn use             = 1,
    dyn free            = 2;

PROC sinman = (INT entry, sin)VOID:
    enter(sinman slot, dynamic, entry, sin, ?, ?);

# Return codes - not fault numbers, as VSM is not user-called #
INT unknown mcaddr      = ABS16r80160001, # unknown to VSM - e.g., a
                                            SYSGENed segment #
    window mcaddr       = ABS16r80160002, # operation not available for
                                            windows #
    closing not window = ABS16r80160003,
    unknown entry       = ABS16r80160004;

# Message channel initialisation #
SLOT rsmsend, rsmreply;
setup send with reply(chan rsm, rsmsend, data reply message,
                      rsmreply, data message);
```

```
INT rsm1, rsm2, rsm3, rsm4;

PROC rsm = (INT a, b, c, d)INT:
   BEGIN
      send data message wait event(rsmsend, a, b, c, d);
      UNTIL messages(rsmreply) = 0 DO wait event OD;
      receive data message(rsm reply, rsm1, rsm2, rsm3, rsm4);
      rsm1
   END;

SLOT rec1, rep1, rec2;

# Channel 2 is non-reply; it is used only for delete messages in which
  a process indicates that it is no longer interested in a certain
  mcaddr. #

setup receive(chan1, rec1, data reply message);
setup reply(rep1, data reply message);
setup receive(chan2, rec2, data message);


# Mcarray #

# The main data structure is 'mcarray', a set of blocks indexed by
  'mcaddr'.  Each entry is one word:

    ----------------------------------
   |     SIN      | w | d | use count |
    ----------------------------------
    31          16 15  14  13        0

  where 'w' indicates whether the entry is for a window and 'd' is used
  to indicate deleted entries.  Use count is count of processes with a
  capability using this mcaddr. #

INT mccapsegmin         = 4, # first MC capability segment #
    mccapsegmax         = 14; # last MC capability segment #
INT mcslots             = (mccapsegmax - mccapsegmin + 1) * 256,
                          # number of MC slots #
    slotspermcblk       = 896; # size of each block of 'mcarray' #
INT mcblks              = (mcslots - 1) % slotspermcblk; # = 3 #
[0:mcblks]REF[]INT mcarray;

# layout of each entry #
INT mcsinshft           = 16,
    mcsinmsk            = lhword,
    mcwindowmk          = 32768,
    mcdeletemk          = 16384, # indicates mcaddr is currently unknown
                                   to VSM #
    mc ucmsk            = 16383;

# Initialisation of the array #
FOR i FROM 0 TO mcblks
DO REF[]INT v = maparray(P(mcblkslot + i)); mcarray[i] := v;
   FOR j FROM 0 TO slotspermcblk - 1 DO v[j] := mcdeletemk OD
OD;
```

```
PROC mcelement = (INT mcaddr)REF INT:
   BEGIN
      INT count = ((mcaddr SHR 28) - mccapsegmin) * 256
                  + ((mcaddr SHR 16) & 255);
      mcarray[count % slotspermcblk][count %* slotspermcblk]
   END;


# SIN map #

# A bit map is maintained, recording whether a SIN is currently active
  in the virtual memory, i.e., whether it has an mcaddr (as a segment,
  not as a window). #

INT sinmax = 7999; # largest SIN #
INT sinmapmax = sinmax % bitswidth; # number of words in bitmap #
[0 : sinmapmax]INT sinmap;
FOR i FROM 0 TO sinmapmax DO sinmap[i] := 0 OD;

PROC known sin = (INT sin)BOOL:
   sin >= 0 ANDF sin <= sinmax ANDF
   (sinmap[sin % bitswidth] & (1 SHL (sin %* bitswidth))) = 0;

PROC marksin = (INT sin)VOID:
   sinmap[sin % bitswidth] !:= 1 SHL (sin %* bitswidth);

PROC unmarksin = (INT sin)VOID:
   sinmap[sin % bitswidth] &:= NOT(1 SHL (sin %* bitswidth));


# SIN hash #

# A hash table is used to convert SINs into mcaddrs.  Each entry is a
  half word, being the most significant half of the mcaddr.  A marker is
  available to indicate when a hash table entry is deleted.  Note that
  the hash table does not contain the normal 'key' and 'value'; instead
  it contains an index into 'mcarray' where the key can be found. #

REF[]INT sinhash = map array(sinhash slot);
INT sinhashsize = 2557,
    sinincr = 13,
    sin delete mk = ABS 16r0100;

PROC sinval = (INT sin)INT:
   sinhash[sin % 2] SHL (IF ODD sin THEN - 16 ELSE 0 FI);

PROC setsinval = (INT sin, val)VOID:
   sinhash[sin % 2] :=
      IF ODD sin THEN (sinhash[sin % 2] & ABS16rffff) ! (val SHL 16)
                 ELSE (sinhash[sin % 2] & ABS16rffff0000) ! val FI;

# The following are the routines which access the SIN hash table #

PROC new mcaddr = (INT sin, BOOL window, INT size, base,
                   REF INT mc)INT:   # allocate new mcaddr #
   IF rsm(new object, size, base, sin !
                        (IF window THEN 1 SHL 31 ELSE 0 FI)) = 0
   THEN rsm1 # rc #
   ELSE INT addr = mc := rsm2;
      # put in mcarray #
```

```
      mcelement(addr) := ((sin SHL mcsinshft) !
                          (IF window THEN mcwindowmk ELSE 0 FI)) + 1;
      IF NOT window
      THEN # put in 'sinhash' #
           BEGIN
              INT try := sin %* sinhashsize;
              UNTIL (sinval(try) & sindeletemk) = 0
              DO IF (try +:= sinincr) >= sinhashsize THEN
                                 try -:= sinhashsize FI OD;
              setsinval(try, addr SHR 16)
           END;
           # put in SIN bit map #
           marksin(sin)
      FI;
      0 # rc #
   FI;

PROC old mcaddr = (INT sin, incr)INT:
   BEGIN
      # SIN must be already known to 'sinhash' #
      # windows in mcarray are ignored #
      INT try := sin %* sinhashsize;
      INT addr;
      INT val;
      UNTIL ((val := sinval(try)) & sindeletemk) = 0
      ANDF(addr := val SHL 16;
           (mcelement(addr) & (mcsinmsk ! mcwindowmk ! mcdeletemk))
           = (sin SHL mcsinshft))
      DO IF (try +:= sinincr) >= sinhashsize
         THEN try -:= sinhashsize FI OD;
      mcelement(addr) +:= incr;
      addr
   END;

PROC delete mcaddr = (INT mcaddr)VOID:
   IF (mcaddr SHR 28) >= mccapsegmin
   THEF REF INT mc = mcelement(mcaddr);
      (mc & mcdeletemk) = 0 ANDF ((mc -:= 1) & mcucmsk) = 0
   THEN INT sin = mc SHR mcsinshft;
        BOOL window = (mc & mcwindowmk) = 0;
        mc := mcdeletemk;
        IF NOT window
        THEN INT try := sin %* sinhashsize;
             UNTIL sinval(try) SHL 16 = mcaddr
             DO IF (try +:= sinincr) >= sinhashsize
                THEN try -:= sinhashsize FI OD;
             setsinval(try, sindeletemk);
             unmarksin(sin)
        FI;
          # Note that the temporary use count must not be decremented
            until the object has been written to disc. There is a danger
            that the object will be deleted after it has been decided to
            write, it but before it has been written - hence the two
            separate entries to SINMAN #
        BOOL ensure = enter(sinman slot, usecount, sin, ?, ?, ?) = 1;
        rsm(delete object, mcaddr, ABS ensure, ?);
        sinman(dyn free, sin)
   FI;
```

```
# MAIN LOOP #

DO
 WHILE messages(rec1) = 0 AND messages(rec2) = 0 DO waitevent OD;

  WHILE messages(rec1) = 0
   DO INT a,b,c,d;
      receive reply data message(rec1, rep1, a,b,c,d);
      INT ans data := 0;
      INT ans = CASE a IN
         # 1: SIN to mcaddr; b = size, c = base, d = SIN #
         IF known sin(d)
         THEN old mcaddr(d, 1)
         ELSE INT mc;
              INT rc = new mcaddr(d, FALSE, b, c, mc);
              IF rc < 0 THEN runtime error("", rc) FI;
              sinman(dyn use, d); mc
         FI,

         # 2: mcaddr to SIN, not for windows #
         IF INT mc = mcelement(b);
            (mc & mcdeletemk) = 0
         THEN unknown mcaddr
         ELIF (mc & mcwindowmk) = 0
         THEN window mcaddr
         ELSE mc SHR mcsinshft
         FI,

         # 3: ensure SIN #
         IF known sin(b)
         THEN rsm(ensure, old mcaddr(b, 0), ?, ?)
         ELSE 0
         FI,

         # 4: disc addr to mcaddr (for SINMAN only) #
         IF known sin(d)
         THEN old mcaddr(d, 1)
         ELSE INT mc; INT rc = new mcaddr(d, FALSE, b, 0, mc);
              IF rc < 0 THEN runtime error("", rc) FI;
              mc
         FI,

         # 5: close window (for STOREMAN only) #
         IF (mcelement(b) & mcwindowmk) = 0
         THEN delete mcaddr(b); 0
         ELSE closing not window
         FI,

         # 6: converse of entry 1 #
         (delete mcaddr(b); 0),

         # 7: open window; b = size, c = base, d = SIN #
         BEGIN
            INT rc = new mcaddr(d, TRUE, b, c, ans data);
            IF rc >= 0 THEN sinman(dyn use, d) FI; rc
         END

      OUT runtime error("VSM", unknown entry); unknown entry
      ESAC;
```

```
      return data message(rep1, ans, ans data, 0, 0)
   OD;

   WHILE messages(rec2) = 0 ANDF messages(rec1) = 0
      DO
      INT a,b,c,d; receive data message(rec2, a,b,c,d);
      delete mcaddr(a)
      OD

OD
```

STOREMAN

STOREMAN provides the public interface to administrative operations upon capabilities - mainly capabilities for segments of memory, whence the name. When requested to create new capabilities or to find information about existing ones, STOREMAN calls SINMAN (in the same process). When requested to perform such operations as moving windows or ensuring that a segment is up to date on disc the real store manager is activated (by message). Requests for the creation and deletion of windows are handled by messages to VSM. STOREMAN also communicates with MODULE, the disc allocation manager, to ensure, where necessary, that the allocation map is properly written to disc.

```
BEGIN
SLOT mn rsm                 = P 4,
     info perm              = P 6,
     cap perm               = P 7,
     sinman                 = P 8,
     messagename vsm        = P 9,
     messagename module     = P 10,
     # stack                = I 0 #
     capsegseg              = I 1;
SLOT send rsm,
     receive rsm, send vsm, receive vsm, send module, receive module;
SLOT a0 = A 0;


# Runtime error #
runtime error := (STRING s, INT i) VOID:
    # for during STOREMAN initialisation #
    (return fault (i); set return code (i); stop);

# Set up message channel #
# The channels are used to send messages to RSM for some requests, to VSM
  for others, and to MODULE to ensure that the disc allocation map is up
  to date. #
set up send with reply (mn rsm, send rsm, data reply message,
    receive rsm, data message);
setup send with reply (messagename vsm, send vsm, data reply message,
    receive vsm, data message);
setup send with reply (messagename module, send module,
    data reply message, receive module, data message);

# STOREMAN creates new segments on demand.  In the case of new capability
  segments it attempts to do this by subsegmentation of a special segment
  known as 'capsegseg'.  This is done in order to keep a lot of small
  objects together as one swap unit. #
INT prlref, inf;
INT endval = segsize (capsegseg) - 1;
INT i = read prl capability (info perm, capsegseg, prlref, inf);
prlref &:= lhword;
REF [] INT a = maparray (capsegseg);
a [0] := endval;
INT bit31 = 1 SHL 31;
```

```
# Interface with SINMAN, RSM, VSM #
INT newseg                  = 2, # SINMAN #
    cap from sin            = 4, # SINMAN #
    sininf                  = 7, # SINMAN #
    capinf                  = 11, # SINMAN #
    changesize              = 12, # SINMAN #
    ensureok                = 1, # RSM #
    outform                 = 2, # RSM #
    move window             = 9, # RSM #
    vsm open                = 7, # VSM #
    close window            = 5, # VSM #
    segtype                 = 1,

# Return codes #
    unknown request                 = ABS 16r80040001,
    accessfault                     = ABS 16r80040002,
    closing not store type          = ABS 16r80040003,
    init from non store             = ABS 16r80040004;

PROC result = (INT i) VOID:
    (IFi < 0 THEN return fault (i) FI; return (i));

PROC find = (INT sought) INT:
    BEGIN
        SKIP # this procedure which attempts to find 'sought' words in
                capsegseg #
    END;

# The next three procedures construct a procedural interface using the
  message system #
INT m1,m2,m3,m4;

PROC send to rsm = (INT request, w3, w4) INT:
    BEGIN
        INT ref, info;
        read prl capability (info perm, a0, ref, info);
        send data message wait event (send rsm, request, ref, w3, w4);
        WHILE messages (receive rsm) = 0 DO wait event OD;
        receive data message (receive rsm, m1, m2, m3, m4);
        m1
    END; # send to rsm #

PROC send to vsm = (INT a, b, c, d, REF INT mc) INT:
    BEGIN
        send data message wait event (send vsm, a, b, c, d);
        WHILE messages (receive vsm) = 0 DO wait event OD;
        INT v1, v3, v4;
        receive data message (receive vsm, v1, mc, v3, v4);
        v1
    END; # send to vsm #

PROC ensure map = VOID:
    BEGIN
        send data message wait event (send module, 15, ?, ?, ?);
        WHILE messages (receive module) = 0 DO wait event OD;
        INT m1, m2, m3, m4;
        receive data message (receive module, m1, m2, m3, m4)
    END; # ensure map #
```

```
# MAIN LOOP #

runtime error := (STRING s, INT i) VOID: (result (i); GOTO recover);

recover:

DO CASE first argument IN

        # 1: ENSUREOK #
        BEGIN
            INT rsm rc = send to rsm (ensure ok, ?, ?);
            ensure map;
            result (rsm rc)
        END,

        # 2: OUTFORM #
        result (send to rsm (outform, ?, ?)),

        # 3: CHANGESIZE  #
        (movecap (a0, n0);
            INT n = enter (sinman, changesize, second argument,
                           ?, ?, ?);
            IF n >= 0 THEN movecap (n0, a0) FI;
            result (n)),

        # 4: NEWSEG, second argument is access, model capability in A0
          or NEWSEG, -1, access, type, size #
        IF second argument = -1
        THEN movecap (a0, n0);
            INT cap = enter (sinman, capinf, ?, ?, ?, ?);
            IF cap < 0
            THEN result (cap)
            ELIF (cap SHR 24) = segtype
            THEN result (init from non store)
            ELIF
            # check validity of requested access #
                BOOL correct =
                IF (cap & read access) = 0
                THEN (second argument & (rcap access ! wcap access)) = 0
                ELIF (cap & rcap access) = 0
                THEN (second argument & (read access ! write access)) = 0
                ELSE FALSE
                FI;
                correct
            THEN INT n = enter (sinman, newseg, cap & rhword,
                                second argument, ?, ?);
                result (IF n > 0 THEN movecap (n0, a0); 0 ELSE n FI)
            ELSE result (access fault)
            FI
        ELSE INT n, pos;
            IF (third argument & (rcapaccess ! wcapaccess)) = 0
                ANDF fourth argument = segtype
                ANDF (pos := find (fifth argument)) > 0
            THEN FOR i FROM pos + 1 TO pos + fifth argument
                DO a [i] := -1 OD;
                create prl capability (cap perm, a0,
                    prlref ! (pos + 1),
                    store capability ! bit31 !
                    (third argument & capaccess) ! fifth argument);
                result (0)
```

```
        ELSE n := enter (sinman, newseg, second argument, third
                         argument, fourth argument, fifth argument);
              # use normal segment creation if capsegseg is full #

            result (IF n > 0 THEN movecap (n0, a0); 0
                             ELSE n FI)
        FI
    FI,

    # 5: CAPINF #
    (movecap (a0, n0);
        INT a, b, c, d;
        enter2 (sinman, capinf, ?, ?, ?, ?, a, b, c, d);
        return2 (a, b, c, d)),

    # 6: GETSIZEACCESS #
    (movecap (a0, n0);
        INT sin = enter (sinman, capinf, ?, ?, ?, ?);
        IF sin > 0 # negative=> error return #
        THEN INT n =
                enter (sinman, sininf, sin & rhword, ?, ?, ?);
            INT size = (n & (byte2 ! rhword)),
                access = (sin & byte2) SHL 8;
            result (size ! access)
        ELSE result (sin)
        FI),

    # 7 OPEN WINDOW b2 = base, b3 = size+access, A0 = segment #
    IF movecap (a0, n0);
        INT sin = enter (sinman, capinf, ?, ?, ?, ?);
        sin < 0
    THEN result (sin)
    ELIF (sin & third argument & cap access) =
            (third argument & cap access)
    THEN result (access fault)
    ELSE INT mcaddr;
        INT rc =
            send to vsm (vsm open, third argument & 65535, # size #
                second argument, # base #  sin & rhword, mcaddr);
        IF rc < 0
        THEN result (rc)
        ELSE create prl capability (cap perm, a0, mcaddr,
                third argument ! hardware bit ! store capability);
            result (0)
        FI
    FI,

    # 8 MOVE WINDOW b2 = new base, b3 = size+access, A0 = window #
    result (send to rsm (move window, second argument,
            third argument)),

    # 9 CLOSE WINDOW A0 = window #
    (INT ref, info;
        INT prl =
            read prl capability (info perm, a0, ref, info) &
            65535;
        IF (info & hardware bit) = 0
        THEN result (closing not store type)
        ELIF INT d;
            INT n = send to vsm (close window, ref, ?, ?, d);
```

```
            n = 0
        THEN result (n)
        ELSE update prl capability (cap perm, prl, 0, 0);
            result (0)
        FI),

    # 10 CLEANSE A0 = cap #
    result (unknown request), # no longer used #

    # 11: DETAILS #
    (movecap (a0, n0);
        INT sin = enter (sinman, capinf, ?, ?, ?, ?);
        IF sin > 0
        THEN INT n =
                enter (sinman, sininf, sin & rhword, ?, ?, ?);
            return2 (0, # success #  n & 0yte3, # size #
                sin & cap access, # access #  sin SHR 24 # type #)
        ELSE result (sin)
        FI),

    # 12: NEW INSTANCE (for PDBs, mainly) #
    BEGIN

        movecap (a0, n0);
        INT info = enter (sinman, capinf, ?, ?, ?, ?);
        IF info < 0
        THEN result (info)
        ELSE INT rc = enter (sinman, cap from sin, info, ?, ?, ?);
            IF rc >= 0 THEN movecap (n0, a0) FI;
            result (rc)
        FI
    END
OUT result (unknown request)
ESAC
OD


END # STOREMAN #
```

DIRSTRUCT

This material defines the data structures used in directory segments. It is used to produce an environment file which is then used in the compilation of DIRMAN, DISCGARB, and RESTART - all of which require knowledge of directory formats.

```
# A directory segment consists of a header followed by a sequence of
  blocks #
INT header size = 6,
    block size  = 6;

# Blocks are linked by their first word #
# OP NEXT = (INT i) INT: directory[i] & rhword #
# OP LAST = (INT i) INT: directory[i] SHR 16 #

# An entry is held in a linked list of blocks. The block(s) containing
  the textual name are followed by block(s) containing the associated
  entry data.  Their structure is as follows:

  1. Block(s) containing the name.
     a) The first block of an entry:
            word0   link
            word1   number of blocks used for entry
            word2   byte1:  no. of characters in name
                    followed by first 3 characters
            remaining words contain 1 char/byte
     b)Any subsequent blocks:
            word0   link
            remaining words:  1 char/byte

  2. Block(s) containing entry data:
     a) First block:
            word0   link
            word1   access matrix
            word2   cap info 0
            word3   cap info 1
            word4   time of preservation
            word5   date of preservation

  Unused blocks are kept on a free chain. #


INT access matrix offset          = 1, # useful block offsets #
    cap info 0 offset             = 2,
    cap info 1 offset             = 3,
    time of preservation offset   = 4,
    date of preservation offset   = 5,
    max offset                    = block size - 1;

INT blocks total                  = 0, # indices to directory header #
    blocks free                   = 1,
    entries total                 = 2,
    freelist head                 = 3,
    hash table start              = 4,
    hash table end                = 5;
```

```
# An entry is kept on one of several separate chains. The appropriate
  chain is selected by a hash function on the name of the entry. #

INT hash table size = hash table end - hash table start + 1;

INT null = 0; # the null index to a directory #

# Format of an access matrix word:

   byte1   access field for those with W-status
   byte2     "      "    "    "    "   X-status
   byte3     "      "    "    "    "   Y-status
   byte4     "      "    "    "    "   Z-status

 Format of these fields:
           directory     pdb           segment       SWC
           ---------     ---           -------       ---
 d7        delete        delete        delete        delete
 d6        update        update        update        update
 d5        create
 d4        W-status
 d3        alter         alter         alter         alter
 d2        X-status      modify        write
 d1        Y-status      inspect       read
 d0        Z-status      link          execute       software #

# bits of an access field for a directory #
INT delete                          = ABS 16r00000080,
    update                          = ABS 16r00000040,
    create                          = ABS 16r00000020,
    w                               = ABS 16r00000010,
    alter                           = ABS 16r00000008,
    x                               = ABS 16r00000004,
    y                               = ABS 16r00000002,
    z                               = ABS 16r00000001;

# Format of capability information words:

   1. For store capabilities:
         cap info word 0 - not used
         cap info word 1 - d0-d15 sin
                           d16-d21 access
                           d24-d27 store type (1=seg,2=dir,3=pdb)
                           d28-d31 file type = store = 0

   2. For software capabilities (SWCs):
         Entries for SWCs will be identified by the file type field
         (d28-d31 of cap info word 1) being not equal to 'store'(= 0).
         d26-d29 of this word indicates type of SWC as usual. #

INT file type field = ABS 16rf0000000,
    store = 0;


ENVIRON DIRSTRUCT # directive to A68C to preserve the environment #
```

DIRMAN

DIRMAN manages all file directories. A file directory relates text names to access statuses and SINs. DIRMAN communicates with SINMAN (in the same process) to procure a capability for an object of given SIN, to discover the SIN of an object for which a capability is to hand, and to request SINMAN to decrement or increment reference counts. When following the directory structure to interpret a multi-part file name, DIRMAN uses the 'segfromsin' request to SINMAN to obtain a capability for the directory segment instead of creating a further instance of itself. DIRMAN is compiled in the environment constituted by DIRSTRUCT.

```
SLOT chan ensurer           = P 5,
     stop perm              = P 6,
     chan clock             = P 7,
     # stack                = I 0 #
     # heap                 = I 1 #
     sinman                 = I 2,
     # not used             = I 3 #
     current file           = getslot,
     working directory      = R 0,
     working directory info slot = R 1,
     file spec              = A 0,
     return slot            = A 0,
     cap to be filed        = A 1,
     a1                     = A 1,
     n0                     = N 0;
SLOT ensurer send, clock send, clock reply;


# Initialisation code - sets up channels and initialises 'runtime error'
  #
set up send (chan ensurer, ensurer send, data message);
set up send with reply (chan clock, clock send, data reply message,
    clock reply, data message);

PROC sys error = (STRING s, INT i) VOID: stop system (stop perm, i);

runtime error := sys error; # this is the standard setting when DIRMAN
                              is running. #

INT sin of current file;

# Directory structure #

# See DIRSTRUCT for definition and explanation of the structure of a
  directory #

REF [] INT directory = map array (current file);

# Blocks are linked by their first word #
OP NEXT = (INT i) INT: directory [i] & rhword;
OP LAST = (INT i) INT: directory [i] SHR 16;

# An entry is kept on one of several separate chains. The appropriate
  chain is selected by a hash function on the name of the entry. #

PROC hash on name = INT: hash table start;
```

```
MODE ENTRY = STRUCT (INT name ptr, entry ptr);

# File Specification #

REF [] INT spec = map array (file spec);
INT separator = ABS ".",
    terminator = ABS "/";
[0: 63] INT name; # current component name #

# Declarations of quantities to do with access status #
INT status; # access status (least significant byte) to current file #
INT full status             = ABS 16r00000037,
    full access             = ABS 16r00370000,
    deletable               = ABS 16r88888888,
    max access request      = ABS 16r80000000,
    read write              = ABS 16r00060000,
    swc access              = ABS 16r00010000,
    read only               = ABS 16r00020000;

# Other constants #

# Entry arguments to SINMAN #
INT seg from sin            = 3,
    cap from sin            = 4,
    preserve cap            = 5,
    remove cap              = 6,
    sininf                  = 7,
    puc                     = 0,# permanent use count #
    changesize              = 12,
    newseg                  = 2,
    make swc                = 13,
    directory type          = 2, # type value for SINMAN #
    segtype                 = 1,
    swctype                 = 4,

# Fault and return codes #
    success                 = 0,
    inc proc count          = ABS 16r1000000,
    # for adding to fault return codes from SINMAN #
    dirman fault = ABS 16r800a0000; # dirman fault series number #

# Fault codes specific to DIRMAN #
INT invalid entry request                   = dirman fault ! 1,
    no access permitted                     = dirman fault ! 2,
    creation not permitted                  = dirman fault ! 3,
    file not directory                      = dirman fault ! 4,
    no such entry                           = dirman fault ! 5,
    invalid access matrix                   = dirman fault ! 6,
    deletion not permitted                  = dirman fault ! 7,
    directory full                          = dirman fault ! 8,
    requested access not permitted          = dirman fault ! 9,
    directory already initialised           = dirman fault ! 10,
    update not permitted                    = dirman fault ! 11,
    alter not permitted                     = dirman fault ! 12,
    undeletable entry not permitted         = dirman fault ! 13,
    invalid file spec                       = dirman fault ! 14,
    invalid file name                       = dirman fault ! 15,
    zero access requested                   = dirman fault ! 16,
    conflict with sinman about type         = dirman fault ! 17,
    seg too small to be a directory         = dirman fault ! 18,
```

```
    argument exseg wrong                    = dirman fault ! 19,
    no argument exseg                       = dirman fault ! 20,
    segment cannot be reserved              = dirman fault ! 21,
    update with wrong type                  = dirman fault ! 22,
    update with wrong access                = dirman fault ! 23;


# Segment reservation #

BOOL writing                = TRUE,
     reading                = FALSE;

PROC reserve for = (BOOL write, SLOT s) VOID:
    BEGIN
        PROC reserve fault = (STRING s, INT i) VOID:
             fail (segment cannot be reserved);

        runtime error := reserve fault;
        IF write
        THEN reserve for writing (s)
        ELSE reserve for reading (s)
        FI;
        runtime error := sys error
    END; # reserve for #

# Ensure #
PROC ensure = (INT sin) VOID:
    send data message (ensurer send, IF sin < 0 THEN 1 ELSE 2 FI,
        sin of current file, sin, ?);

# Retrieve #
PROC retrieve = VOID:
    BEGIN
        retrieve specified directory;
        INT type =
            IF (name [0] SHR 24) < 2
            THEN # special case #
                reduce directory status
            ELSE acquire named entry (cap from sin)
            FI;
        movecap (current file, return slot);
        return (type)
    END; # retrieve #

PROC retrieve specified directory = VOID:
    BEGIN
        INT spec char ptr := 0,
            spec word ptr := 0,
            spec byte ptr := 2;

        INT char, name word ptr, name byte ptr, name char count;
        INT case mask = ABS 8r337;

        PROC bad spec = (STRING s, INT i) VOID:
            fail (invalid file spec);

        runtime error := bad spec;

        INT current spec word := spec [0];
        INT spec length = current spec word SHR 24;
```

```
        PROC nextchar = INT:
            BEGIN
                SKIP # this procedure which gets the next character from
                       the argument string #
            END;

        IF spec length = 0
        THEF ((current spec word SHR 16) & 255) = separator
        THEN nextchar
        FI;

        BOOL finished := FALSE;

        UNTIL finished
        DO runtime error := bad spec;
            name word ptr := 0;
            name byte ptr := 3;
            name [0] := separator SHL 16;
            # an omitted section of code extracts successive components
              of a file name #
            runtime error := sys error;

            name [0] !:= name char count SHL 24;
            IF  finished THEN acquire named entry (seg from sin)
            # forcing retrieval of a segment even though the object is a
              directory #
            FI
        OD
    END; # retrieve specified directory #

PROC reduce directory status = INT:
    # This retrieves an enter capability for a directory, possibly with a
      reduced access status #
    BEGIN
        INT type := directory type;
        INT access := status SHL 16;
        IF second argument = max access request
        THEN check and set requested access (access, FALSE)
        FI;
        type := enter (sinman, capfromsin, sin of current file !
                       access,
                ?, ?, ?);
        IF type < 0
        THEN fail (type + inc proc count)
        ELIF type = directory type
        THEN fail (conflict with sinman about type)
        FI;
        movecap (n0, current file);
        type
    END; # reduce directory status #

PROC acquire named entry = (INT sinman entry request) INT:
    BEGIN
        reserve for (reading, current file);
        INT index = entry ptr OF locate named entry;
        INT cap info 0 = directory [index + cap info 0 offset],
            cap info 1 = directory [index + cap info 1 offset];
        status := calculate rights (index) & full status;
        INT access := status SHL 16;
        IF access = 0
```

```
        THEN release reservation;
             fail (no access permitted)
        FI;
        IF sinman entry request = capfromsin
        THEN IF second argument = max access request
             THEN check and set requested access (access, TRUE)
             FI
        ELSE access := read write
        FI;
        INT type := 4; # will later be reset if file type = store #

        IF (cap info 1 & file type field) = store
        THEN INT sin = cap info 1 & rhword;
            type := enter (sinman, sinman entry request, sin ! access,
                    ?, ?, ?);
            IF type < 0
            THEN release reservation;
                fail (type + inc proc count)
            ELIF type = (cap info 1 SHR 24)
            THEN release reservation;
                fail (conflict with sinman about type)
            FI;
            sin of current file := sin
        ELSE
            # a software capability #
            enter (sinman, make swc, cap info 0, cap info 1, ?, ?)
        FI;

        movecap (n0, current file);
        release reservation;

        IF sinman entry request = seg from sin
            ANDF type = directory type
        THEN fail (file not directory)
        FI;
        type
    END; # acquire named entry #

PROC check and set requested access = (REF INT access, BOOL seg res)
    VOID:
        BEGIN
            # second argument is requested access #
            INT request = second argument & full access;
            IF request = 0
            THEN IF seg res THEN release reservation FI;
                fail (zero access requested)
            FI;
            IF (request &  access) = 0
            THEN IF seg res THEN release reservation FI;
                fail (requested access not permitted)
            FI;
            access := request
        END; # check and set requested access #
```

```
PROC locate named entry = ENTRY:
    BEGIN
        INT name ptr, entry ptr;
        IF  exists named entry (name ptr, entry ptr)
        THEN release reservation;
            fail (no such entry)
        FI;
        (name ptr, entry ptr)
    END; # locate named entry #

PROC exists named entry = (REF INT name ptr, entry ptr) BOOL:
    BEGIN
        SKIP # the body of this procedure which searches for a named
               entry in the directory data structure #
    END;

PROC calculate rights = (INT index) INT:
    calc rights (directory [index + access matrix offset]);

PROC calc rights = (INT access matrix) INT:
    BEGIN
        INT rights := 0;
        IF (status & w) = 0 THEN rights !:= access matrix SHR 24 FI;
        IF (status & x) = 0
        THEN rights !:= (access matrix SHR 16) & 255
        FI;
        IF (status & y) = 0
        THEN rights !:= (access matrix SHR 8) & 255
        FI;
        IF (status & z) = 0 THEN rights !:= access matrix & 255 FI;
        rights
    END; # calc rights #

# Remove #
PROC remove = VOID:
    BEGIN
        retrieve specified directory;
        reserve for (writing, current file);
        ENTRY it = locate named entry;
        IF (calculate rights (entry ptr OF it) & delete) = 0
        THEN release reservation;
            fail (deletion not permitted)
        FI;
        # an omitted section of code does space management inside the
          directory segment #
        INT sin = IF INT oldcap info 1 = directory[entryptr OF it
                                          + cap info 1 offset];
                  (oldcap info 1 & file type field) = store
                  THEN oldcap info 1 & rhword
                  ELSE -1 # software capability #
                  FI;
        directory [entries total] -:= 1;
        release reservation;
        ensure (sin);
        return (success)
    END; # remove #
```

```
PROC add to free list = (INT head ptr, tail ptr, nblocks) VOID:
    BEGIN
        SKIP # the omitted body of this procedure which manages space
               in the  directory structure #
    END;

# Preserve #
PROC preserve = VOID:
    BEGIN
        # second argument is access matrix #
        retrieve specified directory;
        check name;
        reserve for (writing, current file);
        INT cap info 0, cap info 1;
        unseal cap to be filed (cap info 0, cap info 1);
        ensure (insert named entry
                (second argument, cap info 0, cap info 1));
        release reservation;
        return (success)
    END; # preserve #

PROC check name = VOID:
    BEGIN
            SKIP # this procedure which does a syntax check on a
              user-supplied file name #
    END;

PROC unseal cap to be filed = (REF INT cap info 0, cap info 1) VOID:
    # This procedure discovers the SIN of an object for which a
      capability is  about to be preserved #
    BEGIN
        # second argument is access matrix #
        IF (second argument & deletable) = 0
        THEN release reservation;
            fail (undeletable entry not permitted)
        FI;
        movecap (cap to be filed, n0);
        INT info, ignore;
        enter2 (sinman, preserve cap, ?, ?, ?, ?, info, cap info 0,
            cap info 1, ignore);
        IF info < 0
        THEN release reservation;
            fail (info + inc proc count)
        FI;
        IF  valid access matrix (second argument, info & full access)
        THEN release reservation;
            IF (info SHR 24) = swctype
            THEN enter (sinman, remove cap, info & rhword, ,puc, ?, ?)
            FI;
            fail (invalid access matrix)
        FI
    END; # unseal cap to be filed #

PROC valid access matrix = (INT matrix, max access allowed) BOOL:
    BEGIN
        SKIP # the omitted body of this procedure which does a syntax
               check on  access matrices #
    END;
```

```
PROC insert named entry = (INT access matrix,
       cap info 0, cap info 1) INT:
    BEGIN
       # returns SIN of an implicitly deleted file if there is one,
         otherwise -1 #
        PROC insert fail = (INT code) VOID:
            BEGIN
                release reservation;
                IF (cap info 1 & file type field) = store
                THEN enter (sinman, remove cap, cap info 1 & rhword,
                        puc, ?, ?)
                FI;
                fail (code)
            END; # insert fail #

        INT sin of a deleted file := -1;
        INT index, head ptr;
        BOOL new;
        IF INT dummy;
            (new :=  exists named entry (dummy, index))
        THEN IF (status & create) = 0
            THEN insert fail (creation not permitted)
            FI;
    # an omitted section of code handles space management in the
      directory and issues a failure code when the directory is full #
            INT offset := 1;
            FOR j FROM 0 TO nwords in name - 1 # declared
                                                in the omitted part #
            DO offset :=
                    IF offset = max offset
                    THEN index := NEXT index; 1
                    ELSE offset + 1
                    FI;
                directory [index + offset] := name [j]
            OD;
            index := NEXT index;
            directory [index + access matrix offset] := access matrix
         ELSE IF (calculate rights (index) & update) = 0
            THEN insert fail (update not permitted)
            FI;
            INT old cap info 1 =
                directory [index + cap info 1 offset];
            IF (old cap info 1 SHR 24) = (cap info 1 SHR 24)
            THEN insert fail (update with wrong type)
            FI;
            IF NOT valid access matrix (directory
                    [index + access matrix offset], cap info 1 &
                     full access)
            THEN insert fail (update with wrong access)
            FI;
            IF (old cap info 1 & file type field) = store
            THEN sin of a deleted file := old cap info 1 & rhword
            FI
        # when updating, access matrix given as argument is ignored #
        FI;
        directory [index + cap info 0 offset] := cap info 0;
        directory [index + cap info 1 offset] := cap info 1;
        set time and date of preservation (index);
        IF new
        THEN add to entry chain (head ptr, index);
```

```
            directory [entries total] +:= 1
        FI;
        sin of a deleted file
    END; # insert named entry #

PROC set time and date of preservation = (INT index) VOID:
    BEGIN
        INT ignore;
        send data message wait event (clock send, 0, ?, ?, ?);
        UNTIL messages (clock reply) = 0 DO wait event OD;
        receive data message (clock reply, ignore,
            directory [index + time of preservation offset],
            directory [index + date of preservation offset], ignore)
    END; # set time and date of preservation #

PROC get blocks = (INT n) INT:
    BEGIN
        SKIP # more space management #
    END;

PROC add to entry chain = (INT head ptr, tail ptr) VOID:
    BEGIN
        SKIP # yet more space management #
    END;


# Initialise directory #
PROC initialise directory = VOID:
    # this is done only to new directories #
    BEGIN
        reserve for (writing, current file);
        IF (directory [0] :=:= 0) = -1
        THEN release reservation;
            fail (directory already initialised)
        FI;
        INT block area size = segsize (current file) - header size;
        IF block area size < 0
        THEN release reservation;
            fail (seg too small to be a directory)
        FI;
        directory [blocks total] := directory [blocks free] := 0;
        directory [entries total] := 0;
        directory [freelist head] := null;

        FOR i FROM hash table start TO hash table end
        DO directory [i] := null OD;

        INT nblocks = block area size % block size;
        IF nblocks > 0
        THEN link new blocks and put on free list (hash table end + 1,
                nblocks)
        FI;

        release reservation;

    # successful initialisation - return code gives number of excess
      words (in range 0 to block size - 1) at the end of the directory
      segment #
        return (block area size - nblocks * block size)
    END; # initialise directory #
```

```
PROC link new blocks and put on free list = (INT head ptr,
        nblocks) VOID:
    BEGIN
        SKIP
        # the code called by the last procedure to set up the chains #
    END;

# Alter access #
PROC alter access = VOID:
    BEGIN
       # second argument: access rights to be removed;
         third  argument: access rights to be added #
        retrieve specified directory;
        reserve for (writing, current file);
        INT index = entry ptr OF locate named entry;
        INT matrix := directory [index + access matrix offset];
        IF (calc rights (matrix) & alter) = 0
        THEN release reservation;
            fail (alter not permitted)
        FI;
        matrix &:=  second argument;
        matrix !:= third argument;
        IF (matrix & deletable) = 0
        THEN release reservation;
            fail (undeletable entry not permitted)
        FI;
        INT max access allowed =
            IF INT cap info 1 = directory [index + cap info 1 offset];
                (cap info 1 & file type field) = store
            THEN cap info 1 & full access
            ELSE swc access
            FI;
        IF  valid access matrix (matrix, max access allowed)
        THEN release reservation;
            fail (invalid access matrix)
        FI;
        directory [index + access matrix offset] := matrix;
        release reservation;
        ensure (-1);
        return (success)
    END; # alter access #

# The next three sections are concerned with presenting information about
  directory contents to the caller. Details are given for one of them
  only #

# Examine directory #
PROC examine directory = VOID:
    BEGIN
        SKIP # the detail of filling a segment with a description of
               directory contents #
    END;

# File info #
PROC file info = (REF [] INT inf, REF INT ptr, INT index) VOID:
    BEGIN
        SKIP # getting the facts about a particular file #
    END;
```

```
# File details #
PROC file details = VOID:
    BEGIN
        SKIP # another examination precedure #
    END;

# File examine #
PROC file examine = VOID:
    BEGIN
        # 'exseg' is an argument segment into which the result goes #
        retrieve specified directory;
        INT nwords in name = (name [0] SHR 26) + 1;
        INT required exseg size =
            nwords in name + 2 + date of preservation offset;
        runtime error := (STRING s, INT i) VOID:
            fail (no argument exseg);
        IF INT data = seginf (a1);
            (data & rhword) < required exseg size
            ORF (data & write access) = 0
        THEN fail (argument exseg wrong)
        FI;
        runtime error := sys error;

        movecap (a1, return slot);
        REF [] INT exseg = map array (return slot);
        INT ptr := nwords in name + 1;
        reserve for (reading, current file);
        file info (exseg, ptr, entry ptr OF locate named entry);
        release reservation;

        # fill in rest of exseg #
        exseg [0] := status;
        exseg [1] := 1;
        FOR i FROM 0 TO nwords in name - 1
        DO exseg [i + 2] := name [i] OD;

        refine (return slot, read only ! required exseg size,
            return slot);
        return (success)
    END; # file examine #

PROC fail = (INT fault code) VOID:
    BEGIN
        return (fault code);
        GOTO restart # the beginning of the main loop #
    END;
```

```
# MAIN LOOP #
restart:
DO # to infinity #
    movecap (working directory, current file);
    INT working directory info = indinf (working directory info slot);
    status := (enter access SHR 16) & full status;
    sin of current file := working directory info & rhword;
    CASE first argument IN
        initialise directory, retrieve,
        remove, preserve,
        fail (invalid entry request), alter access,
        examine directory, file details,
        fail (invalid entry request), file examine
    OUT fail (invalid entry request)
    ESAC
OD

END # directory manager #
```

MAKEPACK

This program is used to inspect the contents of PDBs and also to update them. New PDBs are initially manufactured empty, and filled in by successive updates. The PDB contains specifications of the sizes of the capability segments to be created by LINKER, and also of their content. The various entry requests in the main loop are self-explanatory. The updating work is carried out on a copy of the PDB, the result being copied back to the original at the end of an update session. An attempt to update a PDB is rejected (rather than held up) if an update session is already in progress. A separate procedure called MAKEPDB furnishes a user interface to MAKEPACK.

```
SLOT    stop slot       = P 5,
        chan ensure     = P 6,
        sinman          = reserve slot (2),
        pdb slot        = getslot,
        true slot       = getslot,
        insp slot       = getslot;

runtime error := (STRING s, INT i) VOID: stop system (stop slot, i);

SLOT    delete slot;
setupsend (chan ensure, delete slot, data message);

SLOT    arg slot        = A 0,
        entry cap       = A 1,
        new arg         = N 0;

REF [] INT pdb          = maparray (pdb slot); # working version #
REF [] INT truepdb      = maparray (true slot); # original PDB #
REF [] INT testpdb      = maparray (new arg); # check interlock #
REF [] INT readpdb      = maparray (insp slot); # PDB to examine #


# Return codes #

INT inadequate status           = ABS 16r800b0001,
    bad capseg number           = ABS 16r800b0002,
    unknown request             = ABS 16r800b0003,
    bad pdb entry               = ABS 16r800b0004,
    sinman trouble              = ABS 16r800b0005,
    bad capseg size             = ABS 16r800b0006,
    update in progress          = ABS 16r800b0007,
    makepack competition        = ABS 16r800b0008,
    unimplemented request       = ABS 16r800b0009,
    weak supporting capability  = ABS 16r800b000a,
    no update session           = ABS 16r800b000b,
    invalid segment type        = ABS 16r800b000c,
    no further entry            = ABS 16r800b000d,
    pdb full                    = ABS 16r800b000e,
    capseg not set              = ABS 16r800b000f,
    too many requests           = ABS 16r800b0010;
INT rc incr = ABS 16r01000000; # to add to return codes passed back #
```

```
# Format of the PDB #
# The PDB is at present a single block with a standard prefix #

INT blksize             = 7 * 128,
    readwrite access    = read access ! write access,
    rcwcaccess          = rcapaccess ! wcapaccess,
    inspect             = read access,
    modify              = write access,
    link                = execaccess,
    accessmask          = ABS 16r3f0000;

INT pcapsegno           = 4,
    icapsegno           = 5,
    rcapsegno           = 6;

INT segtype             = 1,
    fdmtype             = 2,
    pdbtype             = 3,
    swctype             = 4; # VMO types #

# The first 7 words of the PDB are in a standard format.  The first three
  words specify respectively the number of entries in the PDB, the
  pointer to the first word, and the pointer beyond the last word of the
  free area.  Each entry in the PDB occupies three consecutive words. #

INT pdb entries         = 0,
    free pointer        = 1,
    string area         = 2;

# The next three words of the PDB specify the number of doubleword
  entries to be allocated for the P,I,R capability segments.  The format
  of each entry is as follows

   31          24 23          16              8             1
   ---------------------------------------------------------------
  ! ////////////  !    Access    ! ///////////  ! Maximum entry !/!
   ---------------------------------------------------------------

  If I and R capability segments are not required, the corresponding word
  in the PDB is -1 #

INT isize               = -1, # base of size field #
    pcapsegsize         = 3,
    icapsegsize         = 4,
    rcapsegsize         = 5;

# The final word of the header in the PDB is used for a lock to prevent
  simultaneous update sessions taking place on a single PDB.  This word
  is set to 0 during a makepack session, but will be -1 at all other
  times. #

INT makepack lock       = 6,
    first entry         = 7;


INT pdb entry size      = 3;

# Each three-word entry in the PDB specifies a capability segment and
  offset, together with the entry type.  The remaining fields depend on
  the type #
```

```
INT capsegshift         = 28,
    vmotypeshift        = 16,
    typeshift           = 10,
    sinbit              = 1;

INT offsetmask          = ABS 16r1fe,
    pdbslotmask         = ABS 16r700001fe,
    capsegmask          = ABS 16r70000000;

# The entry type specifies the nature of the entry.  The interpretation
  of other fields depends on the entry type. #

INT workspace segment           = 1,
    init from file title        = 2,
    init from sin               = 3,
    cap from file title         = 4,
    vmo from sin                = 5,
    permission                  = 6;

# SINMAN is called to provide services in relation to each entry, in
  particular to supply the access status of a capability via the 'cap
  inf' entry.  Entry requests as follows: #

INT new seg             = 2,
    seg from sin        = 3, # for retrieving the PDB as a segment #
    cap from sin        = 4,
    preserve            = 5,
    sin inf             = 7,
    cap inf             = 11;

INT sizemask            = ABS 16rffffff; # returned by 'sin inf' #

# Slot allocation within the PDB:

  A bit map is kept to record those slots in the P,I,R capability
  segments for which there is currently an entry in the PDB.  During an
  update session the map is altered during each transaction, and it is
  possible to determine from the map whether at any instant the
  capability segment size settings are consistent with the entries. #

[pcapsegno: rcapsegno] INT capseg excess; #for excess requests #
[pcapsegno: rcapsegno] INT default excess; #default freedom #

default excess [pcapsegno] := 0;
default excess [icapsegno] := 20;
default excess [rcapsegno] := 0;

INT slot capseg, slot offset;
INT mapbase = 8 * pcapsegno,
    maplimit = 8 * rcapsegno + 7;
[mapbase: maplimit] INT slot map; # bit array for slot use #

INT map word, map mask;
PROC set slot = (INT slot) VOID: # standard representation #
    BEGIN
        slot capseg := slot SHR capsegshift;
        slot offset := (slot & byte2) SHR 16;
        map word := (slotoffset SHR 5) + (slotcapseg SHL 3);
        map mask := 1 SHL (slotoffset REM 32)
    END; # set slot #
```

```
PROC mark pdb slot = (INT word) VOID:
    BEGIN
        INT capseg = word SHR capsegshift;
        INT offset = (word SHR 1) & byte0;
        INT loc = (offset SHR 5) + (capseg SHL 3);
        INT mask = 1 SHL (offset REM 32);
        slotmap [loc] !:= mask
    END; # mark PDB slot #

PROC max slot = (INT capseg) INT:
    BEGIN
        INT base = 8 * capseg;
        INT last slot := -1;
        FOR i FROM 0 TO 7
        DO IF INT word = slotmap [base + i];
               word = 0
           THEN FOR j FROM 0 TO 31
                DO IF (word & (1 SHL j)) = 0
                    THEN last slot := 32 * i + j
                    FI
                OD
           FI
        OD;
        last slot
    END; # max slot #


# Procedures for examining a PDB #

INT result1, result2, result3, result4;

PROC inspect pdb = VOID: # set up PDB to be examined #
    BEGIN
        enter (sinman, seg from sin, sin of cap ! read access, ?, ?, ?);
        movecap (new arg, insp slot)
    END;

PROC examine capseg = (INT capseg) INT:
    BEGIN
        reserve for reading (insp slot);
        INT ans =
            IF INT size = readpdb [capseg + isize];
                size = -1
            THEN capseg not set
            ELSE (size & rcwc access) !
                  (((size & offsetmask) SHR 1) + 1)
            FI;
        release reservation;
        movecap (null capability, insp slot);
        ans
    END;

PROC examine pdb entry = VOID:
    BEGIN
        INT word0, word1, word2;
        reserve for reading (insp slot);
        INT slot = (slotcapseg SHL capsegshift) ! (slotoffset SHL 1);
        INT entry = locatepdbslot (slot, readpdb);
        IF entry > readpdb [pdbentries]
        THEN result1 := no further entry
```

```
        ELSE INT n = firstentry + (entry - 1) * pdbentrysize;
            word0 := readpdb [n];
            word1 := readpdb [n + 1];
            word2 := readpdb [n + 2];
            INT offset = (word0 SHR 1) & byte0;
            INT capseg = (word0 & capsegmask);
            result1 := capseg ! (offset SHL 16);
            result2 := (word0 & rhword) SHR typeshift;
            IF result2 = permission
            THEN result3 := word1;
                result4 := word2
            ELIF result2 = workspace segment
            THEN result3 := (word1 SHR 8) & accessmask;
                result4 := (word1 & sizemask) !
                              ((word0 SHL 8) & byte3)
            ELSE result3 := word1 & (rhword ! accessmask);
                IF INT rc =
                    enter (sinman, sininf, word1 & rhword, ?, ?, ?);
                    rc < 0
                THEN result1 := rc + rc incr
                ELSE result4 := rc & byte3
                FI
            FI
        FI;
        release reservation;
        movecap (null capability, insp slot)
    END; # examine PDB entry #


INT max new entries = 32;
MODE NEW = STRUCT (INT w0, w1, w2, SLOT s, BOOL del);
[1: maxnewentries] NEW new;
FOR i FROM LWB new TO UPB new DO s OF new [i] := getslot OD;

INT first new entry, next new entry;
SLOT new entry slot;
INT new vmo type, new segment size, new entry access, new entry type;

PROC set pdb entry = INT:
    BEGIN
        new entry type := third argument;
        new entry access := fourth argument & accessmask;
        CASE new entry type IN # switch on entry type #

            # 1: workspace segment #
            IF new vmo type := fifth argument SHR 24;
                new vmo type <= 3
            THEN IF newvmotype = 0 THEN newvmotype := segtype FI;
                new segment size := fifth argument & size mask;
                make work segment entry
            ELSE invalid segment type
            FI,

            # 2: initialise from file title #
            unimplemented request,
```

```
            # 3: initialise from SIN #
            IF setcapinf (entry cap);
                typeofcap = segtype
                ANDF (access of cap & read access) = 0
            THEN make early bound entry
            ELSE weak supporting capability
            FI,
            # 4: capability from file title #
            unimplemented request,

            # 5: early bound capability #
            IF setcapinf (entry cap)
            THEN IF typeofcap = swctype
                THEN make permission entry
                ELIF (access of cap & new entry access) = new entry access
                THEN make early bound entry
                ELSE weak supporting capability
                FI
            ELSE weak supporting capability
            FI
        OUT bad pdb entry
        ESAC
    END; # set PDB entry #

PROC make delete entry = INT:
    BEGIN
        REF NEW entry = get free entry;
        w0 OF entry := (slotcapseg SHL capsegshift) !
                       (slotoffset SHL 1);
        del OF entry := TRUE;
        slotmap [mapword] &:= mapmask;
        0
    END; # make delete entry #

PROC make work segment entry = INT:
    BEGIN
        REF NEW entry = get free entry;
        w0 OF entry := (slotcapseg SHL capsegshift) ! (slotoffset SHL 1)
            ! (new vmo type SHL 16) ! (new entry type SHL typeshift);
        w1 OF entry := (new entry access SHL 8) ! (new segment size);
        w2 OF entry := 0;
        del OF entry := FALSE;
        slotmap [mapword] !:= mapmask;
        0
    END; # make work segment entry #

PROC make permission entry = INT:
    BEGIN
        REF NEW entry = get free entry;
        w0 OF entry := (slotcapseg SHL capsegshift) ! (slotoffset SHL 1)
                        ! (permission SHL typeshift);
        w1 OF entry := firstword of cap;
        w2 OF entry := secondword of cap;
        del OF entry := FALSE;
        slotmap [mapword] !:= mapmask;
        0
    END; # make permission entry #
```

```
PROC make early bound entry = INT:
    BEGIN
        REF NEW entry = get free entry;
        w0 OF entry := (slotcapseg SHL capsegshift) ! (slotoffset SHL 1)
                       ! (new entry type SHL typeshift) ! (sinbit);
        w1 OF entry := new entry access ! sin of cap;
        w2 OF entry := 0;
        movecap (entry cap, s OF entry);
        del OF entry := FALSE;
        slotmap [mapword] !:= mapmask;
        0
    END; # make early bound entry #

PROC get free entry = REF NEW:
    BEGIN
        IF next new entry >= max new entries
        THEN return (too many requests);
            GOTO restart
        FI;
        new [next new entry +:=:= 1]
    END;

# Routines for checking suitability of capabilities #
INT type of cap,
    access of cap, sin of cap, firstword of cap, secondword of cap;

PROC set cap inf = (SLOT slot) BOOL:
    BEGIN
        INT wordoffset = (slot OF slot SHR 15) & offsetmask;
        INT indinf = cseginf (slot);
        INT status =
            IF indinf > 0
                ANDF wordoffset < (indinf & rhword)
            THEN INT info, z;
                REF INT x = firstwordofcap,
                    y = secondwordofcap;
                movecap (slot, new arg);
                enter2 (sinman, capinf, ?, ?, ?, ?, info, x, y, z);
                info
            ELSE -1
            FI;
        IF status < 0
        THEN type of cap := 0;
            access of cap := 0;
            sin of cap := -1
        ELSE type of cap := status SHR 24;
            access of cap := status & accessmask;
            sin of cap := status & rhword
        FI;
        (status >= 0)
    END; # set cap inf #

PROC check pdb inf = (SLOT slot, INT access) BOOL:
    IF setcapinf (slot)
    THEN (access of cap & access) = access
        ANDF (typeofcap = pdbtype)
    ELSE FALSE
    FI; # check PDB inf #
```

```
# PDB work area descriptors #
INT new string area, size of pdb, current pdb sin := -1;

# Calculation procedures #

PROC capseg size = (INT capseg number) INT:
    IF INT n = pdb [capsegnumber + isize];
        n >= 0
    THEN ((n & offsetmask) SHR 1) + 1
    ELSE -1
    FI; # capseg size #

PROC set capseg = (INT number, INT access) INT:
    IF (number > 0)
        ANDF (number <= 256)
    THEN ((number - 1) * 2) ! (access & rcwc access)
    ELSE -1
    FI; # set capseg #

# Procedures for creating or opening a PDB #

PROC newpdb = (INT size, BOOL empty) INT:
    # allocate and initialise a new PDB #
    # if empty, initialise with no entries, otherwise from A0 #
    # result to A0 #
    IF SLOT pdb = getslot;
        SLOT to seg = getslot;
        REF [] INT to = maparray (to seg);

        PROC allocate = (INT size, acc) INT:
            IF INT rc = enter (sinman, newseg, -1, acc, pdbtype, size);
                rc < 0
            THEN freeslot (pdb);
                freeslot (to seg);
                rc + rc incr
            ELSE # rc = SIN #
                movecap (n0, pdb);
                enter (sinman, seg from sin, rc ! write access, ?, ?, ?);
                movecap (n0, to seg);
                0
            FI;
        empty
    THEN IF size < 7
        THEN freeslot (pdb);
            freeslot (to seg);
            pdb full
        ELIF INT rc = allocate (size, link ! inspect ! modify);
            rc < 0
        THEN rc
        ELSE to [pdb entries] := 0;
            to [free pointer] := first entry;
            to [string area] := size;
            movecap (pdb, arg slot);
            freeslot (to seg);
            freeslot (pdb);
            size
        FI
    ELIF NOT check pdb inf (arg slot, link)
    THEN freeslot (to seg);
        freeslot (pdb);
```

```
        weak supporting capability
    ELIF SLOT from seg = getslot;
        movecap (arg slot, n0);
        enter (sinman, seg from sin, sin of cap ! read access, ?, ?, ?);
        movecap (n0, from seg);
        REF [] INT from = maparray (from seg);
        INT from size = segsize (from seg);
        INT to size =
            IF INT min =
                    from size -
                    (from [string area] - from [free pointer]);
                min <= size
            THEN size
            ELSE min
            FI;
        INT rc = allocate (to size, access of cap);
        rc < 0
    THEN freeslot (from seg);
        rc
    ELSE to [pdb entries] := from [pdb entries];
        to [free pointer] := from [free pointer];
        to [pcapsegsize] := from [pcapsegsize];
        to [icapsegsize] := from [icapsegsize];
        to [rcapsegsize] := from [rcapsegsize];
        to [makepack lock] := -1;
        FOR entry FROM first entry BY pdb entry size
            TO first entry + (from [pdb entries] - 1) * pdb entry size
        DO FOR i FROM entry TO entry + pdb entry size - 1
            DO to [i] := from [i] OD;
            IF (from [entry] & sinbit) = 0
            THEN INT sin = from [entry + 1] & rhword;
            enter (sinman, seg from sin, sin ! read access, ?, ?, ?);
            enter (sinman, preserve, ?, ?, ?, ?)
            FI
        OD;
        INT string size = from size - from [string area];
        FOR j FROM -string size TO -1
        DO to [to size + j] := from [from size + j] OD;
        to [string area] := to size - string size;
        movecap (pdb, arg slot);
        freeslot (pdb);
        freeslot (from seg);
        freeslot (to seg);
        to size
    FI; # new PDB #

PROC reserve pdb = (INT sin) BOOL:
    BEGIN
        enter (sinman, seg from sin, sin ! readwrite access, ?, ?, ?);
        reserve for writing (new arg);
        BOOL pdb free = (testpdb [makepack lock] = 0);
        IF pdb free
        THEN testpdb [makepack lock] := 0;
            movecap (new arg, true slot)
        FI;
        release reservation;
        pdb free
    END; # reserve PDB #
```

```
PROC open for update = INT:
    BEGIN
        INT pdb info = enter (sinman, sin inf, sin of cap, ?, ?, ?);
        size of pdb := pdbinfo & sizemask;
        current pdb sin := sin of cap;
        IF INT sin = enter (sinman, newseg, sin of cap,
                            readwrite access, ?, ?);
            sin < 0
        THEN current pdb sin := -1;
            sin + rc incr
        ELSE movecap (new arg, pdb slot);
            IF pdb [pdb entries] = -1 # new PDB #
            THEN pdb [pdb entries] := 0;
                 pdb [free pointer] := first entry;
                 pdb [string area] := size of pdb
            FI;
            pdb [makepack lock] := -1; # free when copied back #
            FOR i FROM mapbase TO maplimit DO slotmap [i] := 0 OD;
            INT pointer := first entry;
            FOR j FROM 1 TO pdb [pdbentries] # mark PDB entries #
            DO INT word = pdb [pointer];
                markpdbslot (word);
                pointer +:= pdb entry size
            OD;
            FOR capseg FROM pcapsegno TO rcapsegno
            DO capseg excess [capseg] := -1 OD;
            first new entry := 1;
            next new entry := first new entry;
            new string area := pdb [string area];
            0
        FI
    END; # open for update #

# Procedures for closing an open PDB #

INT max sins = 256;
[1: maxsins] INT delete list;
INT deleted sins;
BOOL entry match;

PROC abandon session = VOID:
    BEGIN
        free local slots;
        reserve for writing (true slot);
        truepdb [makepack lock] := -1;
        release reservation;
        movecap (null capability, true slot);
        current pdb sin := -1;
        movecap (null capability, pdb slot)
    END; # abandon session #

PROC free local slots = VOID:
    FOR pointer FROM first new entry TO next new entry - 1
    DO movecap (null capability, s OF new [pointer]) OD;
```

```
PROC close after update = INT:
    BEGIN
        INT rc := 0;
        deleted sins := 0;
        FOR capseg FROM pcapsegno TO rcapsegno
        DO fix capseg size (capseg) OD;
        FOR pointer FROM first new entry TO next new entry - 1
        DO REF NEW block = new [pointer];
            INT slot = w0 OF block & pdbslotmask;
            INT entry = locatepdbslot (slot, pdb);
            IF entry match
            THEN add deletion (entry);
                IF del OF block
                THEN remove (entry)
                ELSE replace (entry, block)
                FI
            ELSE IF NOT del OF block
                THEN INT offset := pdb [free pointer];
                    pdb [free pointer] +:= pdb entry size;
                    pdb [pdb entries] +:= 1;
                    IF pdb [free pointer] > new string area
                    THEN rc := pdb full;
                        GOTO failed # at end of rtn #
                    FI;
                    FOR i FROM entry TO (pdb [pdbentries] - 1)
                    DO INT new offset = offset - pdbentrysize;
                        pdb [offset] := pdb [new offset];
                        pdb [offset + 1] := pdb [newoffset + 1];
                        pdb [offset + 2] := pdb [newoffset + 2];
                        offset := newoffset
                    OD;
                    replace (entry, block)
                FI
            FI
        OD;
        # at this point the PDB must be copied across #
        reserve for writing (true slot);
        move (mapsegment (pdb slot), true pdb);
        release reservation;
        send data message (delete slot, 1, current pdb sin, ?, ?);
        FOR i TO deleted sins
        DO senddatamessage (deleteslot, 2, -1, deletelist [i], ?) OD;
        # finally, overwrite slots to help the PRL garbage collector #
failed: # jump here if PDB becomes full #
        movecap (null capability, true slot);
        movecap (null capability, pdb slot);
        free local slots;
        current pdb sin := -1;
        rc
    END; # close after update #

PROC fix capseg size = (INT capseg) VOID:
    BEGIN
        INT min size = maxslot (capseg) + 1;
        INT old size = capsegsize (capseg);
        INT new size := IF oldsize = -1 THEN 0 ELSE old size FI;
        IF INT excess = capsegexcess [capseg];
            excess = -1
        THEN IF newsize < minsize
            THEN newsize := minsize + defaultexcess [capseg]
```

```
            FI
        ELSE newsize := minsize + excess
        FI;
        IF newsize > 256 THEN newsize := 256 FI;
        pdb [capseg + isize] :=
            setcapseg (newsize, pdb [capseg + isize] & rowcaccess)
    END; # fix capseg size #

PROC locate pdb slot = (INT slot, REF [] INT refpdb) INT:
    BEGIN
        INT entry := 1;
        INT pointer := first entry;
        entry match := FALSE;
        INT n;
        FOR i FROM 1 TO refpdb [pdbentries]
            WHILE (n := refpdb [pointer] & pdbslotmask; n <= slot)
        DO IF n = slot THEN entry match := TRUE ELSE entry +:= 1 FI;
            pointer +:= pdb entry size
        OD;
        entry
    END; # locate PDB slot #

PROC add deletion = (INT entry) VOID:
    IF INT offset = first entry + (entry - 1) * pdbentrysize;
        (pdb [offset] & sinbit) = 0
    THEN INT sin = pdb [offset + 1] & rhword;
        deleted sins +:= 1;
        deletelist [deleted sins] := sin
    FI; # add deletion #

PROC remove = (INT entry) VOID:
    BEGIN
        INT offset := first entry + (entry - 1) * pdbentrysize;
        FOR i FROM entry TO (pdb [pdbentries] - 1)
        DO INT new offset = offset + pdbentrysize;
            pdb [offset] := pdb [newoffset];
            pdb [offset + 1] := pdb [newoffset + 1];
            pdb [offset + 2] := pdb [newoffset + 2];
            offset := newoffset
        OD;
        pdb [offset] := -1;
        pdb [offset + 1] := -1;
        pdb [offset + 2] := -1; # clear #
        pdb [pdbentries] -:= 1;
        pdb [freepointer] -:= pdb entry size
    END; # remove #

PROC replace = (INT entry, REF NEW block) VOID:
    BEGIN
        INT offset = first entry + (entry - 1) * pdbentrysize;
        IF (w0 OF block & sinbit) = 0
        THEN SLOT slot = s OF block;
            movecap (slot, new arg);
            enter (sinman, preserve, ?, ?, ?, ?)
        FI;
        pdb [offset: AT 1] := (w0 OF block, w1 OF block, w2 OF block)
    END; # replace #
```

```
# MAIN LOOP #

restart:

DO CASE first argument IN # switch on entry type #

        # 1: examine capability segment size #
        IF checkpdbinf (argslot, inspect)
        THEN IF (second argument >= 4)
                ANDF (second argument <= 6)
             THEN inspect pdb;
                return (examine capseg (second argument))
             ELSE return (bad capseg number)
             FI
        ELSE return (inadequate status)
        FI,

        # 2: examine PDB entry #
        BEGIN
             result2 := -1;
             result3 := -1;
             result4 := -1;
             IF checkpdbinf (argslot, inspect)
             THEN IF (setslot (second argument);
                          slotcapseg < 4 OR slotcapseg > 6)
                  THEN result1 := bad capseg number
                  ELSE inspect pdb; examine pdb entry
                  FI
             ELSE result1 := inadequate status
             FI;
             return2 (result1, result2, result3, result4)
        END,

        # 3: set capability segment size #
        IF checkpdbinf (argslot, modify)
             ANDF sin of cap = current pdb sin
        THEN IF (second argument < 4)
                ORF (second argument > 6)
             THEN return (bad capseg number)
             ELIF third argument <= 256 # maximum capseg size #
             THEN INT loc = second argument + isize;
             pdb [loc] := setcapseg (third argument, fourth argument);
             capseg excess [second argument] := -1;
             return (0)
             ELSE return (bad capseg size)
             FI
        ELSE return (inadequate status)
        FI,

        # 4: set capability segment excess #
        IF checkpdbinf (argslot, modify)
             ANDF sin of cap = current pdb sin
        THEN IF (second argument < 4)
                ORF (second argument > 6)
             THEN return (bad capseg number)
             ELIF INT excess = third argument;
                excess <= 256
             THEN INT size = IF excess < 0 THEN -1 ELSE excess FI;
                capseg excess [second argument] := size;
                return (0)
```

```
    ELSE return (bad capseg size)
    FI
ELSE return (inadequate status)
FI,

# 5: set PDB entry #
IF checkpdbinf (argslot, modify)
    ANDF sin of cap = current pdb sin
THEN IF (setslot (second argument);
         slotcapseg < 4 OR slotcapseg > 6)
    THEN return (bad capseg number)
    ELSE return (set pdb entry)
    FI
ELSE return (inadequate status)
FI,

# 6: delete PDB entry #
IF checkpdbinf (argslot, modify)
    ANDF sin of cap = current pdb sin
THEN IF (setslot (second argument);
         slotcapseg < 4 OR slotcapseg > 6)
    THEN return (bad capseg number)
    ELSE return (make delete entry)
    FI
ELSE return (inadequate status)
FI,

# 7: open PDB for update #
IF current pdb sin = -1
THEN return (update in progress)
ELIF  checkpdbinf (argslot, modify)
THEN return (inadequate status)
ELIF reserve pdb (sin of cap)
THEN return (open for update)
ELSE return (makepack competition)
FI,

# 8: close PDB after update #
IF current pdb sin = -1
THEN return (no update session)
ELIF checkpdbinf (argslot, modify)
    ANDF sin of cap = current pdb sin
THEN return (close after update)
ELSE return (inadequate status)
FI,
```

```
        # 9: abandon update session #
        BEGIN
            IF current pdb sin = -1 THEN abandon session FI;
            return (0)
        END,

        # 10: create new PDB #
        return (newpdb (second argument, ODD third argument))

    OUT return (unknown request)

    ESAC

OD # end of main loop #
```

DISCGARB

This is the asynchronous disc garbage collector. It is run as a process on its own. It asks SINMAN to deliver a snapshot of the SIN directory, indicating the existence and type of all objects which contain SINs. It also receives messages from instances of SINMAN in other processes whenever directories or PDBs are retrieved from the filing system. On the basis of this information, together with knowledge of the SIN of the master directory, it can locate detached directories and dispose of them. Although a directory or PDB whose reference count falls to zero could be disposed of directly, the work is left to DISCGARB.

```
SLOT known slot                 = P 4, # bit map 'a' #
     untouched slot             = P 5, # bit map 'b' #
     needs look slot            = P 6; # bit map 'c' #
SLOT sinman                     = P 7,
     mn clock                   = P 8,
     mn receive                 = P 9,
     mn logger                  = P 10,
     stop perm                  = P 11;

SLOT structure slot             = getslot;

# Error codes #
INT next obj fail               = ABS 16r801c0001,
    check sin fail              = ABS 16r801c0002,
    scan obj fail               = ABS 16r801c0003,
    unexpected failure          = ABS 16r801c0004,
    use count fail              = ABS 16r801c0005;

# SINMAN types #
INT sdmtype                     = 0,
    segtype                     = 1,
    dirtype                     = 2,
    pdbtype                     = 3,
    swctype                     = 4;

# SINMAN entry requests #
INT remove                      = 6,
    sininf                      = 7,
    use count                   = 10,
    ask sinman                  = 14, # to get bit maps #
    stop discgarb               = 15,
    destroy                     = 16,
    change cap                  = 17,
    make cap                    = 18,
    kill cap                    = 19;


# Segment reservation #

BOOL reserved := FALSE;

PROC reserve = (SLOT s) VOID: (reserved := TRUE;
                               reserve for reading (s));
```

```
PROC release = VOID:
       IF reserved THEN release reservation; reserved := FALSE FI;

PROC stop sinman = VOID: enter (sinman, stop discgarb, 0, 0, 0, 0);

runtime error := (STRING s, INT i) VOID: stop system (stop perm, i);


# Bit maps #

REF [] INT known                = maparray (known slot),
    untouched                   = maparray (untouched slot),
    needs look                  = maparray (needs look slot);

INT map size = 8000 % bitswidth;
INT needs look count := 0;
INT obj count := 0;

PROC new obj = (INT sin) VOID:
    # mark object as 'needs look' if it has not been encountered
      before, that is if ( a = 1 AND b = 1 ) OR a = 0.  Note that
      (a = 0)  =>  (b = 0) #
    IF INT word = sin % bitswidth,
            bit = 1 SHL (sin %* bitswidth);
        INT a = (known [word] !:=:= bit) & bit,
            b = (untouched [word] &:=:= NOT bit) & bit;
        a = b
    THEN needs look [word] !:= bit;
        obj count +:= 1;
        needs look count +:= 1
    FI;

INT find ptr := 0; # used circularly; never reset #

PROC next obj = INT:
    BEGIN
        TO mapsize WHILE needs look [find ptr] = 0
        DO find ptr +:= 1; IF find ptr >= mapsize THEN find ptr := 0 FI OD;
        REF INT word = needs look [find ptr];
        IF word = 0 THEN runtime error ("", next obj fail) FI;
        INT bit := 1;
        INT bit count := 0;
        WHILE (word & bit) = 0 DO bit := bit SHL 1; bit count +:= 1 OD;
        word &:= NOT bit;
        needs look count -:= 1;
        find ptr * bitswidth + bit count
    END;

PROC check sin = (INT sin) VOID:
    # called for each SIN in a directory or PDB #
    IF INT type = enter (sinman, sininf, sin, ?, ?, ?);
        type < 0
    THEN runtime error ("", check sin fail)
    ELIF (type SHR 24) = dirtype
        ORF (type SHR 24) = pdbtype
    THEN new obj (sin)
    FI;

INT serial; # set by 'get bit maps'; primary copy is in SINMAN #
```

```
PROC get bit maps = VOID:
    # initialise bit maps from SIN directory #
    BEGIN
        INT c count, mfdsin, d;
        moverow (untouched, n0);
        moverow (needs look, n1);
        enter2 (sinman, ask sinman, 0, 0, 0, 0, c count, mfdsin,
                serial, d);
        FOR i FROM 0 TO mapsize - 1
        DO known [i] := untouched [i] ! needs look [i] OD;
        needs look count := obj count := c count;
        new obj (mfdsin)
    END;

# Channels, etc #

SLOT send clock, receive clock, receive, send logger, receive logger;

set up send with reply (mn clock, send clock, data reply message,
    receive clock, data message);

set up receive (mn receive, receive, data message);

set up send with reply (mn logger, send logger, full reply message,
    receive logger, data message);

INT clock interval = 12000 # 2 minute # ,
    objs per tick = 10; # cycle about once per hour #

# Interpretation of directories and PDBs #

PROC init structure slot = VOID:
    (enter (sinman, make cap, ?, ?, ?, ?); movecap (n0, structure slot));

PROC kill structure slot = VOID:
    (movecap (structure slot, n0); enter (sinman, kill cap, ?, ?, ?, ?));

PROC scan object = (INT sin, PROC (INT) VOID check) VOID:
    IF REF [] INT structure = maparray (structure slot);
        movecap (structure slot, n0);
        INT rc = enter (sinman, change cap, sin ! read access, ?, ?, ?);
        rc < 0
    THEN runtime error ("", scan obj fail)
    ELIF movecap (n0, structure slot);
        INT type = rc;
        reserve (structure slot);
        type = dirtype
    THEN # scan directory #

        # See DIRSTRUCT for definition and explanation of the structure
          of a directory #

        OP NEXT = (INT i) INT: structure [i] & rhword;
        OP LAST = (INT i) INT: structure [i] SHR 16;

        INT header size       = 6,
            block size        = 6;
```

```
    INT blocks total       = 0, # indices to directory header #
        blocks free        = 1,
        entries total      = 2,
        freelist head      = 3,
        hash table start   = 4,
        hash table end     = 5,

        null               = 0, # the null index to a directory #

        access matrix offset  = 1, # useful block offsets #
        cap info 1 offset  = 3,
        max offset = block size - 1;

    INT file type field = ABS 16rf0000000;

    PROC check entries = (INT head) VOID:
        BEGIN
            INT index := NEXT head;
            UNTIL index = null
            DO INT nblocks = structure [index + 1];
                TO nblocks - 1 DO index := NEXT index OD;
                INT cap info 1 =
                    structure [index + cap info 1 offset];
                IF (cap info 1 & file type field) = 0 # store #
                THEN INT sin = cap info 1 & rhword;
                    check (sin)
                FI;
                index := NEXT index
            OD
        END;

    IF structure [0] = -1 # if initialised #
    THEN FOR i FROM hash table start TO hash table end
        DO check entries (i) OD
         # otherwise ignore - probably too small to be a directory
           segment #
    FI;

    release

ELIF type = pdbtype
THEN # scan PDB #
    # layout of PDBs; for authoritative version, see LINKER #
    # there should really be an ENVIRON for this, as for DIRSTRUCT #
    INT pdbentries         = 0,
        free pointer       = 1,
        string area        = 2,
        p capseg size      = 3,
        i capseg size      = 4,
        r capseg size      = 5,
        makepack lock      = 6,
        first entry        = 7;
    INT isize = p capseg size - 4;
    INT w per entry = 3; # number of words per entry #
    INT sinbit = 1;

    INT entries = structure [pdb entries];
    FOR i FROM first entry BY w per entry
        TO first entry + (entries - 1) * w per entry
```

```
        DO INT a = structure [i],
                b = structure [i + 1];
            IF (a & sinbit) = 0 THEN check (b & 65535) FI
        OD;

        release
    ELSE runtime error ("", unexpected failure)
    FI;


# Garbage disposal #


PROC scan garbage = (PROC (INT) VOID work) VOID:
    FOR i FROM 0 TO mapsize - 1
    DO INT word = untouched [i];
        FOR j FROM 0 TO bitswidth - 1
        DO IF (word & (1 SHL j)) = 0
           THEN work (i * bitswidth + j) FI OD
    OD;

INT garbage count;

COMMENT
PROC type = (INT sin)VOID:
            ( garbage count +:= 1; print(sin, blank) );
COMMENT

PROC remove dir pdb = (INT sin) VOID:
    # first pass of 'scan garbage' #
    BEGIN
        garbage count +:= 1;

        PROC check entry = (INT s) VOID:
            # argument of 'scan object' #
            IF (known [s % bitswidth] & (1 SHL (s %* bitswidth))) = 0
            THEF # directory or PDB #
                INT rc = enter (sinman, remove, s, ?, ?, ?);
                rc < 0
            THEN runtime error ("remove dir pdb", rc)
            FI;

        scan object (sin, check entry)
    END;

PROC check use count = (INT sin) VOID:
    # second pass of 'scan garbage' #
    IF INT rc = enter (sinman, use count, sin, ?, ?, ?);
        rc < 0
    THEN runtime error ("check use count", rc)
    ELIF rc = 0
    THEN runtime error ("", use count fail)
    FI;

PROC remove seg = (INT sin) VOID:
    # third pass of 'scan garbage' #
    BEGIN

        PROC check entry = (INT s) VOID:
            IF (known [s % bitswidth] & (1 SHL (s %* bitswidth))) = 0
```

```
            THEF # segment #
                INT rc = enter (sinman, remove, s, ?, ?, ?);
                rc < 0
            THEN runtime error ("remove seg", rc)
            FI;

        scan object (sin, check entry)
    END;

PROC destroy obj = (INT sin) VOID:
    # fourth pass of 'scan garbage' #
    enter (sinman, destroy, sin, ?, ?, ?);


# MAIN PROGRAM #


init structure slot;

DO get bit maps;

    WHILE needs look count > 0 ORF messages (receive) > 0
    DO send data message (send clock, clock interval, ?, ?, ?);
        UNTIL (WHILE messages (receive) > 0
                DO INT sin, ser, d;
                    receive data message (receive, sin, ser, d, d);
                    # if ser = serial the message refers to a previous
                      run of DISCGARB, and must be ignored #
                    IF ser = serial THEN new obj (sin) FI
                OD;
                messages (receive clock) > 0)
        DO wait event OD;
        (INT d; receive data message (receive clock, d, d, d, d));

        TO objs per tick WHILE needs look count > 0
        DO INT sin = next obj; scan object (sin, check sin) OD
    OD;

    # the garbage has been found #

    BEGIN

        garbage count := 0;
        scan garbage (remove dir pdb);
        send full message (send logger, 7, obj count, garbage count,
                           ?, P 0);
        kill structure slot;
        scan garbage (check use count);
        scan garbage (remove seg);
        kill structure slot;
        scan garbage (destroy obj);
        UNTIL messages (receive logger) > 0 DO wait event OD;
        (INT d; receive data message (receive logger, d, d, d, d))

    END

OD
```

REFERENCES

Bourne, S. R., Birrell, A. D., and Walker, I. (1976) "The ALGOL68C reference manual". Cambridge University Computing Service

Cook, D. J. (1978) "The evaluation of a protection system". Ph.D. thesis, University of Cambridge

Dahl, O. J., Myhrhaug, B. and Nygaard, K. (1970) "Common base language". Publication No. S-22, Norwegian Computing Center, Oslo

England, D. M. (1974) "Capability concept mechanisms and structure in System 250". Protection in Operating Systems, IRIA, Rocquencourt, France, p.63

Lauer, H. C. and Needham, R. M. (1978) "On the duality of operating systems structures". (Presented at 2nd international colloquium, IRIA, Rocquencourt, France, October 1978.) Operating Systems, ed. D. Lanciaux, North-Holland Pub. Co., Amsterdam (1979). Reprinted in Operating Systems Rev. Vol.13, Pt.2, p.3 (1979)

Saltzer, J. H. (1966) "Traffic control in a multiplexed computer system". Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts

Saltzer, J. H. and Schroeder, M. D. (1975) "The protection of information in computer systems". Proc. IEEE Vol.63, p.1278

Wilkes, M. V. (1975) "Time-sharing computer systems". 3rd edition, Macdonald and Jane's, London; American Elsevier, New York

Wulf, W. et al. (1974) "HYDRA: The kernel of a multiprocessor operating system". Commun. ACM Vol.17, p.337

BIBLIOGRAPHY

Birrell, A. D. (1977) "System programming in a high level language". Ph.D. thesis, University of Cambridge

Birrell, A. D. and Needham, R. M. (1978) "An asynchronous garbage collector for the CAP filing system". Operating Systems Rev. Vol.12, Pt.2, p.31

Birrell, A. D. and Needham, R. M. (1978) "Character streams". Operating Systems Rev. Vol.12, Pt.3, p.29

Birrell, A. D. and Needham, R. M. (1979) "A universal file server". IEEE Trans. Software Engineering (to be published)

Cook, D. J. (1978) "The cost of using the CAP computer's protection facilities". Operating Systems Rev. Vol.12, Pt.2, p.26

Cook, D. J. (1978) "Measuring memory protection on the CAP computer". (Presented at 2nd international colloquium, IRIA, Rocquencourt, France, October 1978.) Operating Systems, ed. D. Lanciaux, North-Holland Pub. Co., Amsterdam, (1979)

Cook, D. J. (1978) "The evaluation of a protection system". Ph.D. thesis, University of Cambridge

Fenton, J. S. (1973) "Information protection systems". Ph.D. thesis, University of Cambridge

Fenton, J. S. (1974) "Memoryless subsystems". Computer J. Vol.17, p.143

Herbert, A. J. (1978) "A new protection architecture for the Cambridge capability computer". Operating Systems Rev. Vol.12, Pt.1, p.24

Herbert, A. J. (1978) "A microprogrammed operating system kernel". Ph.D. thesis, University of Cambridge

Herbert, A. J. (1979) "A hardware-supported protection architecture". (Presented at 2nd international colloquium, IRIA, Rocquencourt, France, October 1978.) Operating Systems, ed. D. Lanciaux, North-Holland Pub. Co., Amsterdam (1979)

Horton, J. R. (1975) "Addressing and protection". Ph.D. thesis, University of Cambridge

Kirkham, C. C. (1975) "Protection and process structure". Ph.D. thesis, University of Cambridge

Needham, R. M. (1972) "Protection systems and protection implementations". Fall Joint Computer Conference, AFIPS Conference Proc. Vol.41, Pt.I, p.571

Needham, R. M. (1974) "Protection - a current research area in operating systems". International Computing Symposium 1973, North-Holland Pub. Co., Amsterdam, p.123

Needham, R. M. (1977) "The CAP project - an interim evaluation". Operating Systems Rev. Vol.11, Pt.5, p.17

Needham, R. M. (1978) "Protection". (Presented at the August 1978 advanced course University of Newcastle upon Tyne.) Distributed Computing Systems, Computing Laboratory, University of Newcastle upon Tyne (1979)

Needham, R. M. and Birrell, A. D. (1977) "The CAP filing system". Operating Systems Rev. Vol.11, Pt.5, p.11

Needham, R. M. and Walker, R. D. H. (1974) "Protection and process management in the CAP computer". Protection in Operating Systems, IRIA, Rocquencourt, France, p. 155

Needham, R. M. and Walker, R. D. H. (1977) "The Cambridge CAP computer and its protection system". Operating Systems Rev. Vol.11, Pt.5, p.1

Needham, R. M. and Wilkes, M. V. (1974) "Domains of protection and the management of processes". Computer J. Vol.17, p.117

Slinn, C. J. (1977) "Aspects of a capability based operating system". Ph.D. thesis, University of Cambridge

Smith, R. F. (1979) "Protection in data bases through the use of capabilities". Ph.D. thesis, University of Cambridge

Stroustrup, B. (1978) "On unifying module interfaces". Operating Systems Rev. Vol.12, Pt.1, p.90

Stroustrup, B. (1979) "Communication and control in distributed computer systems". Ph.D. thesis, University of Cambridge

Taylor, R. J. B. (1978) "Process coordination and resource management". Ph.D. thesis, University of Cambridge

Walker, R. D. H. (1973) "The structure of a well-protected computer". Ph.D. thesis, University of Cambridge

Watson, D. J. (1978) "An approach to protection through capabilities". Ph.D. thesis, University of Cambridge

In addition to the authors listed above, the following people have made notable contributions to the project:

R. Fairbairns

J. L. Gluza

M. A. Johnson

D. W. Payne

INDEX

Underlined page numbers refer to the appendices

abstract data types 6
access code 3
access control 49-53
access control matrix 50
access validation 24-26
access vector 50
address argument validation 66-67
ALGOL68C 32-33, 92-94
ALTERCAP 83
ALTERDATA 83
ancillary field 23-24
argument decoding 43-44
attentions 36-38
base 3
base-limit register 1
BCPL 32
Birrell, A. D. 90
Bourne, S. R. 92
C-stack 11, 32
C-type access 3
CAL-TSS 75
CAP-3 77
capabilities,
  absolute 9
  evaluation of 10
  format of 16-17
  global 61, 77
  outform 38
  peripheral 17
  permission 34
  preservation of 45-48
  relative 9-10
  restriction of use of 71-73
  retrieval of 45-49
  revocation of 72-73, 85-86
  software 17-18
capability, type of 3
capability
  loading cycle 14-16
  management 34
  register 3
  segment 10
  specifier 13
  store 14
  unit 19, 22-24
central register unit 19-21
command program 42-44
command status 43
communication, interprocess 34-35, 87
Cook, D. J. 70
coordinator 9, 13, 33-34
coroutine 94
count field 23-24
CTSS 1
D-type access 3
data structures, multisegment 59-60, 67
DESPOOL 43
direct mode 28
directory manager 46-48
directory segment 40-41, 45
DIRMAN 46-48, 95, 126
DIRSTRUCT 95, 124
disc allocation map 54
disc management 53-56
Discgarb 59, 95, 153
domain descriptor 78
domain of coordination 63-64
domain of protection 7
ECPROC 33-34
Ensurer 54

enter capabilities, passing of 63-66
enter capability 7
ENTER instruction 7, 11, 29, 30, 32
ENTER COORDINATOR instruction 13, 29, 30
ENTER SUBPROCESS instruction 13, 29, 30
errors 36-38
evaluation of capabilities 10
Fabry, R. S. 7
faults 36-38, 97
floating point unit 19
FLUSH instruction 16, 29
garbage collection 42, 55-56, 60, 61, 153
generation of procedure 45
global capabilities 61, 77
Herbert, A. J. 61, 75
hierarchies
  of processes 13, 62
  of protection 5
Honeywell 6180 66
HYDRA 5, 75
ICL 2900 66
indirection table 14
indirectory 14
Initiator 42
input-output 19
integrity of system 53-55
interlocks 35-36
interprocess communication 34-35, 87
Interrupt process 89
interrupts 36, 89
kernel 75, 90
KILLBLOK 89
last mode 28
Lauer, H. C. 68
LIBRARY 52
limit 3
LINKER 48
local naming 59-61
LOGIN 43
LOGOUT 44
MAKEBLOK 87
MAKEENTER 71
MAKEIND instruction 11, 29
MAKEPACK 45, 95, 138
MAKEPDB 69, 138
map,
  disc allocation 54
  central, in CAP-3 77
master coordinator 13
master file directory 49
master resource list 10
MC 92
memory access validation 24-29
message channel 35, 87
message pool 87
microprogram 19, 29-31
microprogram unit 19, 21-22
MIN 92
minimum privilege, principle of 68-70
modularity 67-68
Module 53
MOVE instruction 29
MOVECAP instruction 16, 29, 30, 92
MOVECAPA instruction 92
MRL 10
MULTICS 33
multisegment data structures 59-60, 67
Needham, R. M. 68, 90
nomenclature, note on 14
normal mode 26
outform capability 38
P-store 80
PARMS 44
peripheral computer 19

peripherals 19
permission matrix 50
permission vector 50
Plessey System 250 5, 67
preservation of capabilities 45-48
primitives 34
privilege, separation of 69
privileged mode 1
PRL 10
PRL entry 14
PRLGARB 42
procedure description block 40-41, 48-50, 138
procedure generation 45
process base 9
process management 33-34
process resource list 10
protected procedure 8, 32-33
  instance of 8, 45
protection environment 9
protection,
  domain of 7
  loss of 64-66
  rings of 5
real store manager 38-39
RECEIVE 87
Redell, D. D. 85
REFINE instruction 16, 29
REPLY 87
Restart 54-55
retrieval of capabilities 45-49
RETURN instruction 8, 11, 29, 30
revocation 72-73, 85
revocation of capabilities 72-73, 85-86
REVOKE 85
ruggedness 6-7
RUN 43
run time system 92
Saltzer, J. H. 30
Schroeder, M. D. 69
SEALCAP 83
SEALDATA 83
segment 2, 3, 38-40
segment descriptor 3
segment reservation 35
segment swapping 57-59
SEND 87
separation of privilege 69
SER 93
SETUP 35
SIMULA 6
SIN directory 41, 98
SINMAN 40-41, 95, 98
slave store 19
stage 1 29
STARTOP 37, 43, 44
STOREMAN 58, 95, 119
subprocess 9, 13
subprocess, control of 61-63
system internal name 40-42
tag field 23-24, 26-27
tagged capability architecture 73-74
TGM store 26
traps 23, 36-37
UNSEALCAP 83
UNSEALDATA 83
USE 93
user file directory 45, 49
user interface 42-44
V-store 21-22
virtual memory 38-40
virtual store manager 39-40, 95, 113
WAIT 89
WAKEUP 89
wake-up waiting switch 30
Wilkes, M. V. 1, 5, 7
windows 39
write-around 92