Technical Report

UCAM-CL-TR-9
ISSN 1476-2986

Number 9

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Evaluation of a protection system

Douglas John Cook

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 1978 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Gonville & Caius College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Contents

Summary

The CAP research project was set up in 1969 to investigate memory protection by designing and building a computer with hardware support for a very detailed protection system based on the use of capabilities. The computer has been built and an operating system written which exploits its protection facilities. It is time, therefore, to assess how successful the project has been. A necessary component of such an assessment is an evaluation of the CAP's protection system and this thesis presents the results of the author's research in this area.

Protection in computer systems is first introduced with a brief description of various models of protection systems and mechanisms for the provision of protection. There follows a description in some detail of the CAP computer and the CAP Operating System with particular attention paid to those aspects of the design which are relevant to the research reported. A brief introduction to performance evaluation techniques is given followed by a discussion of performance evaluation on the CAP computer.

The need for measuring the benefits and costs of protection is discussed and there is a detailed critical description of previous research in this area. A simple model of a protection system is presented as is a protection measure based on this model. There is then a discussion of how the services provided by modules in the system fit into the model and the protection measure. The application of the protection measure to the CAP Operating System is described. The results led to suggestions for the improvement of

the protection aspects of the operating system and these are discussed in detail. The implications of the results for operating system design in general are also discussed.

The experiments to investigate the cost of using the protection provided on the CAP are described next. Some performance evaluation work was done in connection with the protection cost experiments and this too is described.

Chapter 1

1. Introduction

The CAP research project was started in 1969 to investigate protection in computer systems by designing and building a complete system with hardware support for a capability-based protection mechanism. An introduction to the use of capabilities for protection is given in Wilkes's book [Wilkes 1975] and the philosophy behind the design adopted is described in detail in Walker's thesis [Walker 1973] and in various papers by Needham and others [Needham 1972,1973,1974(a),(b)]. The hardware and microprogram are described in the CAP Hardware Manual and the CAP System Programmer's Manual [Herbert 1978(a),(b)]. By late 1976 an operating system for the CAP was sufficiently complete to support ordinary users. It is described in the CAP Operating System Manual [Herbert 1978(c)] and the philosophy behind some of the design decisions is discussed in Slinn's thesis [Slinn 1977]. Three papers presented at the ACM 6th Symposium on Operating Systems Principles give an overview of the current state of the project and discuss the CAP's protection mechanisms and the filing system used in the CAP Operating System [Needham 1977(a),(b),(c)].

The CAP project has reached the stage when it is time to try to assess how successful the project has been. A necessary component of such an assessment is work directed at evaluating the performance of the CAP's hardware, microprogram and operating system. For a project which is concerned with research into protection a

particularly important aspect of assessing its success is the measurement of protection. This thesis presents a technique for measuring protection in computer systems. The technique was applied to the CAP Operating System and the results showed how the operating system could be brought closer to a state of minimum privilege. The author has submitted two papers, one of which has already been accepted for publication, reporting this work on protection measurement [Cook 1978(a),(c)].

There is no shortage of publications dealing with protection: for instance, a survey paper by Saltzer [1975] lists 100 references. However, there is an almost complete absence of papers dealing with the quantitative aspects of protection. Some research projects are reportedly looking at the costs [Saltzer 1974(b)] but when the benefits and/or costs of protection are mentioned at all in the literature it is almost always in qualitative terms such as "each (security policy) incurs a cost for protection which is roughly linear with the degree of security achieved" [Jones 1975] or "this cost (of switching protection domains) in Hydra is considerable" [Cohen 1975]. Some rough cost indications are given in Weissman [1969] and Chastain [1973]. An extensive search of the literature revealed only one paper [Ellis 1974] which addressed the quantitative aspects of the benefits of protection and one thesis [Wyeth 1976] reporting research into a methodology for the quantitative comparison of protection systems. Wyeth's work draws on the accuracy and suitability measures of Jones [1973] which the latter used to derive a qualitatively based partial ordering of

a number of protection mechanisms. Ellis claims to be "the first to present a mathematically rigorous definition (with proofs) of the degree of protection of a system". The author has not found any evidence to refute his claim. Ellis's and Wyeth's work have not exhausted this area of research and the shortcomings of their respective approaches are discussed later in this thesis (Chapter 5).

Also reported in the thesis are a number of performance evaluation experiments carried out in order to assess the cost of using the CAP's protection system. The author made use of the CAP's programmable microprogram as one of the monitoring tools for these experiments. Although there have been very few published cases of the use of microprogramming as a monitoring tool [Saal 1972,Denny 1975,1977], the experiments reported in this thesis demonstrated that it can be used to good effect to complement the use of hardware and software monitors. A paper reporting the results of the author's performance evaluation work has been accepted for publication [Cook 1978(b)].

1.1 Outline of thesis

Protection in computer systems is introduced in Chapter 2 with a brief description of various models of protection systems and mechanisms for the provision of protection. Several capability-based protection systems are mentioned.

Chapter 3 describes the CAP computer and the CAP Operating System in some detail paying particular attention to the aspects of the design which are relevant to the research reported in this

thesis.

A brief introduction to performance evaluation techniques is given in Chapter 4 which concludes with a section devoted to performance evaluation on the CAP computer.

In Chapter 5 the need for a measure of the benefits and costs of protection is discussed and there is a detailed critical description of previous research in this area [Jones 1973, Ellis 1974, Wyeth 1976]. A simple model of a protection system is presented as is a protection measure based on this model. Finally, there is a discussion of how the services provided by the modules in a system fit into the model and the protection measure.

Chapter 6 describes the application of the protection measure to the CAP Operating System and discusses the implications of the results for operating system design.

The experiments to investigate the cost of using the protection provided on the CAP are described in Chapter 7. Some performance evaluation work was done in connection with the protection cost experiments and this too is described.

Chapter 8 contains some concluding remarks and suggestions for further research.

It is inevitable that many technical terms, some specific to the CAP project, are used in this thesis and a Glossary is provided to help the reader. The Glossary includes a brief description of the various modules of the CAP Operating System. It is followed by a list of references. Finally, there are appendices giving the details of the experiments and their results.

Chapter 2

2. Protection and capability-based systems

2.1 Protection

Computer *security* is concerned with the protection of data against accidental or intentional disclosure, destruction or modification [Browne 1976]. It includes *protection* which for the purpose of this thesis will be taken to refer to the logical and physical mechanisms that control access to data inside a computer system. This does not imply that other aspects of computer security (e.g. guarding against wire tapping) are not important, only that they are outside the scope of this thesis, as too are considerations of control protection as distinct from access protection [Cohen 1975] except insofar as control protection is provided by way of access protection. The purpose of protection is to ensure that, at any point in the execution of a job by means of a computer, only those objects are accessible that are required to be so and only in the access mode necessary for performance of the task in hand [Needham 1972]. This is the so-called *principle of minimum* (or *least*) *privilege** adherence to which has been a basic design aim in the development of the CAP Operating System [Needham 1977(a)].

---

* The principle of minimum privilege, that every program and every user of the system should operate with the least set of privileges necessary to complete the job, is stated in Saltzer [1974(a)]. Needham [1972] sees minimisation of privilege as the purpose of protection but does not state the principle formally. Jones [1973] refers to the same idea as the need-to-know principle.

The first abstract model of a protection system was developed by Lampson [1971] and was extended by Graham and Denning [Graham 1972]. The essential features of Lampson's access matrix model are (a) a set of objects, the entities to which access must be controlled, each having an associated unique identification number; (b) a set of domains, the entities which permit access to objects; (c) an access matrix which governs the accessing of objects by domains; and (d) a set of rules governing the manipulation of the access matrix. Graham and Denning allow more than one process to run in a domain and replace Lampson's set of domains by a set of subjects, the entities which request access to objects. A subject is regarded as a (process,domain) pair.

The model developed by Jones [1973] is expressed in terms of the environment of a process, the environment being composed of a set of rights, each specifying an object and an access mode applicable to that object, and three rules, the Enforcement Rule, the Right Transfer Rule and the Environment Binding Rule. The purpose of these rules is to ensure that a process is made to operate in an environment that specifies the subset of objects in the system that the process can reference as well as the variety of ways in which the process can access each object in the subset. The objective of Jones's model is the attainment of the principle of minimum privilege.

Graham and Denning describe three practical implementations for storing the access matrix. The first uses the idea of storing the matrix by rows, the capability approach to protection [Dennis

1966]. The storing of the access matrix by columns, the second approach they describe, corresponds to the access control list approach of Multics [Glaser 1967,Saltzer 1974(a)]. The third approach represents a compromise between the capability and access control list implementations. This is the lock and key system [Needham 1972].

2.2 Capability-based systems

There have been a number of attempts to implement capability-based protection systems one of which, the Plessey System 250 [England 1972,1974], is commercially available. Most memory protection systems in current use are based on the idea of two modes of operation. The protection system operates in one of the modes, the unprivileged mode, but not in the other, the privileged mode. The operating system normally runs in privileged mode and thus has the rights to access all the computer's memory and to execute any of the machine's instructions including the privileged instructions whose use is denied to programs running in unprivileged mode. The two-state machine has been extended to three states in the PDP 11/45 [Digital 1971] and generalised in Multics which uses a protection system based on rings of decreasing privilege [Schroeder 1972(a)]. Even Multics, though an improvement on earlier systems, leaves a good deal to be desired because of the hierarchical structuring of the available protection environments: a process running in a given ring has access to all material in lower numbered rings but no access to any material in higher

rings. Typical of the shortcomings of an hierarchical system is its inability to enable an input routine and an output routine to have access to their own buffers without either the former routine also having access to the output buffer or vice versa [Needham 1972].

A consequence of the inadequacies of systems based on an hierarchy of privilege has been the growth of interest in more general protection systems and, in particular, in systems using capabilities. (It is worth noting in passing that the segment descriptors of Multics are essentially capabilities [Graham 1972] although Multics is not normally classified as a capability-based system.) Fabry [1968] argued in favour of hardware support for the implementation of a capability-based protection system and also introduced the important idea of copying capabilities. The Chicago Magic Number computer described by him was never completed for reasons which the author has not found in the literature. Nevertheless, Fabry's work has had a considerable influence on the design of the CAP.

The premature termination of the development of the CAL system is, on the other hand, well documented [Sturgis 1973,Lampson 1976]. The CAL project set out to develop an operating system with a protection system uniformly based on capabilities. It was terminated because the system proved to be neither efficient enough nor usable enough to be put into service, one of the reasons for the inefficiency being the implementation of protection domain switching entirely in software.

In another capability-based system implemented by software, the Hydra system [Wulf 1974,1975], the costs involved in switching protection domains are described as "considerable" [Cohen 1975] with the undesirable result that they discourage the use of Hydra's mechanism for protecting procedures. In the original Multics system the rings were implemented by software and the resultant high overheads caused design decisions to be taken which are being reversed [Schroeder 1975] now that the protection rings are implemented in hardware [Schroeder 1972(a)].

Hydra provides sophisticated protection mechanisms by means of which various protection policies may be implemented [Jones 1975,Levin 1975]. The mechanisms are based on five philosophical principles [Cohen 1975]:-

1. Information can be divided into distinct objects for purposes of protection
2. Objects are distinguished by type (c.f. class in Simula [Dahl 1966])
3. Access to objects is controlled by capabilities
4. Each program should execute with the smallest set of access rights necessary
5. All knowledge about the representation and implementation of operations for each type of object should be hidden in modules called subsystems.

The concept of type in Hydra centres round the notion that a type is an abstraction of a class of objects, and that the abstraction specifies not only the representation of the objects but also the

operations that apply to them. The protection mechanism ensures that manipulation of an object is possible only by invoking those operations defined for its type. A central feature of Hydra is that users may define their own types in which case the operations for manipulating objects of that type are specified as Hydra procedures. This makes the high cost of using such procedures particularly unfortunate.

Hardware support for protection, the lack of which adversely affects the Hydra system and proved fatal for the CAL system, was foreseen as a requirement by Fabry [1968,1974]. It is difficult to see how a sophisticated protection mechanism can be made cheap enough to use without such supporting hardware. The design of the Chicago Magic Number computer [Fabry 1968] included hardware for handling capabilities: there were special capability registers as well as the more usual data registers. The Plessey System 250 [England 1972,1974] also incorporates a set of capability registers, eight in this case, and provides hardware implemented mechanisms for manipulating capabilities and for switching protection domains.

The Cambridge University CAP system, which is described in detail in Chapter 3, is another with hardware support for capabilities. In a paper about the CAP system, Needham and Wilkes [Needham 1974(a)] set out the following reasons for providing a hardware supported protection system:-

"1. To make it possible for a user to write freely in any language he likes, including machine code, without danger that

he will, by inadvertance or by design, interfere with the operating system or interfere with other users.

2. To enable users to set up protected subsystems without any changes being necessary in the operating system.
3. To limit the amount of damage caused to stored information in the case of a hardware fault or software bug.
4. To make it possible for changes to be made to the operating system without incurring serious risk of catastrophic consequences if a software error is made."

They argue that hardware protection to satisfy their first objective is an essential for any multiprogrammed computer system to which users have free access. Objective 4, which is really a special case of 3, could be extended to cover the development of the operating system as well as changes to it. While developing the operating system for the CAP those involved have made the qualitative observation that bugs have been easier to locate and correct because the protection system has confined their effects.

The CAP system differs from the other capability-based systems described above in being a nested-address-space system whereas they are all global-object-name systems [Lauer 1974]. Hydra, for example, uses an unique identifier to distinguish an object from all other objects that ever existed in the past or will ever exist in the future. In the CAP an object is uniquely identified by the appropriate context-dependent interpretation of the address presented by the process wishing to access the object: the mechanism is described in detail in Section 3.3. Thus, in dif-

ferent contexts, a given object will, in general, be referred to by different addresses. Protection and addressing are intimately bound up in the CAP and the context in which an address is interpreted corresponds to a particular protection regime. Problems arise in connection with checking the validity of arguments passed as the parameters of a call to a protected procedure (i.e. a procedure call which involves a switch to another protection environment). This area is one that gives rise to difficulties whenever there exist several interacting protection regimes each having different access to some virtual address space [Schroeder 1972(b)].

Chapter 3

3. The CAP computer system

3.1 Protection domains

It is appropriate at this stage to clarify what is meant by a protection domain, particularly in relation to the CAP [Needham 1974(a)]. A process needs some body of code, a procedure, in which to run. During its lifetime a process may run in many procedures, some unique to itself, some shared with other processes which are running at the same time. At any particular moment, a process will have access to a certain set of data segments and there will be a certain set of procedures which it is entitled to enter. These sets of data segments and procedures constitute the protection domain in which the process is operating at that time. The protection domain may be modified by additions to or deletions from the sets of data segments and procedures. Also, a process may switch to operate in a different protection domain by entering one of the set of procedures in its current domain. Needham and Wilkes [Needham 1974(a)] take the point of view that changes in the domain of protection in which a process is running can only take place on a change of procedure, what is referred to in this thesis as domain switching; however, in practice domain modification has also been employed in developing the CAP Operating System. In this thesis the term domain change will be used to mean a modification to a domain and the term domain switch for the operation of making a new domain current by entering a procedure. Needham [1972] disc-

usses the use of domain changing and domain switching to bring about a change in the protection regime under which a process is operating.

3.2 The basic machine

The hardware and microprogram of the CAP computer are described in detail in the CAP Hardware Manual [Herbert 1978(b)]. An outline of the basic machine is given here with a more detailed description of those aspects which are particularly relevant to the research reported in this thesis. Unless specifically stated otherwise, the microprogram referred to is the normal mode interpreter written by Dr R.D.H.Walker.

The CAP is a word-addressable computer with a word comprising 32 bits. The macro-programmer has access to 16 registers (B0-B15) each of 32 bits and up to 192K 32-bit words of core store. This store is made up of two modules each of 32K, the Plessey store, which may be (and normally are) interleaved, and a single module of 128K, the Phillips store. The former has a cycle time of 2.5 microseconds, somewhat reduced by interleaving, whereas the latter is much slower with a cycle time of 10-12 microseconds and no possibility of interleaving. The different cycle times of the Plessey and Phillips stores means that care must be taken in interpreting the results of timing experiments. Absolute store addresses are represented by 20-bit patterns presented to the store bus. These are interpreted by the store hardware in accordance with the amount of store available at the time. Interposed between the CAP and the store bus are two 256-word modulo slave

stores for read requests and a 32-word modulo write buffer. These slave stores further complicate the interpretation of timings. All macro-instructions have the same 32-bit format:-

F (8 bits)	Ba (4 bits)	Bm (4 bits)	N (16 bits)

where

F = function code

Ba = one of the B registers: ba = contents of register Ba

Bm = one of the B registers: bm = contents of register Bm

N = an unsigned integer.

In the specification of instructions,

n = bm + N,

s =contents of the store location addressed by n:indicated by [n].

The CAP is not linked to peripherals directly, apart from the intimate teletype and the paper-tape reader controlled by the microprogram, but controls them via a hardware link to a peripheral processor (a Modular One). The peripherals currently connected are:-

(a) paper-tape reader

(b) paper-tape punch

(c) line printer

(d) multiplexor which can support up to 108 terminals - at present four are normally connected

(e) fixed-head fast Burroughs disc of 500K words

(f) moving-head CDC disc unit with exchangeable disc packs each of 7000K words.

At present the link is transferring data more slowly than had been expected, the transfer of a 32-bit CAP word taking about 8 microseconds. Most of the time taken is accounted for by computation in the Modular One.

The hardware logic of the CAP is divided into 9 pages:-

Page A : peripherals, stores, Modular One

Page B : slave stores

Page C : store control

Page D : Capability Unit

Page E : CPU

Page F : microprocessor control

Page H : micro-store extension

Page J : arithmetic

Page K : spare

The hardware configuration is given in Figure 3.1.

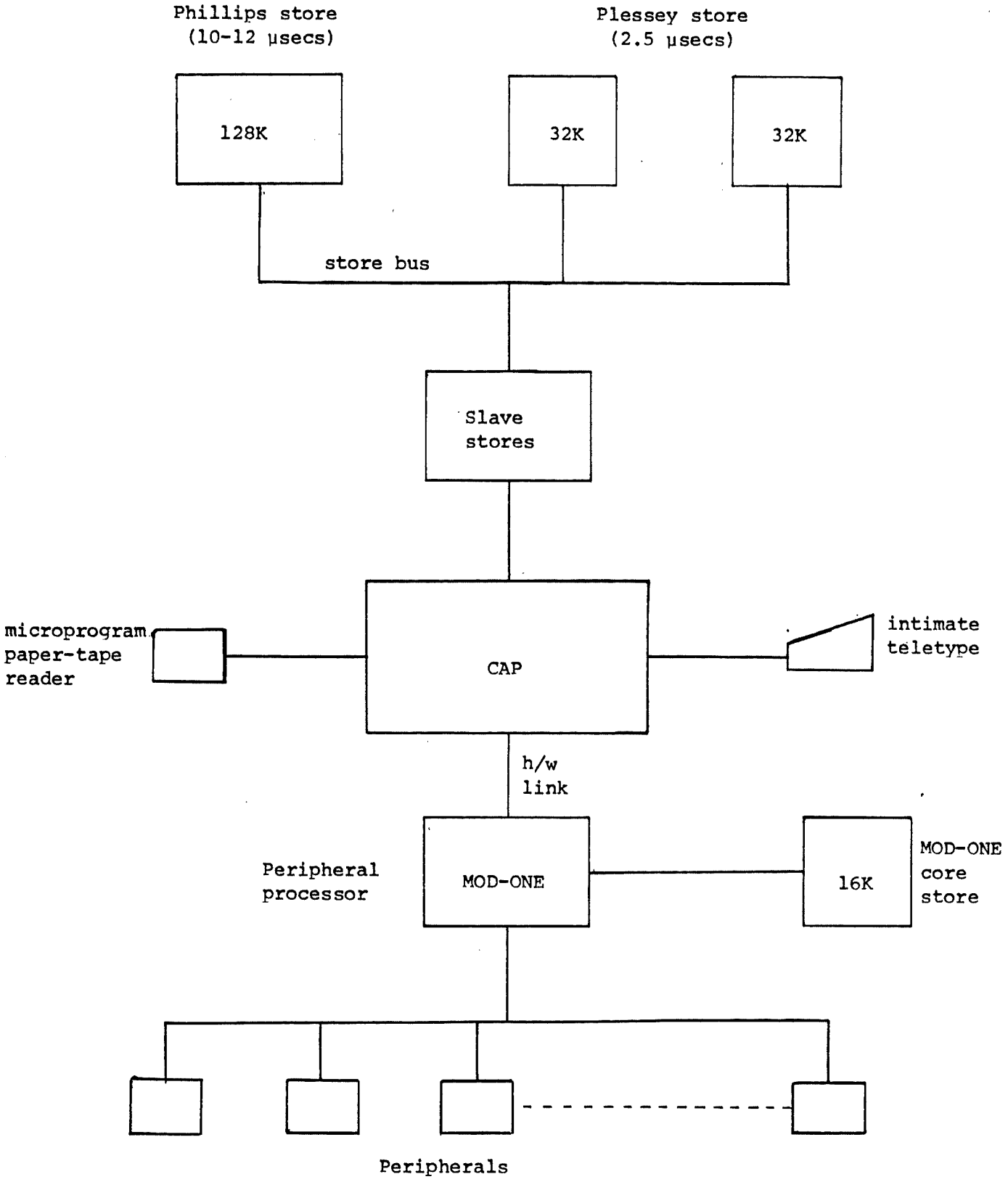


Figure 3.1 : CAP hardware configuration

The interpretation of machine instructions is done by the microprogram. Two registers, FR and AR, are loaded from the main store and the appropriate microcode is executed for the particular instruction. To facilitate this, a separate micro-store, the Function Memory (FM), contains one micro-instruction for each macro-order. This may be a jump to a section of the microprogram to deal with a complex machine instruction or, in simple cases, may in itself be sufficient to complete the interpretation of the instruction. Each Function Memory entry also contains 2 bits (FMR) giving access request information for that machine instruction.

The microprogram can access directly the 16 B registers and also 16 working space registers, the A registers, all of 32 bits. These can be connected to an arithmetic-logic unit whose output is gated into a fast shift register, register D. This is the central register, accepting results and data and transmitting data as necessary.

The arithmetic unit performs floating and fixed point operations and has its own separate control which is started by the microprogram. Data is transmitted to and from the unit via register D.

The entire machine is controlled by the microprogram. The microprogram store of 4K-64 words (the top 64 words are hard-wired) is volatile apart from 7 bootstrap instructions for loading the microprogram. The various parts of the CAP are controlled through a V-store. There are 256 V-addresses which are control signals with data to/from register D enabling various control registers

to be set and sensed and data to be moved to and from D. For example, the paper-tape reader used for bootstrap loading of the microprogram is controlled this way.

The overall layout of the microprogram processor is shown in Figure 3.2.

There are eight types of micro-instruction, each of 16 bits, and these are divided into three formats. Two hardware links are provided for use with the jump-and-link micro-instruction; thus, microprogram subroutines can only be nested to a depth of two (i.e. limited to a call within a called subroutine). Another link is used to store the microprogram location counter after a microprogram trap, i.e. on the micro-processor entering interrupt mode.

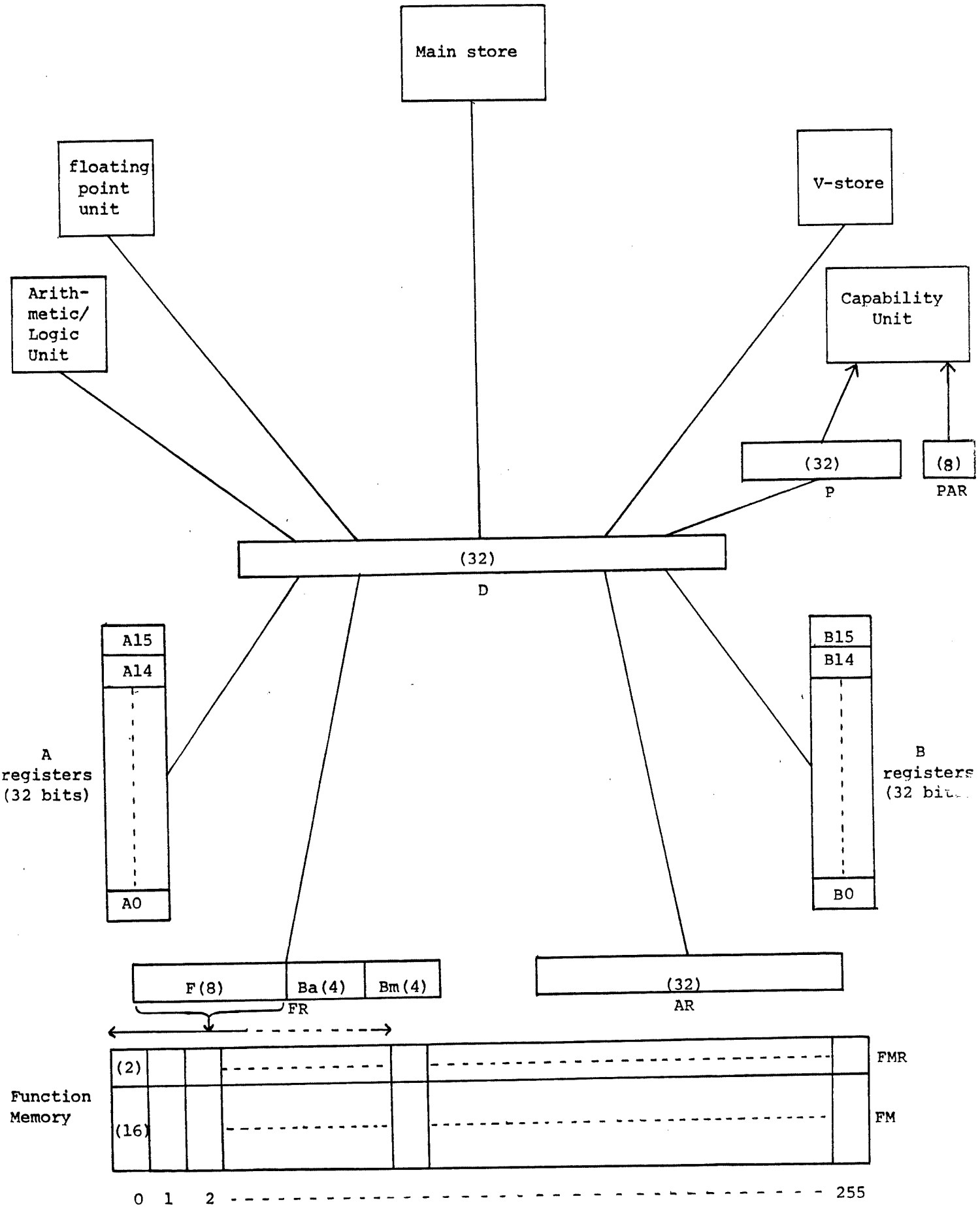


Figure 3.2 : CAP microprogram processor layout

When implementing many simple machine orders very few micro-instructions are needed besides those which are more or less common to all orders. There is a considerable benefit to be obtained from arranging for this common part to be done as quickly as possible and to this end the microprogram always executes its stage one even though this may have to be partly undone later in some cases. The operations incorporated in stage one are :-

(a) an instruction is fetched from main store and the program counter (B15) incremented

(b) the standard address modification (n = bm + N) is done

(c) for Read or Read-and-Write orders, a word is read from main store. Hardware assistance ensures that this is done only for R and RW orders. The 2 FMR bits in the Function Memory entry indicate the type of order.

(d) the appropriate instruction in the Function Memory is executed

(e) for Write and Read-and-Write orders, a word is written to main store.

In many cases, the single micro-instruction in the FM is sufficient to complete the macro-order. Otherwise, the FM instruction is a jump to a piece of microprogram to finish the work. External interrupts are dealt with at the beginning of the interpretation of macro-orders, that is at the time when the jump to the start of stage one takes place.

Description of the Capability Unit is deferred until later (Section 3.5) as it will be more readily understood after the process

structure and protection system of the CAP have been outlined.

3.3 Addressing and protection

The CAP architecture, which together with the CAP's protection system is described in Needham [1977(c)], supports an hierarchic process structure (Figure 3.3). The root process is known as the Master Coordinator and controls the processes immediately subordinate to it. However, the structure is completely general and these processes may, if they wish, act as coordinators for their own sub-processes. A process knows of its superior and its direct subordinates for which it acts as coordinator. It knows that its superior is its coordinator but does not know if any of its juniors is acting as a coordinator in its own right with its own set of sub-processes. Although in theory the hierarchy can be taken to an arbitrary depth, in practice a depth of two or three is likely to be sufficient. The CAP Operating System uses only a two level hierarchy of processes.

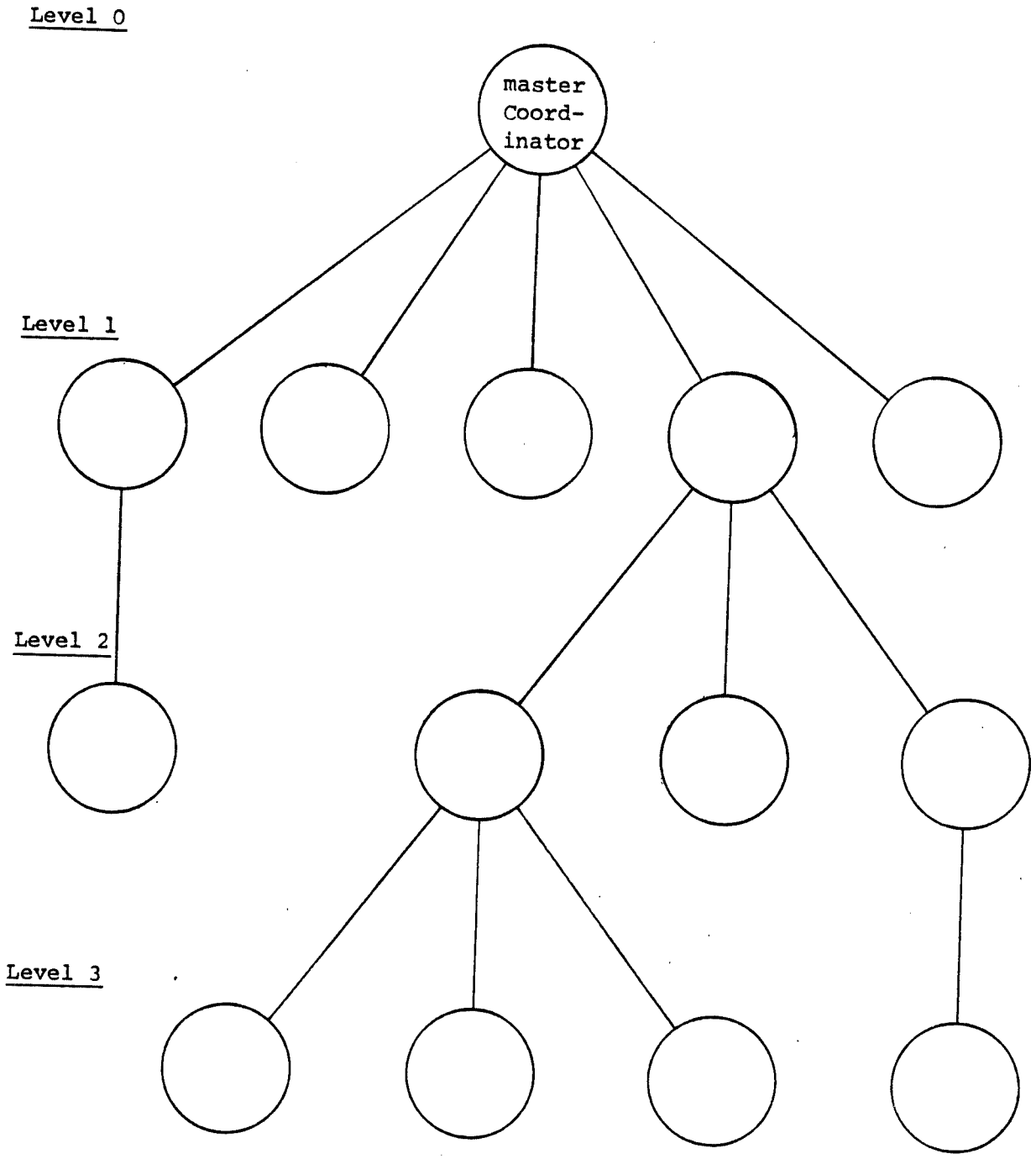


Figure 3.3 : CAP hierarchic process structure

Associated with each process is a Process Resource List (PRL), a Process Base (PB) and, if required, a C-Stack. The last is used in connection with domain switching (see Section 3.4) and the PB is used to hold information about the process and machine state on the process becoming dormant. The PRL contains a set of capabilities which define all the objects which that process potentially has the right to access. At any given time only a subset of these, as specified by a set of up to 16 capability segments, is actually available to the process. Only the PRL at level zero, the Master Coordinator's PRL which is alternatively known as the Master Resource List (MRL), holds absolute core addresses. All other PRLs hold capabilities which refer to the address space of the superior process. A process may set itself up as a coordinator and pass some or all of its rights to its juniors by placing the appropriate capabilities in the juniors' PRLs. However, for protection reasons a process is not allowed to pass down ENTER capabilities (see Section 3.4). This and other aspects of the CAP process structure are discussed in Walker [1973] and in Needham [1974(a),(b)]. Machine instructions are provided to enable a process to enter one of its juniors, ESP (enter sub-process), and for a sub-process to return to its coordinator, EC (enter co-ordinator).

An address presented by a process is interpreted relative to the address space of that process's superior. Such an address, called a general address, consists of three fields which indicate

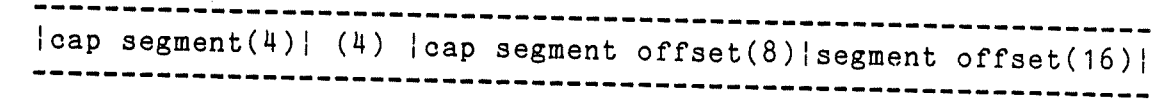
(a) which capability segment is to be used, the capability

segment field

(b) which capability in that capability segment indicates the required segment, the capability segment offset field

(c) which word in that segment is wanted, the segment offset field.

The first two fields together make up the segment specifier and the format of a general address is



The calculation of the absolute address corresponding to a general address presented by a process involves evaluating the segment specifier, by chaining back up through the capabilities in the capability segments and the entries in the PRLs of the process hierarchy until the MRL is reached, to obtain the absolute address of the segment's base and then adding the segment offset to obtain the absolute address of the required word (Figure 3.4). At each stage checks are made by the hardware to ensure that there is no violation of the bounds of the segment or of the access rights permitted to it by the capability or the PRL entry in use at that stage of the address mapping. An associative capability store (see Section 3.5) is used to short circuit this lengthy procedure for evaluating addresses. Since any address presented by a process is evaluated relative to the address space of its superior there is no need to constrain the addresses a process can formulate.

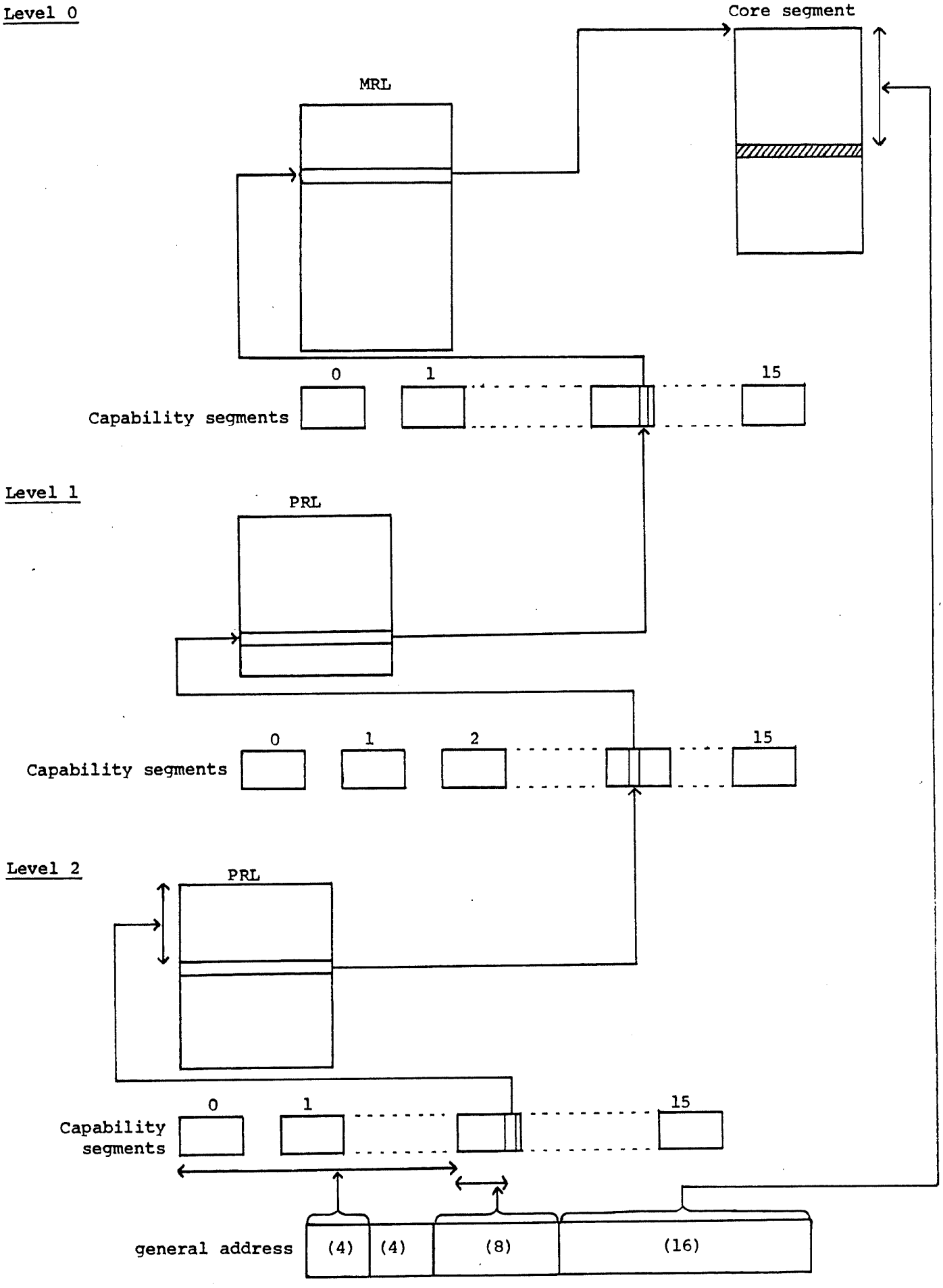


Figure 3.4 : CAP address evaluation (Level 2 process)

The protection supported by the CAP hardware is memory protection with the unit of protection being a segment, a set of contiguous words of memory. Segments contain either data or capabilities, where data includes executable code. A consequence of the use of capability segments for resolving addresses is that addressing is closely bound up with the implementation of protection. The protection domain in which a process is running at any time is specified by the set of capability segments in use at that time. The PRL for the process defines what Wyeth [1976] calls the protection environment of the process and its current protection domain is a subset of that environment. In the CAP a domain change is implemented by changing the contents of the capability segments and a domain switch by replacing the current set of capability segments by a new set. Typical of the kind of situation where it seems more natural to change a process's domain rather than switch to a new one is when the process needs additional resources, a new segment, say. The process can request a suitable segment from its coordinator which can make the segment available by placing an appropriate store capability in one of the process's capability segments. It should be noted that a segment that is a capability segment or a PRL to a process is just a data segment to that process's coordinator. In a situation where it is not that new resources are to be provided but rather that there is to be a change in the accessibility of resources already given to the process a domain switch is more appropriate.

3.4 Protected procedures

One of the most important features of the CAP architecture is the provision of hardware support for switching protection domains. The particular type of capability involved is the *ENTER capability* which is recognised by the hardware and can only be used to enable the process holding it to enter and run in a *protected procedure*. The process is not able to obtain access to information belonging to the protected procedure without entering it. Thus, if a process has in its PRL an ENTER capability for a protected procedure and is able to select that capability via its capability segments, it can switch to operate in a different protection domain by exercising that ENTER capability by means of the ENTER machine instruction. The new protection domain is that of the protected procedure which is entered, the appropriate capability segments being brought into service by the ENTER instruction. The C-Stack is used to keep linkage information so that the old protection domain can be re-established, perhaps slightly changed since capabilities may be returned as results from the called procedure, by the RETURN instruction.

Only five of the possible sixteen capability segments are actually replaced during a domain switch, the remainder being global to all procedures which the process executes and thus contributing to all the protection domains in which the process runs. Of these five new capability segments which must be set up three are specified in the ENTER capability itself. These are capability segments 4, 5 and 6, known as the *P*, *I* and *R capability segments*

respectively, which by convention are used for the following purposes:-

P - to hold capabilities common to all instances of the protected procedure in any process; for example, capabilities for the code segments of the procedure. A special mechanism, the Procedure Capability Segment [Herbert 1978(a)], is provided to enable P capability segments to be shared.

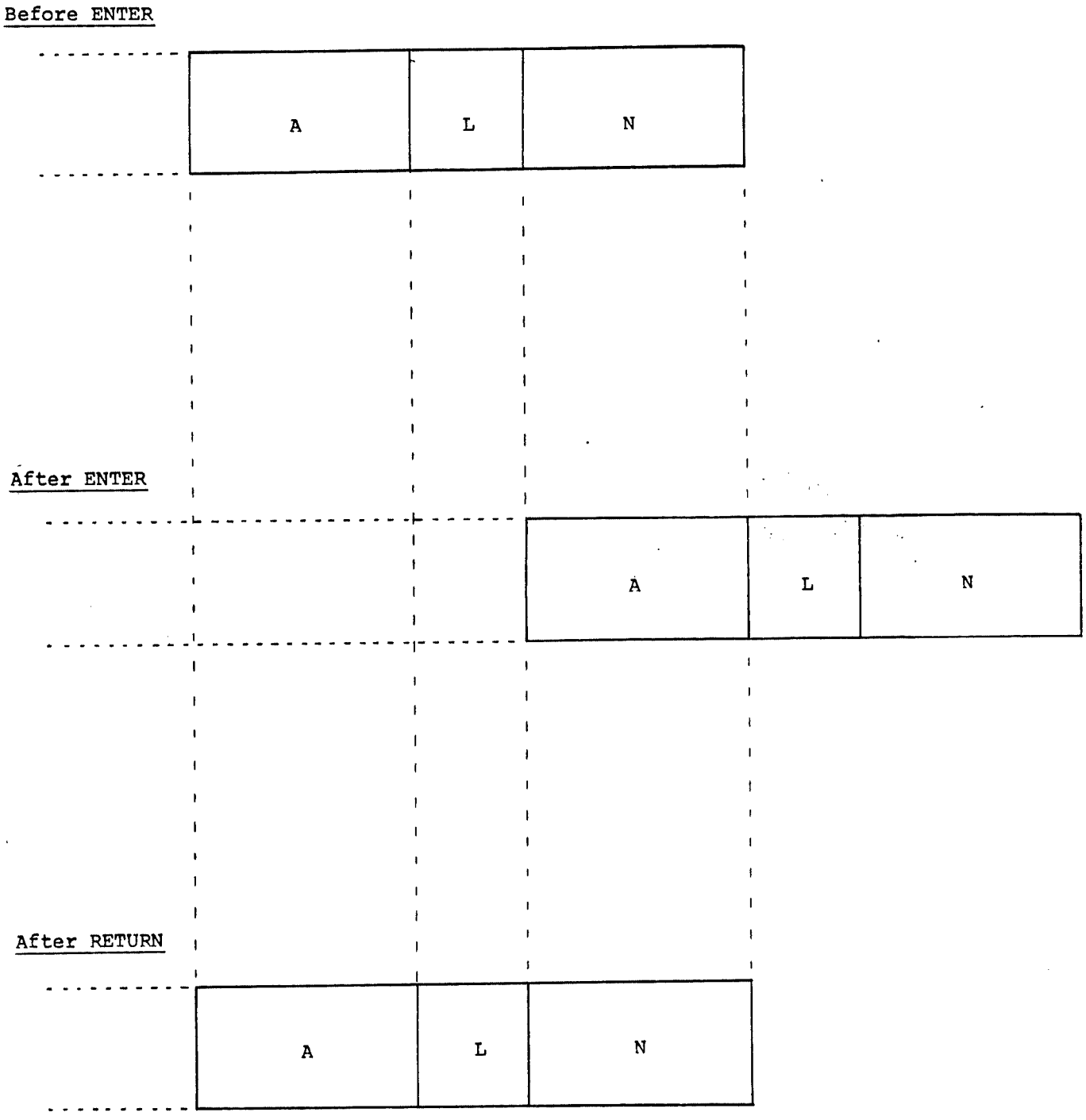
I - to hold capabilities specific to this instance of the protected procedure. It is, therefore, not intended that I capability segments be shared though there is no technical reason to prevent this.

R - to hold capabilities common to all instances of the protected procedure in this process only. The sharing of R capability segments can be arranged by specifying the same R capability segment in several ENTER capabilities.

The other two capability segments replaced are capability segment 2, the A capability segment, which holds capabilities passed from the old protection domain to the new one as arguments of the procedure call, and capability segment 3, the N capability segment, into which will be placed capabilities intended as arguments of a call to another protected procedure. The N capability segment of the old domain becomes the A capability segment of the new domain after execution of the ENTER instruction. The A and N capability segments are implemented as areas on the C-Stack. The

expansion and contraction of the C-Stack on executing an ENTER instruction and its matching RETURN are shown in Figure 3.5.

The hardware supported switching of protection domains by entering protected procedures provides the very flexible non-hierarchic protection mechanism which is an important feature of the CAP and complements the hierarchic protection inherent in the CAP's process structure.



A = 'A' capability segment

L = Linkage information

N = 'N' capability segment

Figure 3.5 : C-Stack - effect of ENTER and RETURN

3.5 The Capability Unit

The Capability Unit, which is interposed between the microprocessor and the store logic and whose operation is transparent to the macro-programmer, provides the hardware support for capabilities and also serves as an associative store to speed up the mapping of a general address into an absolute address. Its functions are :-

(a) translating a general address into a form internal to the Capability Unit.

(b) checking that an access request does not violate the size bounds permitted to the addressed segment.

(c) checking that the access requested is in accordance with the rights permitted to the segment.

(d) computing for an access request the absolute address suitable for presentation to the store logic.

Some of these functions may be suppressed according to the current mode (absolute, last, direct or normal) of operation of the Capability Unit. Mode selection is under microprogram control.

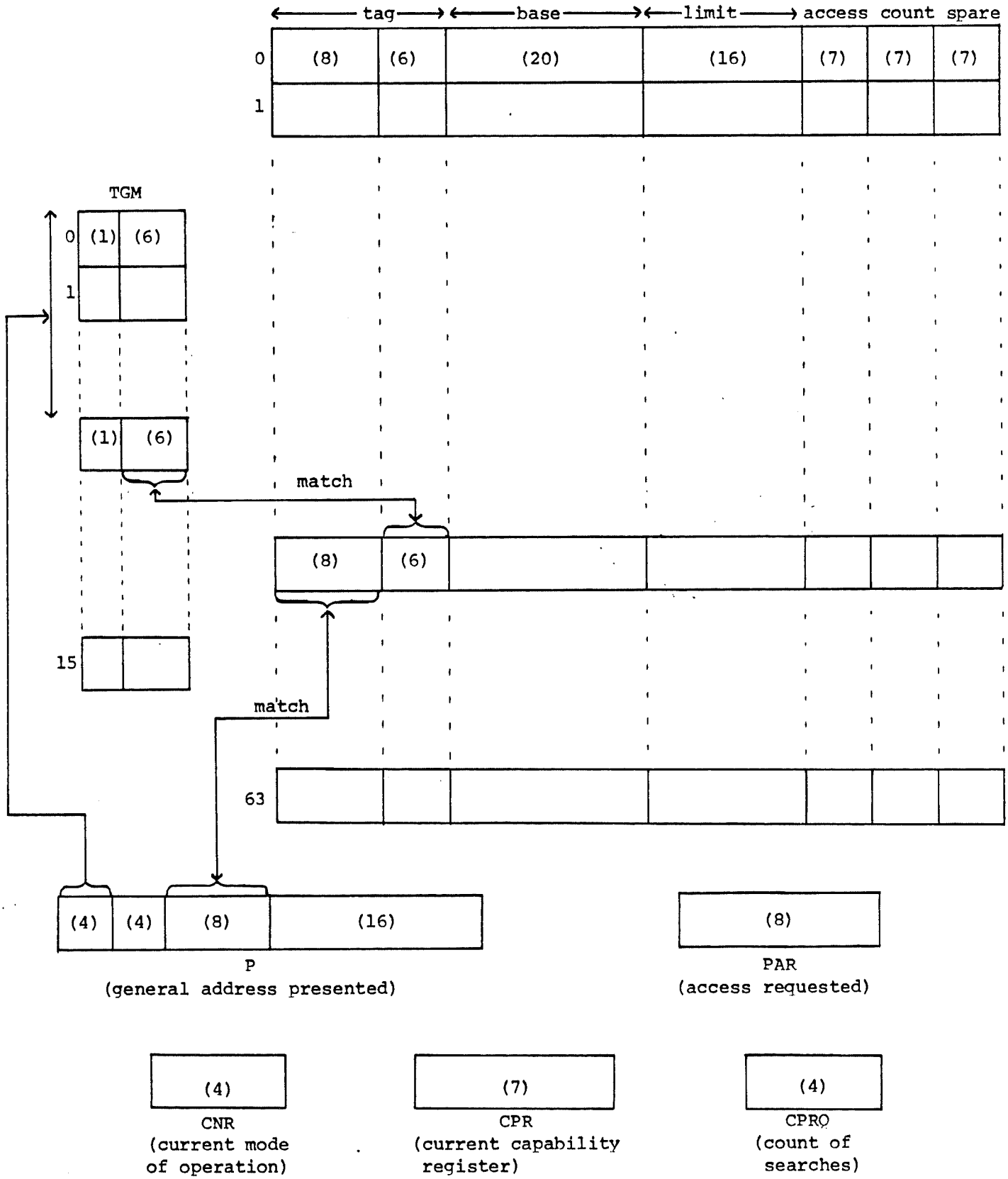


Figure 3.6 : Capability Unit layout

The Capability Unit (Figure 3.6) may be considered as consisting of:-

(a) a capability store of 64 capability registers of 80 bits (including 9 parity bits).

(b) a tag memory store (TGM) of 16 registers of 8 bits (including 1 parity).

(c) a register P into which the general address presented is written by the microprogram.

(d) a register PAR in which the access requested is placed.

(e) auxiliary registers and condition digits which may be read through the V-store.

Each capability register is made up of six fields which can be accessed independently by the microprogram and which collectively describe the location, length and access status of a segment in store. The fields are:-

(a) a tag field of 14 bits (+ 2 parity) used as an unique identifier within the Capability Unit for matching with a general address. This identifier is constructed so as to remain unique across any procedure or process change.

(b) a base field of 20 bits (+ 3 parity) giving the absolute address of the first word of the segment.

(c) a limit field of 16 bits (+ 2 parity).

(d) an access field of 7 bits (+ 1 parity). The least significant 6 bits represent the access rights permitted to the segment.

(e) count and spare fields of 7 bits (+ 1 parity) each for use

at the microprogrammer's discretion. These fields are not consulted by the hardware (apart from parity checking) during a store cycle. The normal mode interpreter uses the count field to keep track of the number of references to an entry in a capability register and the spare field of a given capability register to point to that capability's master.

During a store cycle the Capability Unit performs some autonomous computation. Subject to certain restrictions the microprogram may execute instructions in parallel with this and with the actual reading or writing of store. If the Capability Unit detects a machine error or an error in the access request presented to it the data transfer is aborted and control diverted to a trap location in the microprogram.

Whenever the microprogram generates a store request (by loading a general address into register P), 8 access bits are simultaneously loaded into register PAR. Two bits control special options in the Capability Unit and the other 6 are used by the Unit to check the legality of the request and by the store logic to generate the correct sequence of reading and/or writing. If the store request is to fetch an instruction, the access bits are derived from the 2 FMR bits of the appropriate Function Memory entry. The 6 bits are allocated the following meanings (d31 being the least significant of the six):-

d31 Execute

d30 Read data

d29 Write data

d28 Not used

d27 Read capability

d26 Write capability.

A one in the appropriate bit position indicates that the microprogram is attempting an access of the specified type. The access will be faulted by the Capability Unit if the access bits loaded into PAR have a one in some position and the corresponding bit in the access field of the currently selected capability register is a nought.

Before proceeding to check the store request one of the capability registers must be selected as current. The way this is done depends on the mode of operation of the Capability Unit at the time. In absolute mode no selection is made and all capability checks are bypassed. In last mode also no selection is made and the current register is the one most recently selected in direct or normal mode, the selection in direct mode being derived directly from six bits of register P. In normal mode - as the name implies this is the standard mode of operation - an associative search is made of the capability registers to find one where digits 26-31 of the tag field match digits 26-31 of the current TGM register (which has already been selected by digits 0-3 of register P, i.e. by the capability segment field of the general address) and digits 18-25 of the tag field match digits 8-15 of P, i.e. the capability segment offset field of the general address. (Note that the bit numbering here follows the hardware

convention that the least significant bit of a field is digit 31.) The associative search does not, in fact, take place on all 64 capability registers at once. Instead, the search is made on successive banks of 4 registers at a time. The bank selected as the start point for the search is derived from the value of the currently selected TGM. The TGM registers thus have two uses in normal mode operation:-

(a) as an indexed translation table to expand the top four bits of register P, the capability segment field of the general address, into the six bits used in the search cycle

(b) to determine the start point for the search.

If the Capability Unit search fails to find a match a "not found" trap is generated when the microprogram next attempts a STORE order. Note that reading from and writing to store needs two micro-instructions, a read or write request and a STORE instruction. The Capability Unit carries out its autonomous checking as a result of the read/write request, the store operation being held up until the Unit has completed its work. After a "not found" trap the microprogram performs the so-called reset cycle to calculate the required absolute address (or indicate an error if the requested address was ill-formed) and set up a capability register accordingly. The request is then repeated with a successful match now guaranteed. The reset cycle is relatively expensive in store cycles as it involves chaining up through the capability segments and PRLs of the process hierarchy until the MRL is reached. The

capability store is relied on to slave all the most recently used capabilities so that the reset cycle is only initiated when there is a request referring to a segment which has been dormant for some time or has not yet been used. It is quite possible for the reset cycle to trigger another reset cycle within itself, for example if it tries to read a capability from a capability segment for which there is no evaluated capability present in a capability register. In this case the first reset cycle is abandoned and will be repeated if the second reset cycle is successful. The microprogram endeavours to avoid overwriting capability registers which contain data necessary for another iteration but with a process hierarchy of more than two levels this is very difficult and the possibility exists of the machine looping in the microprogram. Present work on the CAP uses a two-level process hierarchy only.

The reset cycle makes use of the information stored in the master (spare) field of capability registers for capability segments in order to determine the addressing environment which is master to the current one. The master field of a capability segment points to the capability register of the appropriate PRL, whose master field in turn points to the capability register containing the capability for the Process Base of its superior.

In the CAP a general address at a given level of the process hierarchy is interpreted relative to the address space of the next higher level. Only in the MRL, the PRL of the topmost level, do capabilities contain absolute store addresses. The Capability Unit is used as a slaving device to avoid every general address having

to be evaluated by repeated indirection up to the MRL. Clearly, the capability for the MRL must be permanently allocated to one of the capability registers otherwise the machine cannot run in normal mode. By means of the tag fields in the capability registers a tree structure is maintained in the capability store reflecting the process structure relative to which the capabilities were evaluated. It is essential that the contents of the capability store continue to be a true description of the capabilities it holds, even after alterations to capability segments in main store have amended some of the capabilities currently in the capability store. The tree structure enables the microprogram to find those entries affected by a change to a capability represented by an entry higher up the tree. The maintenance of the tree means that, at any given moment, a number of the capability registers are reserved since they are non-terminal nodes in the tree. The capability store is also used for the relatively long-term retention of capabilities required by the protection mechanisms to identify the current Process Base, C-Stack and PRL. This results in additional capability registers being reserved.

When there is a switch of protection domain by entering a protected procedure a number of capability registers will be potentially reusable for new capabilities. However, if the protected procedure is returned from before the scavenging of invalid capability registers is complete then the protection mechanism will find that many of the capabilities now required are still in

the capability store. In the case of rapid protected procedure calls it is possible that the entire set of capabilities for the data and capability segments of two or three procedures may exist concurrently in the capability store leading to very rapid protection domain switches.

A comprehensive discussion of the Capability Unit, Tag Memory and reset cycle is given in Walker's thesis [Walker 1973]; the actual implementation differs slightly from the scheme he describes.

3.6 Peripheral devices

As mentioned earlier (Section 3.2), peripherals are driven by a Modular One to which the CAP is connected by a hardware link. Peripherals are protected via the memory protection mechanism by allowing access to a peripheral device only on presentation of a capability for a word of store corresponding to that device, each device having been allocated a word in the part of store known as the P-Store. Where this makes sense devices are allocated two P-Store words, one corresponding to reading from the device and the other corresponding to writing to it.

3.7 The CAP Operating System

The operating system is not dealt with in detail here: the current state is described in the CAP Operating System Manual [Herbert 1978(c)]. A brief description of the various processes and protected procedures which make up the operating system is given in the Glossary. The design and structure of the operating

system are, naturally, intended to make extensive use of the capability-based protection mechanisms of the CAP. The design is thus centred around the use of capabilities and a primary design objective is that the principle of minimum privilege (see Section 2.1) should be applied. The building blocks from which the operating system is constructed are protected procedures providing services which may be roughly classified under the four headings of gate-keeping, operating system intervention, protected objects and trivial services [Needham 1977(c)]. This classification reflects differences in emphasis and purpose of the four types of protected procedure: the mechanism for their use is the same in all cases. The basic design features of the CAP Operating System are:-

(a) The use of capabilities to provide protection [Needham 1977(a),(c)]. In addition to the store capabilities, which are dealt with by the hardware, and ENTER capabilities, which are presented to the microprogram for interpretation, there are capabilities which are checked by software: these are known as <u>software capabilities</u>. Software capabilities are protected in just the same way as store or ENTER capabilities and are kept in capability segments. They are used, for example, by the Master Co-ordinator to check whether one of its sub-processes which is requesting a particular service is entitled to that service. The process indicates its entitlement to the service by presenting the Master Coordinator with an ap-

propriate software capability. For instance, to stop the system a system-stop-permission capability must be presented. The capability management services of the Master Coordinator, for which the requesting process must have a capability-permission capability, provide examples of the use of domain changing as opposed to the switching of protection domains (see Section 3.1).

(b) The principle of minimum privilege is a fundamental design principle. Privilege here refers not only to the rights of access to store but also to the use of privileged services provided by the Master Coordinator and the right to enter protected procedures or to activate operating system processes to carry out certain services.

(c) A message system is provided by the Master Coordinator to enable its juniors to communicate with each other. The use of the message system is controlled by means of software capabilities.

(d) The virtual memory system is unusual in being handled entirely by software without hardware support. Memory is segmented, not paged, with segments of arbitrary size (from 0 to 32767 words). The segment is therefore the unit of swapping as well as the unit of protection. Segmentation is under user control.

(e) The explicit concept of a file does not exist. Rather there is the concept of a Virtual Memory Object (VMO) which may be in core (an inform segment) or swapped out

(an outform segment). Special facilities are provided for dealing with VMOs larger than 32K-1 words: these can only exist on disc since such large segments cannot be handled in core. There is a mechanism for retaining capabilities for VMOs between runs and a retained VMO corresponds to most people's intuitive idea of a file. The structure of the filing system is unusual in being a directed graph rather than the more normal hierarchic tree structure [Needham 1977(b)].

(f) Input and output are implemented as character streams and facilities for manipulating these streams are provided via protected procedures created in the process which wishes to indulge in input or output.

(g) Facilities are provided to enable a procedure to be advised of and given an opportunity to recover from any faults that may occur while a process is running in that procedure [Needham 1971]. Steps are taken to avoid looping if a procedure faults again while dealing with a fault.

(h) Most of the operating system has been written in Algol 68C [Bourne 1975] with a few small procedures written in machine language either for reasons of efficiency or because the computation must be done entirely in the registers.

The above description has implied that all coordinator services are provided by the Master Coordinator. In practice, some of the work is done in a protected procedure, ECPROC, in the process req-

uesting the service. The handling of faults is done by a procedure, FAULTPROC, which runs in the same protection domain as ECPROC. The main advantages of putting as much work as possible in ECPROC are that this avoids unnecessary serialisation, that because it runs in-process much argument checking can be done by the hardware and microprogram, and that the amount of code in the Master Coordinator is reduced. Thus, a domain change may be effected by means of a domain switch to enter the ECPROC protected procedure.

Chapter 4

4. Performance evaluation

4.1 Review of relevant work

The CAP computer was commissioned towards the end of 1974. Although various ad hoc monitoring experiments have been done since then there has been no organised performance evaluation work. The CAP architecture was simulated [Walker 1973] prior to the building of the actual hardware and some measurements (e.g. instruction counts) were made with this simulator.

The literature on performance evaluation and monitoring falls into three categories. Firstly there are general survey papers [Bell 1973,Calingaert 1967,Lucas 1971,Lynch 1972,Williams 1972] whose main body is usually a discussion of the comparative merits of different monitoring techniques, the techniques covered being the use of personal inspection, accounting systems, hardware monitoring, software monitoring, benchmark programs, simulation models and analytical models [Bell 1973]. Secondly there are papers dealing with particular aspects of performance evaluation or with particular techniques. Examples of such papers are [Bard 1973,Boi 1973,Boyse 1975,Kimbleton 1972,Knuth 1973,MacEwan 1974,Sketler 1974]. Thirdly there are presentations of the results of individual case studies. The volume of material in this category is overwhelming, especially if one includes papers published elsewhere than in the major computing journals. A sample of the

work in this category is provided by papers about Multics [Saltzer 1970,Schroeder 1971], TOPS-10 [Jalics 1974], UNIVAC 1108 [Bordsen 1971] and the Michigan Timesharing System [Pinkerton 1969].

Essentially the message that comes over from papers on performance evaluation is that one should use the tools and techniques most appropriate to the task in hand! "Each analyst must feel his way to a solution for each problem with only helpful hints for guidance" [Bell 1973]. "Ordinary common sense is here, as elsewhere, to be desired" [Calingaert 1967].

One technique which has been very little used and whose merits and demerits are not discussed in the general review papers is the use of a microprogrammed measurement system. A survey paper on microprogramming [Rosin 1974] and an earlier review paper by the same author [Rosin 1969] likewise make no reference to the use of microprograms for performance evaluation. The author is aware of only three reported applications of microprogrammed performance measurement, one on the Burroughs B1700 [Denny 1975,Wilner 1972], one on the extension of this work to the Burroughs B1800 [Denny 1977] and the other on a Standard Computer Corporation IC7000 computer system [Saal 1972,1975]. Saal [1972] contends that "given an existing system with a writable control store, a microprogram measurement system may be the most flexible, inexpensive, reliable, and high-speed means of monitoring the performance of a computer system".

4.2 Performance evaluation: the CAP computer

As far as possible monitoring tools should be designed with

flexibility and generality in mind so that they may be used for a variety of experiments. The research described in this thesis included only a limited amount of performance evaluation work done in connection with measuring the cost of protection on the CAP (see Chapter 7), but even so this design aim was applied. The monitoring tool used in the experiments proved to be easy to adapt to the particular requirements of each experiment; it should also be a simple matter to use it to carry out some of the investigations suggested for further research (see Section 8.2).

There are no magnetic tape units attached to the CAP. There is disc storage but data stored on disc is not as secure as data stored on magnetic tape. There is a strong incentive, therefore, for any monitoring experiments done on the CAP to be designed to produce small volumes of data or else to make use of data compression techniques [Batson 1970,Denning 1971]. The author was able to adopt the former approach for his experiments.

The CAP computer has store modules with different speed characteristics. Because of this, the presence of slave stores for reading and writing, and the slaving function of the Capability Unit (all of which are described in Chapter 3) the results of timing experiments must be interpreted with care. A number of such experiments were done and in all of these a stop-watch was used for measuring time. The Modular One computer used as a peripheral processor has a real time clock which could have been used instead. To the CAP this looks like a serial input device and, although the clock's "tick" can be set down to one centisecond,

too short an interval imposes high loads on the microprogram and the Master Coordinator to handle the interrupts generated. The author decided that a stop-watch would be sufficiently accurate, would perturb the system less than using the Modular One's hardware clock and would be simpler to use.

The CAP incorporates various hardware monitoring facilities, in particular a counter. By appropriate setting of the manual switches this counter can be used to count occurrences of a wide variety of events. As it is possible to count the number of times a specified microprogram instruction is obeyed the hardware counter is a very flexible monitoring tool. It is simple to use but has the disadvantage that occurrences of only one type of event can be counted at a time. Details of the CAP's monitoring facilities are given in the CAP Hardware Manual [Herbert 1978(b)].

The use of the microprogram for monitoring introduces certain problems. Microprogram store is limited so compactness of code is essential, even if it is at the expense of clarity. Another constraint is that subroutine calls can only be nested to a depth of two (i.e. the limit is a subroutine call from within a subroutine). These constraints together with the disadvantages inherent in the use of a low-level language, in this case an assembler which sometimes has side effects which are not obvious from the written code, make it difficult to understand and modify the microprogram. In addition, great care must be taken to ensure that an unexpected interrupt will not lead to disaster.

To conduct the experiments described in this thesis (see Chapter

7) the author used a combination of modifications to the microprogram, monitoring code inserted in the Master Coordinator program and event counting using the hardware counter. This provided a very flexible and powerful approach to monitoring the performance of the CAP.

Chapter 5

5. Measuring Protection

A general expression of the need for protection measurement is given in Weissman [1972]: "For the future the (computer security) field must confront what I believe to be the two major problem areas - metrics and certification. As in most engineering endeavours, we cannot improve performance, lower costs, or even identify significant variables until we can quantify, measure, and establish numerical scales of values". In a 1976 editorial Waite [1976] made a plea that efforts to improve software reliability should include cost/benefit analyses. Protection has a contribution to make in improving the reliability of software but, as Waite pointed out, although "it seems possible to estimate costs rather accurately, there is almost no way to estimate benefits". Randell [1977], too, mentions the lack of quantitative design tools. "Ideally, all ... design issues would be decided upon ... by mainly quantitative methods. ... However it would seem that many of the design tools involved in achieving high levels of overall reliability from large and complex hardware/software systems will continue for a long time to require large measures of creative skill and experience on the part of the designers." A method of measuring the benefits and costs of protection would enable the designers of computer systems, especially operating systems, to quantify in part the results of their endeavours.

There seems to be a need for a measure suitable for the ordinary

user too. Experience with Multics has shown that users have difficulty in making use of sophisticated protection mechanisms [Schroeder 1972(b)]. Protection measurement would contribute to the comprehensibility of protection mechanisms [Redell 1974] by giving the user a means of quantifying the results of his actions.

5.1 Protection costs

Measuring the costs of the protection provided by a system is, in principle, relatively straightforward. However, the protection may be closely bound up with other aspects of the system, for example protection and addressing are inextricably intertwined in the CAP, and in practice it may well prove difficult to apportion certain cost elements. It would be hard, for example, to allocate the total hardware cost of the CAP's Capability Unit between the addressing and protection operations in which it participates. The cost model proposed by Wyeth [1976] is used as the basis for measurements of the costs associated with the provision of protection. Cost figures are of interest principally when comparing different strategies on the same system. Provided the cost elements which are hard to apportion are fixed (as opposed to variable) costs then the apportionment problem is not a significant one.

Wyeth identifies six cost components:-

(a) Cost of creation of a protection domain

(b) Cost of deletion of a domain

(c) Cost of maintenance of a domain

(d) Cost of switching from one domain to another

(e) Cost of changing a domain

(f) Cost of protection enforcement.

Each cost component involves the invocation of a set of primitive actions and the actual form of Wyeth's cost measure for a particular protection mechanism is obtained by determining the primitive actions

$$P_{i1}, P_{i2}, \ldots P_{ik}$$

involved in the i-th cost component (i=1,2,...,6) and the costs

$$C_{i1}, C_{i2}, \ldots C_{ik}$$

of each primitive. The total cost is calculated by summation.

5.2 Protection benefits

In trying to formulate a suitable protection measure the author attempted to determine the objectives that such a measure could be used to attain. Two approaches correspond to meeting the following two objectives:-

(a) to provide a user with a measure of how well (or badly) protected are the various objects (e.g. data files, programs) which he owns in the system

(b) to establish how closely the principle of minimum privilege (see Section 2.1) is adhered to in a computer system.

The first objective requires a measure (an exposure measure) that considers the degree of protection which has been provided for an individual object; the second requires a measure (a privilege measure) that considers the privileges which have been allowed to a subject (using object and subject in the sense of Graham and

Denning's model [Graham 1972]).

5.2.1 Exposure measures

An exposure measure would enable the user of a computer system to do cost/benefit analyses of the different degrees of protection he could request for an object. This would help him to decide how much protection he was prepared to pay for. Ellis's [1974] degree of protection of a system is an exposure measure.

The components of Ellis's model of a protection system are:-

(a) Finite set, A, of active elements called subjects;

$A = \{A1, A2, \ldots Au\}$.

(b) Finite set, B, of passive elements called objects;

$B = \{B1, B2, \ldots Bv\}$. An element may be both active and passive. Therefore, in general, $A \cap B \neq 0$.

(c) Set $C = A \cup B$

(d) An access code, c_i, a binary n-digit number, is associated with each $C_i \in C$.

Call an access code a_i for a subject A_i, b_j for an object B_j.

Let c_{ik} be the k-th binary digit of access code c_i.

(e) An access mechanism, $g(a_i, b_j)$, is defined as a total boolean function of an ordered set of two access codes, and takes the value 1 if an access from A_i to B_j is allowed,

0 if such an access is not allowed.

In his paper Ellis studies the family of access mechanisms

consisting of boolean functions applied bitwise to the two access codes and summed. This family is expressed formally as

$$\begin{cases} g=1 \text{ if } \sum_{k=1}^{n} f(a_{ik}, b_{jk}) \geq m, \\ g=0 \text{ otherwise} \end{cases}$$

where f is an arbitrary boolean function of two binary digits,

n is the number of bits in each access code,

m is the access threshold and satisfies $0 \leq m \leq n$.

An example of such a mechanism is the IBM 360 hardware key system [IBM 1970]. Ellis considers the situation where the protection mechanism is unable to provide access codes such that the assignment of these codes to subjects and objects is possible in a way that will give each subject exclusive access to one or more objects. In other words, he is concerned with systems in which the access mechanism is not able to prevent unauthorised accesses from taking place: one or more subjects may be assigned access codes which allow access to objects to which the subjects should not have access. His measures of the absolute and relative degrees of protection of a system indicate the extent to which the system is susceptible to such unauthorised accesses. These measures are defined as:-

(a) absolute degree of protection of a system,

$$d_{abs} = (1 + y)^{-1}$$

(b) relative degree of protection of a system,

$$d_{rel} = \frac{|A| - x - y}{|A| - x}$$

where $|A|$ is the cardinality of the set A,

x is the average, over B, of the number of subjects which have authorised access to any particular $B_j \in B$,

y is the equivalent average for unauthorised access.

The first measure is absolute in the sense of not varying according to the size of the system but only according to the number of unauthorised accesses. The second measure is relative to the size and structure of the system.

If no unauthorised accesses are allowed, $d_{abs} = d_{rel} = 1$.

Whereas Ellis is interested in systems where unauthorised access is unavoidable, the author's concern is with systems where the protection mechanism has the ability to prevent all undesired access. Ellis's measure is, therefore, inadequate. (In practice, the full power of the mechanism may not be used, either because of the cost of doing so or for other reasons.)

There are two major problems with exposure measures. Firstly, the notion of ownership is not sufficiently clear-cut: it is difficult to see how it could be applied in the CAP system where, for example, an object created by Jim, who initially retained a capability for it, might remain in existence long after Jim had disposed of his capability, the object's continued existence being caused by Bill also having retained a capability for it. The second, and more important problem is that the degree of exposure of an object or set of objects tells you nothing about whether that degree of exposure is essential to enable the object to be

used as was intended. On the other hand, a privilege measure addresses just this problem. In terms of the access matrix model of a protection system [Graham 1972], an exposure measure summarises the access matrix column by column (object by object) whereas a privilege measure summarises it row by row (subject by subject). It is fairly easy by looking at a subject's algorithms to find out what that subject should be able to access so a privilege measure is helpful in assessing whether the subject has more privileges than it needs. An exposure measure is of little help in assessing overprivilege because it is difficult to determine which subjects should have access to an object by looking at the object itself.

5.2.2 Privilege measures

As mentioned earlier (Section 2.1) the objective of Jones's [1973] model of a protection system is the attainment of the principle of minimum privilege. This need-to-know principle is the basis from which stem her accuracy and suitability measures. She postulates demands defined as constraints on the contents of protection domains and on the new protection domains which a process may enter and proposes that her measures be used to compare protection systems. The accuracy measure indicates how accurately a protection domain suits the execution performed within it. It is defined for an execution environment as

> "the ratio of the number of rights exercised in the environment compared to the total number of rights which could be exercised within the environment during the performance of one

task".

As Jones points out, the accuracy measure does not reflect the understocking which occurs if an unnecessarily fine decomposition of a problem solution generates multiple tasks each to be performed in almost identical protection domains. The accuracy measure has a maximum value of 1, when the domain is exactly tailored to its use, and a lower bound of 0 approached as the number of unused rights in a domain increases. As a measure of how well a particular system satisfies a given demand Jones introduces the suitability measure which is defined as follows:

> "The suitability measure of a system with respect to a given demand is the average of the accuracy measures of the implementations of all domains required by the demand specification".

Thus, provided the demand specification reflects the minimum privileges required, Jones's suitability measure supplies a means of determining how closely the principle of minimum privilege is approached. Jones does not go into the details of how numerical suitability factors would be calculated in practice and therefore does not encounter the problems which are discussed later in this section and in the ensuing one (5.2.3). As an example of the use of her measures, Jones presents a qualitative comparison of several representative systems which results in a partial ordering of them by their ability to satisfy a specific demand. In comparing protection systems Jones takes no account of the costs involved in the provision of protection in each case.

Wyeth [1976] sets out a methodology for comparing protection systems which covers both costs and benefits. His work, which is strongly influenced by the accuracy and suitability measures of Jones, is, as far as the author is aware, the only previous attempt to make quantitative comparisons of the costs and benefits of protection systems.

Wyeth defines the environment of a process as

> "the set of existing objects potentially accessible by the process",

and the domain of a process at time t (measured in process time) as

> "the set of access permissions which the process is able to exercise at time t".

Thus, in the CAP system, a process's environment is specified by the entries in its PRL and its domain at a given time is defined by the capabilities in the set of capability segments in use at that time. Wyeth uses a simple protection model which consists of the identification of five sets of objects pertaining to a process: the totality of objects in the system, T; the environment of a process, E; the theoretically accessible set of objects, A; the accessible set as defined by an implementation, A'; the referenced set, R. The sets A, A' and R are defined as follows:-

(a) "The theoretically accessible set, A, is a subset of the environment, E. A contains those objects within E which the process may access at time t, ... (the specific objects included in A being) ... defined by some external specification."

(b) "The accessible set as defined by a particular implementation, A', is the set of objects ... (which the process can address) ... at time t."

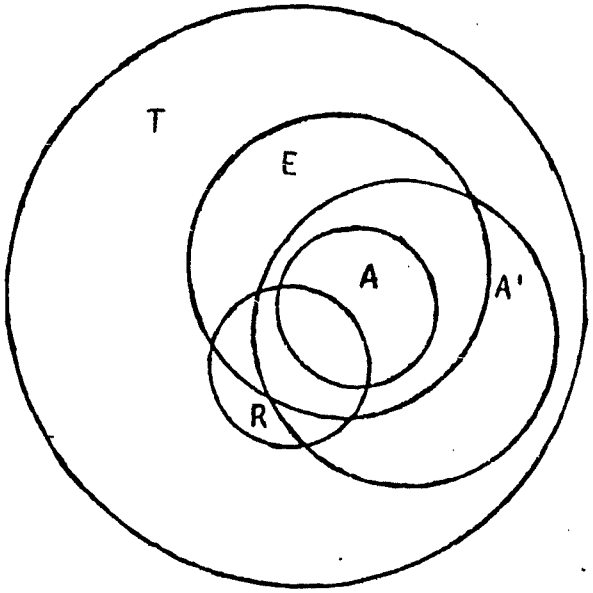
(c) "The referenced set, R, (initially empty) corresponding to domain D, is composed of those objects referenced by the process up to time t since the process switched execution to domain D."

The set A corresponds to what should be accessible to a process at a given time, whereas the set A' corresponds to the objects which the process can actually address at that time.

An example may help to make the distinction between A and A' clear. Consider a program written in Algol 60. The scope rules of the language lay down the specific subset of all the variables declared in the program which is validly accessible during the execution of a block or procedure in the program. This subset includes the local variables declared in the given block or procedure, the variables in the scope of the enclosing block but not re-declared as local variables, and any parameters passed to enclosing procedures. The environment, E, corresponds to all the variables declared in the program. The theoretically accessible set, A, at some time during the execution of the given block or procedure corresponds to the subset defined by the scope rules, which are the external specification in this example. (The number of instructions executed will suffice as a measure of time.) The set of objects actually accessible in the given block or procedure when the program is run is determined not by the scope rules but

by the way the compiler writer has chosen to implement those rules. This set corresponds to A', the accessible set as defined by a particular implementation. The compiler writer may adhere strictly to the scope rules, in which case A = A', or he may choose to ignore some of the restrictions imposed by them, in which case $A \subset A'$.

The relationship between the sets of Wyeth's model are shown by means of a Venn diagram in Figure 5.1 which is taken from Wyeth [1976].



T	totality of objects
E	environment
A	theoretically accessible set
A'	accessible set defined by an implementation
R	set of objects actually referenced

Figure 5.1 : Relationship between the sets of Wyeth's model

The protection mechanism is presumed to detect attempts to access objects not contained in A'. An undetected protection violation will occur when an object $X \in A'$ is referenced and $X \notin A$. Wyeth, like Ellis [1974], is concerned with measuring the potential for unauthorised access. He defines his measure as follows:

> "The value or benefit of a protection mechanism is defined to be $|A|/|A'|$ (where $|A|$ denotes the cardinality of the set A) averaged over the domains of execution of a sample set of programs".

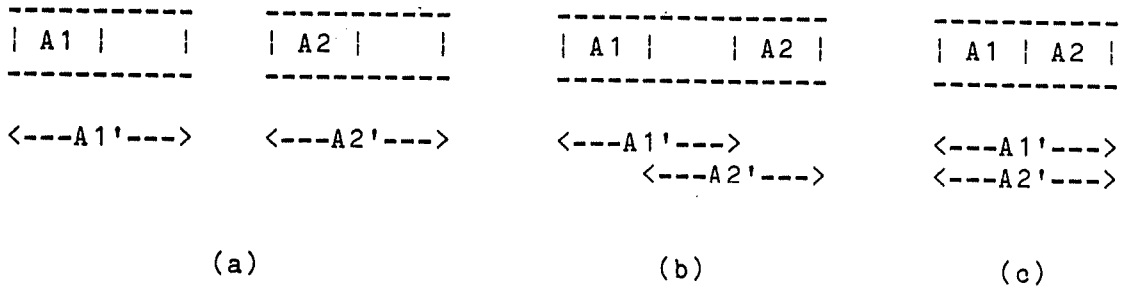
Wyeth claims that, since his measure is independent of the referenced set R, it is independent of the detailed behaviour of the actual programs and asserts that this independence is necessary to avoid measuring aspects of processes rather than of protection mechanisms. However, the sets A and A' may change dynamically, the precise changes depending on the detailed behaviour of the actual program, so that Wyeth's measure does not achieve the independence he claims for it.

In terms of Wyeth's model, Jones's accuracy measure for a given domain is the ratio $|R|/|A|$. Averaged over a number of domains this becomes Jones's suitability measure. These measures are dependent on the detailed dynamic behaviour of processes and are intended to be so. This concern with process behaviour dependence is related to the problem of whether one should attempt to measure a protection mechanism or an application of that mechanism. In other words, is it more useful to measure what a particular protection mechanism can do or how it is used? As the author's

research was concerned with evaluating the CAP computer system he concentrated on measuring how effectively the CAP's protection mechanisms were applied in practice rather than assessing the power of the mechanisms themselves. He did not, therefore, insist on a measure of the benefit of protection being independent of the actual behaviour of the processes contributing to the measurement.

Wyeth applied his methodology in two practical experiments, one comparing various implementations of Algol W, and the other comparing the IBM System 360 DOS/VS operating system with a modified version. These experiments are not gone into in detail here although they demonstrate some of the shortcomings that are discussed next.

Wyeth glosses over the difficulties involved in applying weights to the objects in the system according to their importance. To illustrate the need for such weighting, consider two processes P1 and P2 with theoretically accessible sets A1 and A2, and implementation defined accessible sets A1' and A2' respectively. Suppose |A1| = |A2| and |A1'| = |A2'| = 2*|A1|. Consider the three possible situations illustrated below, where the boxes represent portions of memory. In each case Wyeth's protection benefit measure for the two processes = 1/2. But the protection implications in the three cases are very different.



Qualitatively, p(a) > p(b) > p(c) , where p(x) is a measure of the degree of protection of the pair of processes in case (x). As another example, in the CAP project we would consider access to a segment which is a PRL to be much more important from the protection point of view than access to an ordinary segment of the same size. Wyeth's experiment with DOS/VS demonstrates how ignoring value-weighting renders his approach less effective. The main difficulty is in deciding what weights to apply: value judgements are subjective and are, therefore, incompatible with an objective measure. Ellis and Jones do not consider the value-weighting problem at all.

Wyeth limits himself to dealing with the accessibility of data. He does not consider how to incorporate a process's rights to ask for services to be performd by the modules in the system. In the CAP, limiting the distribution of ENTER capabilities for protected procedures controls access to the services provided by those procedures. Controlling the availability of services is very much part of the CAP's protection and should be included in protection measurement. However, there are non-protection reasons for deciding how to split the software of a computer system into modules. At present system designers have to take such decisions on

the basis of intuition and experience: a suitable protection measure would aid them in this decision making.

Another problem associated with privilege measures of protection is deciding what to use as the basic unit when specifying the things to which a process needs to have access in order to perform its current task. Taken to its limit in relation to memory protection the principle of minimum privilege would restrict a process to accessing a single bit at a time. In practice a coarser grain is appropriate: for memory protection on the CAP a 32-bit word seems the natural choice as the basic unit. The unit for the services provided by modules is less easy to select. A possible choice is to use as a unit each individual service described in the specification of the module. In practical studies of protection using a privilege measure, basic units would have to be chosen which were appropriate for that particular experiment.

5.2.3 Protection benefits: pragmatic considerations

This section on protection benefits is terminated by a brief discussion of two further points.

Protection domains will, in general, be subject to change and the particular set of domains in which a process runs during its lifetime will depend on the the detailed execution of the process. For a given domain at a given time it is possible, by inspecting the specification of the domain, to determine all the data segments which the process is allowed to access and all the procedures it has the right to enter at that time. The extension of the search to discover all the objects in the system for which the

process could potentially acquire rights, although theoretically possible, is likely to be impractical in a real system especially if, as in the CAP Operating System, certain processes are able to manufacture arbitrary capabilities. The author did not attempt measurement of the potential protection situation but restricted his attention to the actual domains in which processes run.

It is inevitable that the protection mechanisms available on a computer system will influence the design of software for that system, just as the facilities available in a particular programming language will affect the way programs are written in that language. Measuring protection in the context of a particular application of a protection system implies that one is not trying to assess separately the influence the protection system has had on its environment, that is on the structure of the operating system, the writing of users' programs, etc. The effectiveness of a protection mechanism depends on the power of the facilities provided by the mechanism: however, this latent power cannot make any impact unless it is effectively utilised in an application of the basic protection mechanism. The author believes, therefore, that it is appropriate to develop a protection measure that reflects the power of the application of the protection mechanism. Given that the purpose of providing protection in computer systems is to ensure adherence to the principle of minimum privilege, an objective that is widely accepted by those working on computer protection, a privilege measure is more appropriate than an exposure measure. A suitable measure would be of value both to

ordinary users of a computer system and to system designers.

5.3 A protection system model

The protection model used by the author is an extension of the one proposed by Wyeth[1976]:the extended model is referred to as the minimum privilege model of a protection system. A process has a repertoire of services* it can perform and the code executed and the domains in which it runs depend on which service it has in hand. There is not necessarily a one-to-one relationship between services and domains: a process may well perform several services in the same domain. The services of a process are made available to other processes as functions*. Wyeth's theoretically accessible set, A, and accessible set as defined by an implementation, A', are extended to refer to the period of time a process is running in a given domain whilst performing a particular service and are renamed the specification set, S, and the implementation set, I, respectively, with the following definitions.

> The specification set, S, corresponding to domain D and service C is made up of those objects which, according to some external specification, need to be accessible to the process while it is running in domain D during the performance of service C.
>
> The implementation set, I, as defined by a particular implementation, is the set of objects which the process can address while it is running in domain D during the performance of service C.

---

* This terminology is derived from Wulf [1975].

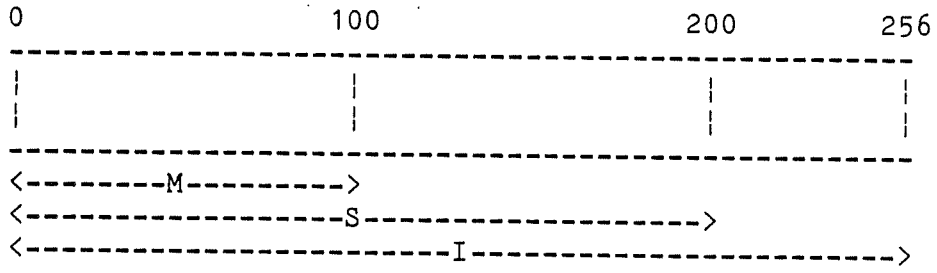
An additional set, M, the minimum privilege set, is introduced and is defined as follows.

> The minimum privilege set, M, corresponding to domain D and service C contains only those objects which must be accessible to the process while it is running in domain D to enable it to perform service C.

Unless the external specification which defines the specification set, S, is wrong (i.e. is insufficient to enable the process to do its job), the sets S and M obey the relationship $M \subseteq S$. Similarly, $M \subseteq I$. The implementation is based on the specification and will ensure that $S \subseteq I$.

As an example consider a simple program which runs in one domain only and has only a single task in its repertoire, namely doing some calculation the details of which need not concern us. To keep the illustration simple we deal only with access to words of memory used as workspace and select one word of memory as our basic object. The specification of the program states that 200 words of workspace are required. Thus, $|S| = 200$. The program is to be run on a computer where workspace can only be allocated in multiples of 128 words. The person implementing the program, therefore, allocates 256 words as workspace making $|I| = 256$. Inspection of the program's code reveals that in fact only 100 words of workspace are needed: $|M| = 100$. Perhaps the programmer made a late improvement to the program and forgot to amend the specification or perhaps he simply miscalculated the workspace requirements. In summary, then, the members of the specification, implementation

and minimum privilege sets are words of workspace memory and the sizes of the sets are |S| = 200, |I| = 256 and |M| = 100 which can be represented pictorially as follows:-



Wyeth mentions that his model can be extended to refer to (object,access right) pairs rather than simply to objects and the minimum privilege model is based on this extended version of Wyeth's.

As already mentioned the services of a process are made available to other processes as functions. However, the modules into which the system is decomposed may not all be processes so the notion of function is made more general:

> the services provided by a module are made available to other modules as functions.

Thus, a function is a (module,service) pair. The module providing a service need not be a software module, as has been implied so far. The notion of a function as just defined is equally applicable to the services, such as peripheral operations, provided by hardware modules. It is interesting to note the distinction that for access to a segment (i.e. to virtual memory) the object is the segment with the access right being some combination of

read, write and execute, whereas for access to physical store the object is a function, for example (store logic,read word), corresponding to one of the services provided by a hardware module and the access right is the right to invoke the function.

5.3.1 A protection measure

The protection model is intended to provide a framework for the formulation of protection measures which are of practical value. To measure the potential for unauthorised access the equivalent of Wyeth's benefit of protection measure ($|S|/|I|$) is appropriate (see Section 5.2.2). However, systems can be designed for which $S = I$ and the benefit measure has the ideal value of unity. For example, the protection mechanisms of the CAP are powerful enough to enable the ideal benefit measure to be reached as far as access to data in memory is concerned, unless the unit used for the specification of S is smaller than the CAP's unit of protection (one word of store). This does not mean that these systems provide perfect protection because the specification set, and thus the implementation set, may contain more rights than are strictly necessary for the process to do the job it has in hand.

A measure which indicates how much the implementation deviates from minimum privilege is the degree of overprivilege which is defined as follows.

The degree of overprivilege, d, of a process running in domain D during the performance of service C is $1 - |M|/|I|$.

The degree of overprivilege has the value 0 if there is no overprivilege, and an upper bound of 1 as the privilege excess

increases. For the example given in Section 5.3 the degree of overprivilege is 1 - 100/256 = 0.61.

The degree of overprivilege is defined in terms of the accessibility of objects to a process and really measures the lack of minimisation of access not of privilege. Overprivilege implies too much access but excessive access does not necessarily imply overprivilege: it depends to which objects the unnecessary access is allowed. If a process has, quite legally, acquired the right to call a cosine routine, say, it is not very important if the process is also (unnecessarily) given the right to call a sine routine. The process has more access than it needs but it is arguable that this extra access does not carry with it any additional privilege: the sine could in any case be calculated from the cosine. However, if the process's legal possession of the right to suspend itself pending the receipt of a message also carries with it the right to stop the entire system then the situation is rather different. The simplification of basing the degree of overprivilege on the accessibility of objects is adopted because the number of objects accessible to a process is something that can be counted. Determining how much privilege, if any, is associated with the accessibility of a particular object is a difficult problem which is considered later in this thesis (see Section 6.1.1).

The ratio |M|/|I| is similar to Jones's accuracy measure [Jones 1973] (see Section 5.2.2). It differs in the definition of the numerator being in terms of the objects which must be accessible

rather than the rights actually exercised. The degree of overprivilege varies with the size and structure of the domain in which the process is executing. Analysis of the set difference I - M may provide insight into ways of restructuring the system and modifying the implementation which will lower the degree of overprivilege. The value of |I| provides an indication of how trusted is the domain in which the process is running.

5.3.2 Protection measurement applied to functions

The protection model and the degree of overprivilege protection measure are defined in terms of the objects accessible to a process. It is important to note that functions are considered to be objects, the only access right applicable being the right to call the function. An original feature of the work described in this thesis is the attention paid to functions in the context of protection in computer systems. The author believes that it is important for software designers to pay heed to functions as well as data when considering the protection implications of their designs. The following example illustrates how the protection measure can be applied to functions.

Consider a simple system in which jobs are read from a card reader and written to a disc file, and results are read from a disc file and printed on a line printer. The time of output is also printed. To simplify the example we disregard the processing of the jobs to produce the results and only look at the overprivilege of the process(es) which provide the input and output services. We also assume that the implementation exactly

matches the design specification and that each process runs in one domain which is distinct from the domain of any other process. Three designs are considered and the example shows that a process may be overprivileged if it performs more than one service from the same domain or if it has access to all the functions provided by another process but need only have access to some of them. In the first design there are five processes:

CLOCK - interface to the hardware clock,

DISC - interface to the disc drive,

PRINTER - interface to the line printer,

READER - interface to the card reader,

SPOOL - input and output.

The functions made available by these processes are (CLOCK,read time), (DISC,read from disc), (DISC,write to disc), (PRINTER,print character), (READER,read card), (SPOOL,input) and (SPOOL,output). To perform its input service the SPOOL process needs to call the functions (DISC,write to disc) and (READER,read card). For its output service it needs the functions (CLOCK,read time), (DISC,read from disc) and (PRINTER,print character). The SPOOL process runs in the same domain for both of its services. Thus the implementation sets for the two services are identical: each contains all five functions made available by the other processes. The protection state is as follows:

(a) SPOOL process, input service.

Implementation set	Minimum privilege set
(CLOCK,read time)	(DISC,write to disc)
(DISC,read from disc)	(READER,read card)
(DISC,write to disc)	
(PRINTER,print character)	
(READER,read card)	

Degree of overprivilege = 1 - |M|/|I| = 1 - 2/5 = 0.60

(b) SPOOL process, output service.

Implementation set	Minimum privilege set
as for input	(CLOCK,read time)
	(DISC,read from disc)
	(PRINTER,print character)

Degree of overprivilege = 1 - 3/5 = 0.40

In the second design the SPOOL process is split into two processes, SPOOLIN and SPOOLOUT, which perform the input and output services respectively. SPOOLIN has access to the services of the DISC and READER processes, and SPOOLOUT to those of the CLOCK, DISC and PRINTER processes. The protection state for this design is:

(a) SPOOLIN process, input service.

Implementation set	Minimum privilege set
(DISC,read from disc)	(DISC,write to disc)
(DISC,write to disc)	(READER,read card)
(READER,read card)	

Degree of overprivilege = 1 - 2/3 = 0.33

(b) SPOOLOUT process, output service.

Implementation set	Minimum privilege set
(CLOCK,read time)	(CLOCK,read time)
(DISC,read from disc)	(DISC,read from disc)
(DISC,write to disc)	(PRINTER,print character)
(PRINTER,print character)	

Degree of overprivilege = 1 - 3/4 = 0.25

The second design is clearly better than the first from the protection point of view. The third design requires a mechanism which only allows a process access to a subset of the functions provided by another process. SPOOLIN would then have access to the (DISC,write to disc) function but not to (DISC,read from disc), and vice versa for SPOOLOUT. With this mechanism the degree of overprivilege is zero for both SPOOLIN and SPOOLOUT.

The designer now has quantitative information about the protection implications of his three designs: this should help him to choose between them.

Chapter 6

6. Measuring protection: the CAP Operating System

6.1 Audit experiment

To demonstrate that the protection model described in Chapter 5 provides a basis for the measurement of protection in practice the use of protection in the CAP Operating System [Herbert 1978(c)] was investigated. A static analysis was done in which, rather than consider all the objects in the system, only functions were dealt with. It seems likely that, for practical purposes, it will be more convenient and useful to consider subsets of the protection model than to deal with all objects at once, a subset being selected on the basis of some categorisation of the objects in the system. The results of the analysis, which will be referred to subsequently as the audit, showed how, by making changes in the protection mechanisms provided and in the way they are used in the CAP Operating System, a minimum privilege situation could be approached more closely.

Each CAP Operating System process has a number of services in its repertoire and a request to activate a process specifies which of its services the process is being asked to perform. The service of initialisation (setting up data structures and message channels, etc.) is implicitly requested when a process is created. In the course of performing a given service the process will run in a number of protection domains and in each domain a different set of objects will be accessible to the process. The process

will, in general, run in a particular domain during the performance of several different services. The set of objects accessible in a given domain to the process performing the specified service is the implementation set for that (process,service,domain) triple. The minimum privilege set for a (process,service,domain) triple is derived similarly.

In the audit each process in the CAP Operating System was considered in turn. By listing the functions accessible in each domain in which a process ran while performing a given service the implementation sets for all the (process,service,domain) triples were obtained. The corresponding minimum privilege sets were obtained in a similar way. It should be noted that the process may switch to a particular domain more than once in performing a given service and that the implementation (or minimum privilege) set is the union of the sets for each time the process is running in that domain.

An object is accessible in a given domain if the process has a capability which gives it the right to access the object. As mentioned earlier, the only objects considered in the audit were functions. In the CAP Operating System services are provided by processes, protected procedures and peripheral devices. The functions corresponding to a process's repertoire of services are accessible to another process provided the latter has a capability (a message-channel capability) which enables it to send a message to the former. The functions corresponding to a protected procedure's repertoire of services are made accessible by the possession

of a capability (an ENTER capability) enabling the process to call that procedure. The functions corresponding to the services a peripheral device can be called upon to perform are made accessible by the possession of a P-store capability for that device.

In the CAP system a process often switches to a privileged domain (by entering a protected procedure) to manufacture rights for use in the domain to which it subsequently returns. The called protected procedure will have rights which it will, in general, not exercise. The definitions of the sets I and M are such that the objects made accessible by the possession of these rights, which the called protected procedure must have in order to do its job, are included in M as well as in I.

The audit took the author about six months to complete; details of what was done are given in Appendix A. Almost the entire operating system is written in Algol 68C [Bourne 1975]. If a readable high-level language had not been used, doing the audit would have been an Herculean task indeed!

The audit involved detailed study of the structure and programs of the operating system and had the valuable side effect of exposing a number of weaknesses in the design and implementation, some of which had not been fully appreciated previously. These are detailed in Appendix B.

6.1.1 Audit: Part 1

Initially the audit results were processed on the basis of all functions being of equal importance. As mentioned earlier the

problem of attempting to reflect realistically the relative values/importance of the objects in a system has received very little attention in previous work concerned with measuring protection [Jones 1973, Ellis 1974, Wyeth 1976]. It is not an easy problem to resolve. In general the value of an object is based on subjective judgement and it would probably be impossible to justify any particular set of weights applied in an endeavour to take account of the relative importance of the objects contributing to a measure of protection. Besides, the application of weights implies the calculation of a single number to indicate how good or bad is the protection provided in the system under investigation: such a number would not be of any practical value to the designers or users of the system. On the other hand, to ignore the value weighting problem completely will also render ones results of little use in practice. The solution adopted was to divide the functions into a limited number of categories and to look at the degree of overprivilege within each category. In this way the differences in importance of the objects in the system could be handled without attempting to associate an individual weighting factor with each object. The prime motivation was to isolate those functions whose invocation has protection implications. Each function was allocated to category P, U, V, or W on the basis of the following criteria.

P: functions whose invalid use has breach of protection implications (i.e. has an effect outside the calling protection domain; creates, modifies or deletes capabilities).

U: functions whose invalid use consumes system resources.

V: functions whose invalid use has security implications in the sense of allowing the caller to get hold of information to which he is not entitled.

W: functions which do not fall into category P, U or V; the prevention of their invalid use helps to speed up the locating of bugs.

Invalid use of a function means invocation of the function when it need not even be accessible. The categorisation of the functions is given in Appendix C. It should be noted that the way the CAP Operating System has been designed means that there is no clear boundary between functions that are considered part of the operating system and those which are not [Needham 1977(a)]. When a user logs in he is allocated a USER process which, after doing some initialisation work, enters the command program's protected procedure. The user can then communicate direct with this program. For the purpose of the audit a line had to be drawn somewhere and it was decided to take entry into the command program as defining exit from the operating system. The USER process service run-user-process was, therefore, the outermost service included in the audit.

A special protected procedure called TESTER exists to enable repair work to be carried out on the operating system if required. As its name implies it was originally used for testing the operating system during the earlier stages of its development. TESTER is all-powerful but can only be entered during system

start-up. It has the rights to invoke any of the functions in the system and can be given commands to make it do so. The value of |I| for TESTER and for all domains entered as a result of running TESTER is, therefore, very large although the degree of overprivilege is 0. Because of TESTER's special nature the RESTART process's run-TESTER service was excluded from the audit.

The audit results were re-processed for each category of function separately. Having calculated the implementation and minimum privilege sets, I and M, for all functions as described earlier, these sets were then divided into separate implementation and minimum privilege sets for P, U, V and W category functions. These subsets of I and M are designated I_p, and so on. The degree of overprivilege calculated from I_p and M_p is designated d_p, and similarly for d_u, d_v and d_w. These degrees of overprivilege were computed: the complete results of the audit are given in Appendix E. Since the use of protected procedures is central to the design of the CAP Operating System [Needham 1977(a),(c)], the degree of overprivilege for each of the domains corresponding to the protected procedures which make up the operating system is of special interest. To present the results in a compact way the average values of d were computed for these domains, the average being over the process services in the performance of which that domain is entered. In Algol 68-like language the algorithm is:-

```
FOR each domain
DO INT n:=0,REAL sum:=0;
```

```
    FOR each process
    DO FOR each service
      DO
        sum +:= degree of overprivilege[process,service,domain];
        n +:= 1
      OD
    OD;
    average degree of overprivilege[domain] := sum/n
  OD
```

Averages were also computed for d_p and $|I_p|$ on the same basis. It should be noted that the number of results from which the average was computed varied considerably, the range being from 1 to 68, and that the average was computed from all the results which had been collected (see Appendix E).

It may be instructive to illustrate the procedure by working through in detail how the averages for d_p and $|I_p|$ were calculated for the MAKEENTER* domain. The MAKEENTER protected procedure, which manufactures ENTER capabilities and capability segments, is called only during the performance of the initialise services of the USER, LPRDESPOOL and PTPDESPOOL processes and the USER process's run-user-process service. (The LPRDESPOOL and PTPDESPOOL processes are used for spooling output to the line printer and paper tape punch respectively: USER processes run users' computa-

---

* A brief description of the processes and protected procedures of the CAP Operating System can be found in the Glossary. Full details are given in the CAP Operating System Manual [Herbert 1978(c)].

tions.) The first step is to identify the members of the implementation and minimum privilege sets for the four (process, service, domain) triples (USER, initialise, MAKEENTER), (USER, run-user-process, MAKEENTER), (LPRDESPOOL, initialise, MAKEENTER) and (PTPDESPOOL, initialise, MAKEENTER). During the initialisation of the USER process an ENTER capability for the MFD (Master File Directory) is added to the MAIN domain, the domain in which the process runs first of all, by the following sequence of protected procedure ENTERs and RETURNs:-

Current domain	Domain switched to	Service requested
MAIN	SINMAN	sin-of-MFD
SINMAN	MAIN	(RETURN)
MAIN	SINMAN	cap-from-sin
SINMAN	MAKEENTER	make-enter
MAKEENTER	SINMAN	(RETURN)
SINMAN	MAIN	(RETURN)

The MAKEENTER domain here contains ENTER capabilities for the SETUP, ECPROC, STOREMAN, MAKEENTER, IOC and FAULT protected procedures in its G capability segment*, put there at system generation, and an ENTER capability for the MFD, which it has just manufactured, in its A capability segment. It also has three software capabilities for presentation to the ECPROC protected procedure: system-stop-permission, capability-permission and info-permission. The only P category functions made accessible by the possession of these ENTER and software capabilities are those corresponding

---

* Capability segment 0, one of those which contributes to all domains in which the process runs (see Section 3.4), is known as the G (global) capability segment.

to ECPROC's handle-blunders, resolve-VM-faults, stop-system, create-PRL-entry, create-capability, update-capability and delete-capability services. These functions, therefore, make up the implementation set for the (USER, initialise, MAKEENTER) triple. Examination of the code of the MAKEENTER program shows that, when performing its make-enter service, it needs to be able to access the functions corresponding to ECPROC's handle-blunders (in case of a programming error), resolve-VM-faults (in case a virtual memory fault occurs), stop-system (in case a blunder causes its run time error procedure to be called) and create-PRL-entry services. These functions make up the minimum privilege set.

The implementation and minimum privilege sets for the other triples are derived in a similar way. Note that for the LPRDESPOOL and PTPDESPOOL processes' initialise services the MAKEENTER domain is entered thrice, once to make up an ENTER capability for the MFD, as described above, once as a consequence of the MFD being entered to retrieve an ENTER capability for the spool directory, and once as a consequence of IOC being entered with a request for an interactive stream protected procedure (ISPP). In the latter two cases the sequence of ENTERs and RETURNs is:

Current domain	Domain switched to	Service requested
MAIN	DIRMAN (MFD)	retrieve (spool directory)
DIRMAN (MFD)	SINMAN	cap-from-sin
SINMAN	MAKEENTER	make-enter
MAKEENTER	SINMAN	(RETURN)
SINMAN	DIRMAN (MFD)	(RETURN)
DIRMAN (MFD)	MAIN	(RETURN)
MAIN	IOC	request-interactive-stream
IOC	MAKEENTER	make-enter
MAKEENTER	IOC	(RETURN)
IOC	MAIN	(RETURN)

The implementation and minimum privilege sets are the unions of the sets for the three entries.

It is worth noting that following the request to IOC for an interactive stream, MAKEENTER manufactures an ENTER capability for the stream.The possession of this ENTER capability, which is put in MAKEENTER's A capability segment, makes three P category functions accessible in the MAKEENTER domain. These functions are, therefore, included in the implementation set. The ISPP ENTER capability is not actually exercised in the MAKEENTER domain but, as its creation is an essential part of the make-enter service (indeed, it is the very purpose of the service), the functions corresponding to the three ISPP services are included in the minimum privilege set too.

The degree of overprivilege can now be calculated for each triple. The position is summarised in Table 6.1 (the sets for PTPDESPOOL are identical to those for LPRDESPOOL).

Process	Service	I_p	M_p	d_p
USER	initialise	ECPROC: handle-blunders resolve-VM-faults stop-system create-PRL-entry create-capability update-capability delete-capability	ECPROC: handle-blunders resolve-VM-faults stop-system create-PRL-entry	.43
	run-user-process	ECPROC: handle-blunders resolve-VM-faults stop-system create-PRL-entry create-capability update-capability delete-capability o/p ISPP(tty): write-buffer close reset-state i/p ISPP(tty): read-buffer close reset-state	ECPROC: handle-blunders resolve-VM-faults stop-system create-PRL-entry	.69
LPRDESPOOL	initialise	ECPROC: handle-blunders resolve-VM-faults stop-system create-PRL-entry create-capability update-capability delete-capability o/p ISPP(printer): write-buffer close reset-state	ECPROC: handle-blunders resolve-VM-faults stop-system create-PRL-entry o/p ISPP(printer): write-buffer close reset-state	.30

Table 6.1

The averages for d_p and $|I_p|$ can now be calculated for the MAKEENTER domain (see Table 6.2).

Process	Service	d_p	$\|I_p\|$
USER	initialise	.43	7
	run-user-process	.69	13
LPRDESPOOL	initialise	.30	10
PTPDESPOOL	initialise	.30	10
average		.43	10.0

Table 6.2

The results for all the domains are given in Table 6.3.

DOMAIN	$\bar{d}$	$\bar{\bar{d}}_p$	$\|I_p\|$
DIRMAN	.81	.59	15.8
ECPROC	.51	.37	15.9
IOC	.77	.59	15.4
ISPP	.84	.17	4.0
LINKER	.84	.69	16.0
LOGON	.83	.13	8.0
MAIN	.74	.56	12.3
MAKEENTER	.67	.43	10.0
PRLGARB	.79	.40	8.4
SETUP	.45	.29	15.0
SINMAN	.79	.72	26.5
STOREMAN	.80	.74	25.5

Table 6.3

The figures in Table 6.3 show that more care was taken to control the accessibility of P functions than for the functions as a whole (for all domains $\bar{d} > \bar{\bar{d}}_p$). Overall the degree of overprivilege figures show that even for the P functions, the CAP Operating System as it was at the time of the audit fell far short of a state of minimum privilege as far as the accessibility of functions is

concerned. In addition to computing the average degrees of overprivilege the set difference $I_p - M_p$ was analysed. The number of (process,service,domain) triples for which each P category function was accessible was counted and the percentage of these for which the function need not have been accessible was calculated. These frequency counts are given in Appendix D. From this analysis it was clear that a number of steps could be taken to improve the degree of overprivilege figures. The actions which would have most impact were:-

1. By the use of software capabilities or access bits in ENTER capabilities restrict the functions made accessible by the possession of an ENTER capability for a protected procedure to the subset of that procedure's repertoire which needs to be accessible in the calling domain. This was already done with ECPROC by insisting that requests for most of ECPROC's services be accompanied by the presentation of a software capability proving that the caller was authorised to request that service. An alternative mechanism would make use of access bits in the ENTER capability itself. This mechanism had not yet been implemented when the audit was done. The called protected procedure is able to examine the software capability or the access bits presented before deciding whether or not to perform the service asked for.
2. Restrict the functions made accessible by the possession of a message-channel capability for a process to the subset of that process's repertoire which needs to be accessible in the

calling domain. There is enough space in channel-access and message-channel capabilities to enable function access bits to be implemented quite easily in a manner analagous to the implementation of access bits in ENTER capabilities. Alternatively, a mechanism could be provided based on the use of software capabilities, but this seems likely to be more cumbersome to implement.

3. When a process no longer needs to have access to a given function in a given domain then the right which makes that function accessible in that domain should be destroyed. For example, the processes which drive the peripheral devices claim the appropriate device during their initialisation: thereafter the ability to reserve a device, by calling the (ECPROC, claim-device) function, is no longer required and the right whose possession makes it accessible, the peripheral-permission software capability, can be destroyed. This can be done by using the MOVECAP instruction to overwrite the capability with an invalid one. Where the function is made accessible by the appropriate access bit being set in an ENTER or message-channel capability that bit can be unset by means of the REFINE instruction.

6.1.2 Audit: Part 2

To demonstrate the effect of making these changes the audit results were reworked on the assumption that the above actions had been taken. Only the P category functions were considered, these being the most important as far as the integrity of the system is

concerned; the revised figures for $\overline{d}_p$ and $|\overline{I}|_p$ are given in Table 6.4.

DOMAIN	$\overline{d}_p$	$\overline{\lvert I \rvert}_p$
DIRMAN	.36	10.0
ECPROC	.37	15.9
IOC	.25	8.4
ISPP	.17	4.0
LINKER	.55	11.0
LOGON	.00	7.0
MAIN	.15	5.5
MAKEENTER	.28	8.0
PRLGARB	.03	5.4
SETUP	.01	12.0
SINMAN	.39	12.1
STOREMAN	.33	9.8

Table 6.4

They show that for all domains except ECPROC and ISPP there has been a large reduction in the average degree of overprivilege and that the specification sets are smaller on average, markedly so in several cases. The reason for no change in the case of ISPP is that the privilege excess is entirely accounted for by the ENTER capabilities for the USER process's input and output streams being held in the G capability segment in order to make them globally accessible in the USER process. This point is discussed again later (see Section 6.1.5).

The other exception, ECPROC, shows no change because its overprivilege is the result of its unique authority to enter the Master Coordinator requesting services (other than wait-event which can be called from any domain). The Master Coordinator could

have insisted that ECPROC present a software capability when requesting a service and steps could then have been taken to reduce the degree of overprivilege figures for ECPROC. However, one of the reasons for having ECPROC is to avoid the overheads which would be incurred by the Master Coordinator checking software capabilities: ECPROC has the responsibility of carrying out checks on behalf of the Master Coordinator and may be treated as if it were part of the Master Coordinator.

The frequency counts for the P category functions were also recalculated (see Appendix D) and these too showed a marked improvement.

6.1.3 Audit: Part 3

The figures in Table 6.4 were based on each protected procedure having the rights it needed to be able to perform all of the services in its repertoire. Thus the capability segments referred to in the ENTER capability for the protected procedure held the same capabilities irrespective of the domain in which that ENTER capability would be used. But, in general, a process running in a given domain will only request a protected procedure called from that domain to perform a subset of its full repertoire. The rights built into that ENTER capability for the called protected procedure can, therefore, be pruned to exclude those rights only used in the performance of services which will never be requested from that domain.

An example may help to clarify the point. The MAKEENTER protected procedure has two services in its repertoire, make-enter

and make-cap-seg. The former constructs an ENTER capability and the latter makes up a capability segment. To perform make-enter the (ECPROC, create-PRL-entry) function is called but the (ECPROC, update-capability) function is not used, whereas to perform make-cap-seg the update-capability service is needed but the create-PRL-entry service is not. For a domain from which only the make-cap-seg service will be requested the ENTER capability for the MAKEENTER protected procedure can be such that the (ECPROC, update-capability) function is accessible in the MAKEENTER domain but the (ECPROC,create-PRL-entry) function is not.

The effect of restricting the rights built into ENTER capabilities is shown by the figures in Table 6.5 and by the corresponding frequency count figures in Appendix D.

DOMAIN	$\overline{d}_p$	$\overline{\lvert I \rvert}_p$
DIRMAN	.07	7.0
ECPROC	.37	15.9
IOC	.02	6.6
ISPP	.17	4.0
LINKER	.55	11.0
LOGON	.00	7.0
MAIN	.15	5.5
MAKEENTER	.15	7.0
PRLGARB	.03	5.4
SETUP	.01	12.0
SINMAN	.03	8.1
STOREMAN	.05	7.2

Table 6.5

There are further reductions from the figures in Table 6.4 in the average degree of overprivilege and in $\overline{\lvert I \rvert}_p$ for the SINMAN, STOREMAN,

IOC, MAKEENTER and DIRMAN domains. Overall the results in Table 6.5 show that the system is now getting quite close to a state of minimum privilege.

6.1.4 Audit: Part 4

The remaining overprivilege is mostly accounted for by the default input and output streams for user programs being globally available in the USER process by having their stream ENTER capabilities in the G capability segment. If this had not been done the figures would have been as given in Table 6.6 with all domains except ECPROC, IOC and MAIN having $\overline{d}_p = 0$.

DOMAIN	$\overline{d}_p$	$\overline{\lvert I \rvert}_p$
DIRMAN	.00	6.3
ECPROC	.36	15.8
IOC	.02	6.6
ISPP	.00	3.0
LINKER	.00	5.0
LOGON	.00	7.0
MAIN	.14	5.4
MAKEENTER	.00	5.5
PRLGARB	.00	5.0
SETUP	.00	11.8
SINMAN	.00	7.7
STOREMAN	.00	6.6

Table 6.6

The special nature of ECPROC has already been discussed (see Section 6.1.2). The reason for IOC's average degree of overprivilege being non-zero is that the message channels to the device

interface processes and to the despoolers are not set up until they are needed. This is to avoid setting up message channels which will not be used. For most processes and protected procedures that require message channels these are set up as part of the process's or procedure's initialisation and the right to set up message channels can then be destroyed. For IOC the right to set up send-type message channels cannot be disposed of in this way as it is needed for IOC to be able to perform its despool service. When IOC is entered from a given domain with a despool request on one call and a request for one of its other services on another call, the right to set up send-type message channels is accessible in the IOC domain for both calls but is only needed for the despool request. This situation arises in the MAIN domain of the USER process.

All the domains except MAIN included in Tables 6.3 to 6.6 correspond to protected procedures of the same name. For each process the MAIN domain is the domain in which the process runs initially when it is created. A process running in its MAIN domain will, in general, switch to another domain by executing an ENTER instruction and will switch back to its MAIN domain by executing the matching RETURN instruction. A cautionary note is in order regarding the interpretation of the averaged figures for MAIN domains. These figures demonstrate the protection implications of the program structure adopted for the CAP Operating System processes, but it must be remembered that the individual domains contributing to the average range from that of a process such as

Real Store Manager (RSM) which is responsible for the integrity of vital system data structures to one like CLOCK which provides a simple service. The program structure used for processes is [Needham 1977(a)]:-

```
BEGIN
    initialisation code #setting up message channels, claiming
        peripheral devices, etc.#
END
DO   #to infinity#
    WHILE messages(input) = 0 DO wait event OD;
    receive message with reply(a,b,c,d);
    CASE a IN
        services offered
    ESAC;
    return reply(p,q,r,s)
OD
```

The average degree of overprivilege for the MAIN domains could be reduced by encapsulating each of the services in a process's repertoire in its own protected procedure.* However, this would mean separating into different programs services which are logically connected (and may use common subroutines) and, as the average degree of overprivilege of .14 is already reasonably low, it is probably not worth the candle. It is worth noting that if it were possible to restrict a process's rights according to the domain from which it is called, as was done for protected proced-

---

* If this were done the average degree of overprivilege for the IOC domain would be reduced to .00 in Table 6.6.

ures in calculating the figures for Table 6.5, there would be no need to modify the processes' internal structure. However, this cannot be done with the current CAP system.

6.1.5 Audit: Part 5

The reason the default input and output streams were made globally available in the USER process was that it was considered essential to be able to guarantee that any user program would be able to output a message without its having to rely on being passed an output stream ENTER capability as an argument. To do this it is only necessary to make the function corresponding to the output stream's write-buffer service globally available, for example by the appropriate setting of access bits in an ENTER capability for an output stream. Any additional input and output facilities required by the user program could be made available by passing it appropriately restricted ENTER capabilities for input and output streams. (There is no reason why there should not be in a domain more than one ENTER capability for a particular protected procedure.) As already mentioned, access bits in ENTER capabilities had not been implemented when the audit was done. Had only the function corresponding to the output stream's write-buffer service been made globally accessible in the USER process the figures would have been as in Table 6.7. This represents probably the closest the CAP Operating System could come to minimum privilege as far as functions are concerned without the minimisation of privilege being at the expense of other design considerations such as the grouping together of logically

related services in the same program. It is noteworthy that this considerable improvement in protection would be obtained without any modification of the operating system's structure and would incur very little run-time overhead, although the system programs would be slightly larger.

DOMAIN	d	I
	p	p
DIRMAN	.02	6.4
ECPROC	.36	15.8
IOC	.02	6.6
ISPP	.04	3.2
LINKER	.17	6.0
LOGON	.00	7.0
MAIN	.14	5.4
MAKEENTER	.05	5.8
PRLGARB	.01	5.1
SETUP	.00	11.8
SINMAN	.01	7.8
STOREMAN	.02	6.7

Table 6.7

6.1.6 Audit: Part 6

So far protection measurement has been used to study the application of the CAP's protection mechanisms in the CAP Operating System as a whole. A major use of protected procedures in the operating system is in the role of gate-keeper [Needham 1977(c)]. The protection measures enable a comparison to be made, from the protection point of view, of the system with and without gate-keeping. For example, in the PRINTER process the STOREMAN protected procedure acts as a gate-keeper to the SINMAN protected procedure and to the Real Store Manager (RSM) and Virtual Store

Manager (VSM) processes. The results used to produce Table 6.6 were reworked for the PRINTER process on the assumption that STOREMAN was not used: the effect is shown in Table 6.8. The results clearly demonstrate the advantage of using a gate-keeper.

	with STOREMAN		without STOREMAN	
Service	d / p	I / P	d / p	I / p
initialise PRINTER	0.17	6.0	0.57	14.0
print line	0.00	4.0	0.64	11.0
print document	0.00	4.0	0.09	11.0
average	0.06	4.7	0.43	12.0

Table 6.8

There is an additional consideration which is not reflected in these figures. The SINMAN protected procedure trusts its callers to pass it sensible arguments and it is only reasonable to do this if there are very few domains from which SINMAN can be entered.

6.2 Protection measures and operating system structure

The experiments done with the CAP Operating System have demonstrated that protection measurement is a useful tool for operating system design work. However, an operating system designer has to take into account not only protection considerations, which can now be quantified, but also such things as the complexity of the system,its modularity, the logical relationships between different items in it, and its efficiency and effectiveness. When analysing the protection aspects of an operating system

we can draw a distinction between, on the one hand, the protection mechanisms being inadequate and, on the other, a failure to make full use of them. The CAP protection mechanisms have been shown to be sufficiently powerful, with a few relatively minor modifications, to enable an operating system to be written in which a minimum privilege position is approached very closely. The results presented earlier in this chapter show that these mechanisms were not being fully utilised in the CAP Operating System at the time of the audit.

The basic design objective to be met by the CAP Operating System was that the principle of minimum privilege should be applied rigorously in relation to access to memory [Needham 1977(a)]. In determining the structure of the operating system the principle that was applied in practice was that each system data structure should be accessible to only one protected procedure which would, therefore, have the exclusive responsibility for managing that data structure. This requirement is unnecessarily restrictive and it is sufficient to ensure that the responsibility for managing a system data structure rests solely with a single program, the distinction being that the protected procedure defines the protection domain which may be entered by a process and the program is that component of the protection domain in which the process will run. Relaxing the one-protected-procedure-per-data-structure constraint and tailoring the rights bound into the ENTER capability for a protected procedure to match the requirements of the domain in which that capability will be used helps to minimise

privilege, as has been shown in Section 6.1. Ideally the ENTER capability should be specific not just to the domain but to the performance of one particular service in the domain. However, the grouping into one domain of logically related services is justified on structural grounds and it is then convenient and sensible to unite the ENTER capabilities required by the various services.

This idea of domain-specific ENTER capabilities can be extended to apply to process activation. Each activation of a process would then cause the same program to be executed but the capability segments accessible to the process, and the rights contained in them, would not necessarily be the same for each activation. It would not be possible to implement these domain-specific process activation capabilities with the Normal Mode Interpreter CAP microprogram, but the alternative being developed by Herbert [1978(d)] would be able to support such a facility.

The experiments described in Section 6.1 used as a protection measure the degree of overprivilege, d, evaluated for functions only. Complete containment (d = 0) could be achieved by modifying the structure of the CAP Operating System processes and by applying three different techniques, or a combination of them, to restrict the functions made accessible by the possession of an ENTER capability. In the CAP Operating System a receive-message capability plays a dual role: it is used (a) to check whether there are any messages ready to be received, and (b) as a software capability authorising the receipt of messages on that channel.

If two separate (say, poll-channel and receive-message) capabilities were used instead then processes could be restructured with a separate channel for each service provided by the process and the main program simply polling these channels. On detecting a message on a channel the main program would enter the appropriate protected procedure to receive the message and provide the service requested. The two main objections to this restructuring are (a) that it runs contrary to the structural argument for the grouping of logically related services in the same program, and (b) that it is likely to make the operating system structure more complex.* If domain-specific process activation capabilities were available such restructuring would not be needed.

The three techniques which can be applied to minimise the privilege associated with the possession of an ENTER capability are:-

1. providing a separate protected procedure for each service so that each ENTER capability only imparts the right to request a single service. It is worth noting that this would not result in any increase in the number of domain switches which take place during the running of the system. However, the hardware was not designed to cope with so many protected procedures and a run-time penalty would be paid in degraded hardware performance. The other objections to this technique are those

---

* The author feels that, just as a monolithic operating system structure is likely to be beyond the limitations of human understanding, excessive fragmentation can lead to a system becoming too complex to understand. Up to a point dividing the system into modules reduces its complexity but eventually a diminishing returns position is reached.

which apply to the restructuring of processes into several protected procedures.

2. setting access bits in the ENTER capability to authorise the use of only a subset of the protected procedure's repertoire of services.

3. issuing software capabilities covering the individual services in a protected procedure's repertoire. A software capability can have additional information built into it. For example, the peripheral-permission software capability could incorporate the identity of the device(s) for which it was valid.

Any one of these techniques would be sufficient for total containment but there are advantages in allowing the operating system designer some flexibility of choice [Needham 1977(a)].

Chapter 7

7. Cost of protection - CAP Operating System

7.1 Introduction

The audit experiments described in Chapter 6 have demonstrated that the CAP computer's protection mechanisms are sufficiently powerful to enable the CAP Operating System to come very close to a state of minimum privilege. The present chapter is concerned with trying to establish what it costs to provide this protection. The cost model used has already been described in Section 5.1. Because much use is made of protected procedures in the CAP Operating System the cost component for switching from one protection domain to another is of particular interest.

It is common for commercially available computers to be 2-state machines, operating in one state when running ordinary programs and in the other when running so-called privileged programs. In IBM terminology the two states are problem state and supervisor state respectively [IBM 1970]. When the computer is in supervisor state it runs without protection. Typically, when an ordinary program calls for one of the services provided by the operating system, the computer switches to supervisor state, executes the code to perform the desired service, and then reverts to problem state. Thus, the operating system runs in the same protection domain irrespective of which service it is performing. There is no protection while the computer is in supervisor state so the operating system's privilege is maximised rather than minimised:

this is in marked contrast to the situation with the CAP Operating System. The techniques for measuring protection presented in this thesis could be applied to provide a quantitative basis for comparing protection in the operating system of the CAP with that of a 2-state computer.

In the 2-state computer only a single protection barrier is provided whereas on the CAP computer there are many. As a rough indication of the cost of the extra protection provided, experiments were done to count the number of protection barriers actually encountered during the running of programs on the CAP computer.

In the CAP Operating System a user requests an operating system service by entering the appropriate protected procedure. This protected procedure may, in turn, call another, and so on. Although a protection domain switch is more complicated than the switch to supervisor state in a 2-state machine, the first domain switch in response to a call for an operating system service is comparable to the change to supervisor state because each represents the first protection barrier encountered in providing the service. The differences between the protection provided in the operating systems of the CAP and a 2-state computer are that in the CAP Operating System (a) the first domain switch is not always to the same protected procedure, and (b) more than one domain switch may take place during the performance of an operating system service. In the experiments, each time an operating system service was called a count was made of the number of domain switches which

took place during the performance of that service. Results were collected from the running of two programs, the Algol 68C compiler [Bourne 1975] and the system generation program, Genesis. The CAP Operating System has been available for too short a time for any pattern of normal use to be established. Indeed, much of the work done on the CAP is still concerned with developing the operating system. These two programs were selected because (a) they are much used, (b) they are substantial programs, and (c) one, the compiler, is compute-heavy whereas the other handles very large volumes of data.

In addition to counting domain switches, the time taken to execute the instructions to enter and return from a protected procedure was determined experimentally. The length of time it takes in practice to effect the switch from one domain to another, or from one process to another, is considerably influenced by the slaving effectiveness of the Capability Unit (see Section 3.5). Experiments were done to assess how well the Capability Unit performed in this respect.

The details of the experiments are given in subsequent sections. Two features of the CAP computer facilitated the carrying out of these experiments. Firstly, the CAP has a built-in hardware counter. As already mentioned in Chapter 4, this can be set to count a wide variety of events such as the number of times a word is read from the slave stores or the number of times a selected microprogram instruction is obeyed. Further details of the hardware counter can be found in the CAP Hardware Manual [Herbert

1978(b)]. Secondly, the CAP has a programmable microprogram which can be modified for monitoring purposes. The use of hardware counters is a well-established technique for monitoring a computer's performance. As provided on the CAP it is simple and convenient to use since the selection of what is to be counted is by means of manual switches and does not involve any detailed knowledge of circuit diagrams or the like. However, it has the disadvantage that occurrences of only one event can be counted at a time. As has been mentioned before (Chapter 4) it is more unusual to use microprogramming as a monitoring tool. Nevertheless, this provides a very flexible method of studying a computer's performance.

7.2 Details of experiments

7.2.1 Counting domain switches for operating system services

As in the case of the audit (Chapter 6) a more-or-less arbitrary decision had to be made as to which protected procedures were considered to be part of the operating system. When a user logs in he is allocated one of a stock of USER processes. All the protected procedures for which that USER process had an ENTER capability as soon as the process was allocated were considered to be in the operating system. This was in line with the criterion applied in the audit. Some of the services provided by these protected procedures require the caller to present a software permission capability (see Section 3.7). Such services were excluded from what were considered to be operating system services for the purposes of this experiment. To avoid confusion, the services

classed as operating system services available in a USER process will be referred to as primary services. A list of the primary services is given in Appendix F.

The microprogram was modified to force the Master Coordinator to be entered on the successful completion of every ENTER and RETURN instruction. Code was inserted into the Master Coordinator to detect when a USER process requested a primary service and to count the number of ENTER instructions obeyed until that service had been completed, referred to as the consequential ENTERs for that primary service. The mechanism for handling faults, which may be virtual memory faults, outform ENTER faults or genuine blunders (see Appendix A), in the CAP Operating System is for the Master Coordinator to simulate the execution of an ENTER instruction to enter the FAULTPROC protected procedure. These simulated ENTERs were counted separately but were treated as if they were genuine ENTER instructions. The Master Coordinator's facilities for printing system trace information were extended to enable the monitoring results to be printed.

The CAP Operating System is a multiprogramming system and normally several USER processes would be active at the same time. Some of the modules which make up the operating system are processes not protected procedures and, in general, the operating system may be multiprogramming between several USER processes and several operating system processes. The primary services are all provided by protected procedures but the performance of a primary service may involve the sending of a message to activate an

operating system process. The sending of such messages and any domain switches which take place in the process activated by the message are clearly just as direct consequences of the request for the primary service as are the consequential domain switches within the USER process where the request originated. However, if more than one USER process is running it is extremely hard to sort out whether a particular operating system process has been activated as a result of a primary service call in, say, USER process A or USER process B. The difficulty was overcome by making sure that only one USER process was running during the experiments. This was achieved by the simple expedient of disconnecting all but one of the terminals. Multiprogramming during the experiments was, therefore, between a single USER process and any active operating system processes. In the experiments, if an operating system process was running it would, in most cases, have been activated as a direct consequence of a primary service call in the single USER process which would have suspended itself waiting for the operating system process to complete its work. Sometimes the activation of the operating system process may have been caused indirectly during the performance of a primary service, perhaps because a virtual memory fault had to be dealt with or because garbage had to be removed from a Process Resource List (PRL) which had become full. No distinction was made in the experiments between the direct and indirect activation of operating system processes. In order to minimise the incidence of garbage collection of the USER process's PRL during the experiments the garbage

collector was run immediately before each monitored run. The CLOCK process ran periodically to read the hardware clock. However, this independent running of the CLOCK process did not involve any domain switching or message sending and so could be disregarded. Thus, all domain switching and message sending which took place in any process during the performance of a primary service requested by the USER process were regarded as consequences of the call for that primary service. Not all domain switching and message sending occurred during the performance of a primary service so the total numbers of messages sent and of ENTER instructions obeyed during each run were recorded.

As already mentioned results were collected from runs of the Algol 68C compiler and the system generation program. The compilation was of the CAP Operating System Master Coordinator, whose source was 1750 lines long and produced an object code segment of 3772 words. The Genesis runs were to generate the complete CAP Operating System, the total size of the generated system being over 200000 words. With the monitoring system the running time for these programs was about one third longer than with the normal system: this was mainly because of the forced entering of the Master Coordinator on every ENTER and RETURN instruction. Under the conditions of the experiments the slower running of the system would not affect the number of domain switches nor the number of messages sent.

7.2.2 Counting messages sent

A slight modification to the monitoring code inserted in the

Master Coordinator enabled a count to be made of the messages sent as a consequence of the USER process requesting a primary service. The normal mechanism for sending a message to a process is to call the appropriate service of the ECPROC protected procedure. When dealing with a virtual memory fault the FAULTPROC protected procedure sends a message to the RSM (Real Store Manager) process. Because FAULTPROC runs in the same protection domain as ECPROC it is able to shortcut the normal message sending mechanism. Code had already been added to the Master Coordinator to count virtual memory faults so all that was necessary to obtain the number of consequential messages was to adjust the number of messages counted by adding one message for each virtual memory fault.

7.2.3 Instruction timing

To complement the counting of domain switches, the time taken to switch from one domain to another and then to revert to the first domain was measured. In the CAP Operating System all processes except the Master Coordinator are at the second level in the process hierarchy (see Section 3.3). Accordingly, a simple system was generated to enable instructions to be timed by running a process at that level. The method used was to time a program which repeated the appropriate instruction or set of instructions a certain number of times and then to repeat the timing for a different number of iterations. The time taken to obey the instruction or set of instructions once was obtained by subtraction and division. A stop-watch was used to measure the time taken. The test programs forced the CAP to enter wait state on completion of

the instruction repetitions. This condition is indicated by a characteristic note on the CAP's loudspeaker and an easily recognisable pattern of lights on the computer's front panel, both of which made clear when the watch should be stopped. By making the number of iterations large enough the use of a stop-watch is sufficiently accurate for instruction timing. The hardware counter was used to check that the appropriate instruction was being repeated the expected number of times.

The oscillator speed can be changed on the CAP by setting manual switches. In normal use the CAP runs at speed 4 and this speed was selected for all the experiments.

In Section 3.2 it was noted that the core store of the CAP has modules with very different timing characteristics and that connected between the CAP processor and the store bus there are slave stores for reading and writing. The hardware counter was used to count the number of store reads and writes which involved a read from or write to core during the running of the timing experiments. It was found that, for the instructions being timed, all store reads were from the slave store and, except for one test program, all store writes were to the slave store. The exception was the test program for measuring the time taken to enter a protected procedure with a capability passed as an argument and then to return with a capability as a result. In this case 13% of the store writes involved a write to core store. The system generation program allocates core store in the faster Plessey modules first so the core writes in this case were to the Plessey stores.

Domain switching without passing any arguments was studied first. This was done by measuring the time taken to execute an ENTER instruction plus a RETURN instruction. Starting from scratch the microprogram interpreter enters the top level process, conventionally known as the Master Coordinator. In the timing system the Master Coordinator was a very simple program which immediately entered its only sub-process. For timing the ENTER and RETURN pair of instructions this sub-process repeatedly entered a protected procedure whose program was a single instruction, namely RETURN. Having completed the specified number of repetitions the sub-process entered the Master Coordinator which immediately went into wait state. The CAP has facilities to enable capabilities to be passed from one protection domain to another. The mechanism has already been described in Section 3.4. Very briefly, what happens is that, on execution of an ENTER instruction, the N capability segment of the calling domain becomes the A capability segment of the entered domain. On RETURN this change is reversed. In the CAP Operating System many protected procedure calls involve the passing of a capability argument and/or the returning of a capability result. A further timing experiment was done to measure the time taken to switch protection domains with one capability argument and one capability result. The sequence

of actions timed was:-

Calling domain	Entered domain
1. Copy argument capability into N capability segment	
2. Enter protected procedure	
3.	Copy argument capability from A capability segment
4.	Copy result capability into A capability segment
5.	Return
6. Copy result capability from N capability segment	

Numerical arguments and results are passed from one protection domain to another in the B registers of which there are 15 (see Section 3.2). The instruction to load a register from a store location (BS) and its converse (SB) were timed. The instruction timings all had to be corrected for the inclusion of the test and count instruction (TCS) in the repeated loop. This instruction was, therefore, timed separately. The final results were expressed as multiples of the time for the load from store instruction to provide a rough yardstick for comparison with other computers.

7.2.4 Slaving effectiveness of the Capability Unit

When a protection domain or process switch takes place a number of entries in the Capability Unit become invalid and are marked as such (see Section 3.5). The capability registers occupied by

these entries are available for re-use. The microprogram endeavours to allocate capability registers in such a way that the invalidated entries may still be in the Capability Unit when they are next required. If a required entry is not found in the Unit a reset cycle is triggered to set it up (see Section 3.5). To provide an indication of the Capability Unit's effectiveness as a slaving device for domain and process switching, experiments were done to count the number of reset cycles saved because the appropriate invalidated entry was still in the Capability Unit. As in the experiments to count domain switches (see Section 7.2.1) the results were obtained from running the Algol 68C compiler and the system generator. Three experiments were done to measure the slaving effectiveness for domain switching alone, process switching alone, and for both together.

By means of the hardware counter the number of successful reset cycles was recorded for a run using the normal microprogram and then for a run using a modified version of the microprogram. For the first experiment the microprogram was changed to ensure that on every switch from one protection domain to another, either by an ENTER or a RETURN instruction, the entries for the P,I and R capability segments and all entries dependent on them were removed from the Capability Unit instead of just being marked as invalid. These were the only entries which might have survived, as invalid entries, in the Unit until the next time they were needed. Details of the action of the ENTER and RETURN instructions are given in Section 3.4.

For the second experiment the modification ensured that every time a process was suspended and the Master Coordinator resumed, either because the process executed an Enter Coordinator (EC) instruction or was involuntarily suspended as a result of an interrupt, all the Capability Unit entries for that process were deleted. All the CAP Operating System processes except the Master Coordinator are at level 1 in the process hierarchy and switching from one level-1 process to another always takes place via the Master Coordinator. The number of switches between level-1 processes was measured by counting the number of times the Enter Subprocess (ESP) instruction was successfully executed by the Master Coordinator during runs of the Algol 68C compiler and the system generation program.

The time taken to complete a reset cycle depends on the level in the process hierarchy of the process whose capability is being evaluated. This was not measured experimentally but the cost can be readily calculated in terms of store cycles [Walker 1973]. To evaluate a capability for a data segment of the Master Coordinator requires a total of four store cycles, two to read a capability segment entry (a capability occupies two machine words) and two to read an entry in the Master Resource List. To evaluate a capability for a data segment of a level-1 process an additional five store cycles are needed, four to read an entry from a level-1 capability segment and an entry from a level-1 Process Resource List plus one cycle to extract a capability segment identity from the Process Base of the Master Coordinator, making nine store

cycles altogether. Most of the reset cycles saved under the conditions of the experiments would have been for data segments of level-1 processes.

Counting the reset cycles saved by the Capability Unit's slaving across process and domain switches does not give a complete picture of the benefits which accrue from this optimisation in the management of the Capability Unit. If the Unit were not managed in this way the microprogram would have to delete entries which became invalid because of a process or domain switch. This is in fact what the modifications to the microprogram achieved and an indication of the time that was saved by the optimisation was obtained by measuring the elapsed time for runs using the various microprograms. It was found that the ratio of the time saving to the number of reset cycles saved was reasonably constant. It should be noted that this time is not a measure of the time taken to perform a reset cycle because a large part, probably most, of the time would have been taken up in the removal of entries from the Capability Unit. Also, there would have been more activations of the CLOCK process during the runs using the modified microprograms as these took longer than the runs with the normal microprogram.

7.3 Results of experiments

The full results of the experiments and listings of the programs used for timing instructions are given in Appendix G. In the experiments to count domain switches and messages and to investigate slaving in the Capability Unit the results are the averages over

five repetitions of each experiment. This number of repetitions was chosen because little variation was observed in the five readings and because the results did not need to be obtained to a high degree of accuracy. Each instruction timing experiment was repeated ten times because greater precision was desirable. In these experiments the variation in the ten readings was extremely small.

7.3.1 Counting domain switches and messages sent

The results of these experiments are summarised in Table 7.1.

	Algol 68C	Genesis
Number of primary service calls	25896	4540
Number of consequential ENTERs	2810	6135
Consequential ENTERs/primary service call	0.11	1.35
Number of consequential messages	380	891
Consequential messages/primary service call	0.01	0.20

Table 7.1

For Algol 68C thirteen primary services were called from five different protected procedures. In running Genesis nineteen primary services were called from six protected procedures.

The interactive stream and spooled stream protected procedures provide a service called endof which can be called to test whether the end of a stream has been reached. As can be seen from the full results the experiments showed that the Algol 68C library

input/output routines were calling endof twice for all but the final request to read a buffer. These calls of endof accounted for the majority of the primary service calls. However, a call of read-buffer will return -1 if the end of the stream has been reached. The library was subsequently changed to use this mechanism and not to call endof at all. Table 7.2 shows the effect on the results of removing all calls of endof.

	Algol 68C	Genesis
Number of primary service calls	11786	1733
Number of consequential ENTERs	2810	6135
Consequential ENTERs/primary service call	0.24	3.54
Number of consequential messages	380	891
Consequential messages/primary service call	0.03	0.51

Table 7.2

7.3.2 Instruction timing

The results of the timing experiments are given in Table 7.3. The use of the slave stores during these experiments has already been commented on in Section 7.2.3.

Instruction(s)	time	multiple of BS instruction
BS	2.14μs	1.0
SB	1.85μs	0.9
TCS	2.80μs	1.3
ENTER + RETURN	0.24ms	114.6
ENTER + RETURN + capability argument + capability result	0.47ms	222.3

Table 7.3

Each numerical argument passed to, or numerical result returned by, a protected procedure requires one BS and one SB instruction taking a total of 3.98μs (1.9*BS). From the above results it can be estimated that passing a capability from one protected procedure to another, either as an argument or a result, takes 0.12ms (54*BS).

7.3.3 Slaving effectiveness of the Capability Unit

The results of the experiments to determine how successful the microprogram was at retaining entries in the Capability Unit over protection domain and process switches are given in Table 7.4. Table 7.5 gives the results of the experiments to count level-1 process switches and also includes the number of domain switches: this is twice the total number of ENTERs recorded during the experiments to count consequential ENTERs (see Section 7.2.1).

	Algol 68C		Genesis	
	reset cycles	% saved by slaving	reset cycles	% saved by slaving
No slaving across process or domain switches	460662	-	235865	-
Slaving across domain switches only	127480	72	151078	36
Slaving across process switches only	426322	8	195399	17
Slaving across process and domain switches	84911	82	85351	64

Table 7.4

	Algol 68C	Genesis
Number of level-1 process switches	5979	7237
Number of domain switches	59560	21412
Domain switches : level-1 process switches	10:1	3:1

Table 7.5

Switching from one process to another takes place less often than switching from one domain to another. It is not surprising, therefore, to find that the Capability Unit is less effective for slaving entries across process switches than across domain switches. The greater effectiveness for domain switching and the

less effective slaving for process switching during runs of Algol 68C compared with Genesis runs reflects the higher ratio of domain switches to process switches for Algol 68C.

The results of the experiments to measure the time saved by the Capability Unit's slaving across process and domain switches are given in Table 7.6.

	Algol 68C		Genesis	
	time secs.	% extra vs slaving	time secs.	% extra vs slaving
No slaving across process or domain switches	324.4	43	179.6	27
Slaving across domain switches only	235.9	4	155.9	10
Slaving across process switches only	306.9	36	170.0	20
Slaving across process and domain switches	226.4	-	141.6	-

Table 7.6

On average over all the Algol 68C and Genesis runs, for every reset cycle saved by slaving the time taken to run the program was reduced by about 0.24ms (113*BS).

7.4 Discussion of results

Counting the consequential ENTERs for a primary service provides an indication of the extra cost of the protection provided in the

CAP Operating System over and above that provided in a 2-state machine. However, caution must be exercised in interpreting these results because, although the first domain switch which takes place in response to a request for a primary service is treated as being in a sense equivalent to the switch to supervisor state in a 2-state computer, it is a much more costly operation. For example, on an IBM 370/165 to switch to supervisor state and then return to problem state requires the execution of a supervisor call (SVC) instruction (2.08μs) and a load program status word (LOAD PSW) instruction (0.98μs) taking a total of 3.06μs [IBM 1975]. This is equivalent to 19.1 times the load from store (LOAD) instruction (0.16μs), very much less than the 114.6 times the BS instruction for ENTER + RETURN on the CAP. Of course, switching protection domains on the CAP is a much more complex operation than switching states on the IBM 370. Also, a consequence of using capabilities is that much less checking needs to be done in the called procedure (or its equivalent) to validate passed arguments and this goes some way towards redressing the balance.

The objective of the experiments described in this part of the thesis was to compare the cost of using the powerful protection mechanisms of the CAP with that of using the very simple protection provided on a 2-state machine. Any comparison of two things which differ so much can only be coarse. It would be more reasonable to compare the CAP with a computer system whose protection mechanisms are of similar sophistication, for example Hydra [Wulf 1974] or the Plessey System 250 [England 1972]. No details

have been published of the performance of either system although, as noted in Section 2.2, Cohen [1975] describes the costs involved in switching protection domains on Hydra as "considerable". This is because Hydra does not have hardware support for the use of capabilities.

All the experiments were done with only one USER process running. Under normal conditions several USER processes would be active and process switching would be between more level-1 processes than in the experiments. The slaving effectiveness of the Capability Unit for process switching would, therefore, be lower than was measured.

The instruction to write the contents of a register to store (SB) on the CAP takes less time than the reverse operation (BS). Two factors contribute to this difference. Firstly, although the number of microinstructions obeyed is the same in each case, those for the SB instruction require 25 microprocessor steps whereas those for the BS instruction take 28 steps. The microinstructions obeyed in executing the two instructions and the number of microprocessor steps they take are given in Figure 7.1. Details of the format and interpretation of machine instructions are given in Section 3.2. The second factor is that for SB one microinstruction intervenes between starting the operand access (BM+AR->P.FETCH) and actually accessing store (AD->STORE). There is thus an opportunity for the execution of this instruction (BA->D) to overlap with the autonomous operation of the Capability Unit and the store logic (see Section 3.5). The ratio of the

number of microprocessor steps taken for the BS instruction to the number taken for SB is 1.12:1. The measured times for the instructions are in the ratio 1.16:1 indicating that both factors have an effect.

microinstructions	comment	steps
(a) BS instruction		
B15+1->I.FETCH	//start instruction fetch,increment B15	6
STORE->D,AD	//instruction from store	5
BM+AR->P.FETCH	//compute 'n', start operand access	6
STORE->D,AO	//operand from store	5
BA=D:RESTART	//ba:=s	6
	total steps	28
(b) SB instruction		
B15+1->I.FETCH	//start instruction fetch, increment B15	6
STORE->D,AD	//instruction from store	5
BM+AR->P.FETCH	//compute 'n', start operand access	6
BA->D	//operand to register D	4
AD->STORE:RESTART	//operand to store,s:=ba	4
	total steps	25

Figure 7.1 - BS and SB instructions

Chapter 8

8. Conclusions

The audit experiments described in Chapter 6 show that the technique for measuring protection presented in this thesis is capable of being applied to make actual measurements of a real protection system. Several of the suggestions put forward for improving the protection state of the CAP Operating System have subsequently been acted upon and others are under consideration thus providing further evidence of the value of this technique. These changes would probably not have been thought of without a systematic analysis of the protection aspects of the operating system. The measurement technique could be applied to other protection systems although it is particularly suitable for those which use capabilities.

The experiments to determine the costs of the protection provided on the CAP demonstrate that, at least for the programs run during the experiments, the number of domain switches which take place as the result of a request for one of the operating system's services is not unreasonably high. The time taken to switch from one domain to another, although not as low as had originally been hoped, was also not unduly high.

8.1 General guidelines

On the basis of the protection measurement experiments described in Section 6.1 the following are put forward as guidelines for designers and implementers of software, particularly operating

systems, on computers which provide sophisticated protection mechanisms. They are intended to help the designer or implementer to apply the principle of minimum privilege.

1. Restrict the objects accessible in a given domain at a particular time to those that need to be accessible for the process executing in that domain to be able to perform the service it has in hand at that time. For data the restrictions should limit the part of the segment which is accessible and the manner in which it may be accessed.
2. Ensure that any excessive rights which are incorporated during development work (e.g. for testing purposes or because certain facilities are not yet implemented) are removed as soon as they are no longer needed.*
3. Ensure that a process or protected procedure which receives a request for one of its services checks, for example by inspecting the software capability or access bits presented to it, that the caller is entitled to request that service.
4. Ensure that rights are destroyed as soon as they are no longer needed.+
5. If there is a mechanism for transferring rights between domains, ensure that these rights cannot be transferred unintentionally on a subsequent domain call. For example, in the CAP system ensure that on returning from a protected pro-

---

* It seems possible that failure to follow this advice may be a common source of errors in operating systems.
+ The CAP's REFINE instruction enables some of the rights built into a capability to be removed without destroying the whole capability.

cedure any capabilities present in the N capability segment, either arguments passed to the protected procedure or results returned by it, are destroyed after use. Otherwise, they will still be present in the N capability segment on the next call to a protected procedure.

6. Be wary of making rights accessible in all domains of a process. In particular, note that putting ENTER capabilities in global capability segments makes it possible for the corresponding protected procedures to be entered recursively.

A computer's protection mechanisms, however sophisticated, are of no benefit unless the designers and implementers of software for that computer make good use of those mechanisms. Adherence to the above guidelines would help in the minimisation of privilege; carrying out an audit, as was done for the CAP Operating System, is an effective way of checking that they are being followed.

8.2 Suggestions for further research

The detailed audit of the protection aspects of the CAP Operating System was done manually. If the process could be automated the time taken to apply protection measurement to an operating (or other large) system would be greatly reduced making the technique more attractive as a design tool. Investigating how to automate protection measurement could provide a fruitful area for further research.

In this thesis the protection state of an operating system has been investigated. Another possible extension of the research would be to apply the technique to a data base management system.

Some monitoring experiments were done on the CAP but there remains a lot more work to be done in this area.

Glossary

The first part of the glossary lists terms used in the thesis. The number in parentheses indicates the section in which the defining occurrence of that term is found. The second part lists the CAP Operating System processes and protected procedures: full details of these are given in the CAP Operating System Manual [Herbert 1978(c)].

1. Terms and abbreviations

A capability segment (3.4): capability segment 2, the 'argument' capability segment.

access matrix model (2.1): protection system model used by Lampson [1971], modified version used by Graham [1972].

accuracy measure (5.2.2): Jones's [1973] measure of how well a domain suits the task performed in it.

audit (6.1): the systematic analysis of the protection aspects of the CAP Operating System.

BS (7.2.3): instruction to load a register from store.

C-stack (3.3): stack used in implementing protected procedure calls.

CAP (2.2): capability research computer built at Cambridge University.

capability (2.1): unforgeable incorruptible ticket of permission.

capability segment (3.3): segment which holds capabilities.

Capability Unit (3.5): CAP computer's hardware which supports the use of capabilities.

CAP Operating System (3.7): operating system for the CAP.

consequential ENTERs (7.2.1): domain switches which take place as a consequence of a call for a primary service.

consequential messages (7.2.2): messages sent as a consequence of a call for a primary service.

degree of overprivilege (5.3.1): 1 - |M|/|I|.

domain change (3.1): modification to a domain.

domain switch (3.1): leaving one domain and entering another

EC (3.3): enter coordinator instruction.

ENTER (3.4): instruction to enter a protected procedure.

ENTER capability (3.4): capability which must be presented when entering a protected procedure; interpreted by microprogram.

ESP (3.3): enter sub-process instruction.

exposure measure (5.2): measures how well protected the objects in a system are.

function (5.3): the services provided by a module are made available to other modules as functions.

Function Memory (3.2): hardware used to speed up instrction interpretation.

FM (3.2): Function Memory.

gate-keeper (6.1.6): protected procedure used to validate a call for an operating system service.

G capability segment (6.1.1): capability segment 1, the 'global' capability segment.

general address (3.3): virtual address presented by a process:

interpreted relative to the address space of that process's coordinator.

Genesis (7.1): program for generating the CAP Operating System.

I (5.3): implementation set.

I capability segment (3.4): capability segment 5, the 'interface' capability segment.

implementation set (5.3): the set of objects which the process can address.

inform (3.7): in main memory.

M (5.3): minimum privilege set.

Master Coordinator (3.3): root process in hierarchy.

Master File Directory (6.1.1): directory in which are preserved capabilities for User File Directories (UFDs) and operating system directories (e.g. spool directories). It has nothing to do with the Master Coordinator but provides root access to the general naming structure of the filing system.

Master Resource List (3.3): PRL of Master Coordinator.

message-channel capability (6.1): software capability which must be presented when sending a message to a process.

MFD (6.1.1): Master File Directory.

minimum privilege model (5.3): the protection system model used in this thesis.

minimum privilege set (5.3): the set of objects which must be accessible to the process in order that the desired service be performed.

MRL (3.3): Master Resource List.

N capability segment (3.4): capability segment 3, the 'new argument' capability segment.

object (2.1): an entity to which access must be controlled.

outform (3.7): not in main memory.

PB (3.3): Process Base.

P capability segment (3.4): capability segment 4, the 'program' capability segment.

P category function (6.1.1): function whose invalid use has breach of protection implications.

P-store (3.6): a set of absolute locations in main memory used to control access to peripherals.

P-store capability (6.1): capability which must be presented with a request to operate a peripheral.

primary service (7.2.1): publicly available operating system service.

principle of minimum privilege (2.1): every program and every user of a system should operate using the least set of privileges necessary to complete the job.

privilege measure (5.2): measure to determine how closely a system adheres to the principle of minimum privilege.

PRL (3.3): Process Resource List.

problem state (7.1): state of a 2-state machine in which user programs run.

Process Base (3.3): segment used for preservation of machine and process state on a process becoming dormant.

Process Resource List (3.3): specifies which objects are potentially accessible to the process.

protected procedure (3.4): procedure with its own fully encapsulated address space.

protection (2.1): control, by logical and physical mechanisms, of access to objects inside a computer system.

protection domain (3.1): the set of objects accessible to the process.

protection environment (3.3): all objects potentially accessible to the process.

R capability segment (3.4): capability segment 6, the 'resource' capability segment.

RETURN (3.4):instruction to return from a protected procedure.

reset cycle (3.5): microprogram code executed to evaluate a store capability to give an absolute address.

S (5.3): specification set.

SB (7.2.3): instruction to copy register's contents to store.

segment (3.3): set of contiguous words of memory.

service (5.3): each module in the system has a repertoire of services it can perform.

software capability (3.7): capability which, although protected in the same way as store and ENTER capabilities, is interpreted by software.

specification set (5.3): set of objects which, according to some external specification, need to be accessible to the process.

stage one (3.2): microprogram code executed as the initial stage in the interpretation of all instructions.

store capability (3.3): capability for segment of main memory; interpreted by hardware.

subject (2.1): an active entity whose access to objects must be controlled.

suitability measure (5.2.2): Jones's [1973] measure of how well a particular system satisfies a given demand.

supervisor state (7.1): state of a 2-state machine in which the operating system runs.

Tag Memory (3.5): part of Capability Unit; used as indexed translation table to convert general address into internal form used in the Capability Unit.

TGM (3.5): Tag Memory.

U category function (6.1.1): function whose invalid use consumes system resources.

V category function (6.1.1): function whose invalid use has security implications.

Virtual Memory Object (3.7): unit of swapping in the virtual memory system.

VMO (3.7): Virtual Memory Object.

W category function (6.1.1): function not in category P, U or V.

2. CAP Operating System

2.1 Processes

BRSDISC: drives Burroughs disc unit.

CDCDISC: drives CDC disc unit.

CLOCK: interface to hardware clock.

ENSURER: ensures that file directories etc. are up to date on disc.

LPRDESPOOL: spools output to line printer.

MC: Master Coordinator; schedules processor, handles interrupts, etc.

MODULE: manages map of CDC disc allocation.

PRINTER: drives line printer.

PTPDESPOOL: spools output to paper tape punch.

PUNCH: drives paper tape punch.

READER: drives paper tape reader.

RESTART: starts up operating system.

RSM: real store manager.

TELETYPE: drives teletypes and VDUs.

USER: runs a user's computation.

VSM: virtual store manager.

2.2 Protected Procedures

DIRMAN: file directory manager.

ECPROC: gate-keeper for calls to Master Coordinator; provides some coordinator services directly.

FAULTPROC: handles faults; runs in same protection domain as EC-PROC.

IOC: input output controller; manages allocation of peripherals.

ISPP: manages interactive input and output streams.

LINKER: links outform ENTER capabilities so that they can be used to enter protected procedures.

LOGON: checks and manages passwords for logging in.

MAKEENTER: manufactures capability segments and ENTER capabilities.

PRLGARB: recovers garbage from PRL.

SETUP: sets up message channels.

SINMAN: manages directory of system internal names of VMOs.

STOREMAN: gate-keeper providing user interface to SINMAN, RSM and VSM.

References

Bard 1973
Bard,Y. "Experimental evaluation of system performance", IBM Systems Journal, vol. 12, no. 3, 1973, pp 302-314.

Batson 1970
Batson,A., Ju,S.-M., Wood,D.C. "Measurements of segment size", CACM, vol. 13, no. 3, March 1970, pp 155-159.

Bell 1973
Bell,T.E. "Performance determination - the selection of tools, if any", Proc. AFIPS National Computer Conference, vol. 42, June 1973, pp 31-38.

Boi 1973
Boi,L., Drucbert,J.P. "OSSYOSCOPE: system on auxiliary processors for measuring operating systems", Proc. International Computing Symposium, 1973, pp 195-199.

Bordsen 1971
Bordsen,D.T. "UNIVAC 1108 hardware instrumentation system", Proc. ACM SIGOPS Workshop on system performance evaluation, Harvard, April 1971, pp 1-28.

Bourne 1975
Bourne,S.R., Birrell,A.D., Walker,I. "ALGOL68C Reference Manual", Cambridge University Computer Laboratory, July 1975.

Boyse 1975
Boyse,J.W., Warn,D.R. "A straightforward model for computer performance prediction", Computing Surveys, vol. 7, no. 2, June 1975, pp 73-93.

Browne 1976
Browne,P.S. "Computer security - a survey", Proc. AFIPS National Computing Conference, vol. 45, 1976, pp 53-63.

Calingaert 1967
Calingaert,P. "System performance evaluation: survey and appraisal", CACM, vol. 10, no. 1, January 1967, pp 12-18.

Chastain 1973
Chastain,D.R. "Security vs performance", Datamation, vol. 19, no. 11, November 1973, pp 110-111 & 116.

Cohen 1975
Cohen,E., Jefferson,D. "Protection in the Hydra operating system", Proc. 5th Symposium on Operating System Principles, November 1975, pp 141-160.

Cook 1978(a)
Cook,D.J. "Measuring memory protection", accepted for 3rd

International Conference on Software Engineering, Atlanta, Georgia, May 1978.

Cook 1978(b)
Cook,D.J. "The cost of using the CAP computer's protection facilities", to appear in Operating Systems Review, vol. 12, no. 2, April 1978.

Cook 1978(c)
Cook,D.J. "Measuring memory protection on the CAP computer", submitted to 2nd International Symposium on Operating Systems, Le Chesnay, France, October 1978.

Dahl 1966
Dahl,O.-J., Nygaard,K. "Simula - an Algol-based simulation language", CACM, vol. 9, no. 9, September 1966, pp 671-678.

Denning 1971
Denning,P.J., Eisenstein,B.A. "Statistical methods in performance evaluation", Proc. ACM SIGOPS Workshop on system performance evaluation, Harvard, April 1971, pp 55-61.

Dennis 1966
Dennis,J.B., van Horn,E.C. "Programming semantics for multiprogrammed computations", CACM, vol. 9, no. 3, March 1966, pp 143-155.

Denny 1975
Denny,W.M. "Micro-programming measurement techniques for the Burroughs B1700", in "Lecture Notes in Computer Science", ed. G.Goos & J.Hartmanis, vol. 26, pp 453-462, Springer-Verlag, Berlin, 1975.

Denny 1977
Denny,W.M. "The Burroughs B1800 microprogrammed measurement system: a hybrid hardware/software approach", Proc. 10th Annual Workshop on Microprogramming, Niagara Falls, October 1977, pp 66-70 (in SIGMICRO Newsletter, vol. 8, no. 3, September 1977).

Digital 1971
Digital Equipment Corporation "PDP-11/45 Processor Handbook", 1971.

Ellis 1974
Ellis,C.A. "Analysis of some abstract measures of protection in computer systems", Colorado University Dept. of Computer Science, NTIS PB-235-297, May 1974.

England 1972
England,D.M. "Architectural features of System 250", Infotech State of the Art Report no. 14, 1972, pp 395-427

England 1974
England,D.M. "Capability concept, mechanisms and structure in System 250", Proc. International Workshop on Protection in Operating Systems, IRIA, August 1974, pp 63-82.

Fabry 1968
Fabry,R.S. "Preliminary description of a supervisor for a machine oriented around capabilities", University of Chicago ICR Quarterly Report, no. 18, August 1968.

Fabry 1974
Fabry,R.S. "Capability-based addressing", CACM, vol. 17, no. 7, July 1974, pp 403-412.

Glaser 1967
Glaser,E.L. "A brief description of the privacy measures in the Multics operating system", Proc. AFIPS SJCC, vol. 30, 1967, pp 303-304.

Graham 1972
Graham,G.S., Denning,P.J. "Protection - principles and practice", Proc. AFIPS SJCC, vol. 40, 1972, pp 417-429.

Herbert 1978(a)
Herbert,A.J.(ed.) "CAP System Programmer's Manual", Cambridge University Computer Laboratory, 1978.

Herbert 1978(b)
Herbert,A.J.(ed.) "CAP Hardware Manual", Cambridge University Computer Laboratory, 1978.

Herbert 1978(c)
Herbert,A.J.(ed.) "CAP Operating System Manual", Cambridge University Computer Laboratory, 1978.

Herbert 1978(d)
Herbert,A.J. "A new architecture for the Cambridge capability computer", Operating Systems Review, vol. 12, no. 1, January 1978, pp 24-28.

IBM 1970
IBM "System 360 Principles of Operation", IBM Manual no. GA22-6821-8, November 1970.

IBM 1975
IBM "System 370 Model 165 Functional Characteristics", IBM Manual no. GA22-6935, January 1975.

Jalics 1974
Jalics.P.J., Lynch,W.C. "Selected measurements of the PDP-10 TOPS-10 operating system", Proc. IFIP Congress, Stockholm, August 1974, pp 242-246.

Jones 1973
Jones,A.K. "Protection in programmed systems", PhD Thesis, Carnegie-Mellon University, June 1973.

Jones 1975
Jones,A.K., Wulf,W.A. "Towards the design of secure systems", Software Practice and Experience, vol. 5, no. 4, Oct-Dec 1975, pp 321-336.

Kimbleton 1972
Kimbleton,S.R. "Performance evaluation - A structured approach", Proc. AFIPS SJCC, vol. 40, 1972, pp 411-416.

Knuth 1973
Knuth,D.E., Stevenson,F.R. "Optimal measurement points for program frequency counts", BIT, vol. 13, no. 3, 1973, pp 313-322.

Lampson 1971
Lampson,B.W. "Protection", Proc. Fifth Princeton Symposium on Information Sciences and Systems, Princeton University, March 1971, pp 437-443, reprinted in Operating Systems Review, vol. 8, no. 1, January 1974, pp 18-24.

Lampson 1976
Lampson,B.W., Sturgis,H.E. "Reflections on an operating system design", CACM, vol. 19, no. 5, May 1976, pp 251-265.

Lauer 1974
Lauer,H.C. "protection and hierarchical addressing structures", Proc. International Workshop on Protection in Operating Systems, IRIA, August 1974, pp 137-148.

Levin 1975
Levin,R., Cohen,E., Corwin,W., Pollack,F. Wulf,W. "Policy/mechanism separation in Hydra", Proc. 5th Symposium on Operating System Principles, November 1975, pp 132-140.

Lucas 1971
Lucas,H.C.(jr.) "Performance evaluation and monitoring", Computing Surveys, vol. 3, no. 3, September 1971, pp 79-91.

Lynch 1972
Lynch,W.C. "Operating system performance", CACM, vol. 15, no. 7, July 1972, pp 579-585.

MacEwan 1974
MacEwan,G.H. "On instrumentation facilities in programming languages", Proc. IFIP Congress, 1974, pp 198-203.

Needham 1971
Needham,R.M. "Handling difficult faults in operating systems", Proc. 3rd Symposium on Operating System Principles, October 1971, pp 55-57.

Needham 1972
Needham,R.M. "Protection systems and protection implementations", Proc. AFIPS FJCC, vol. 41, 1972, pp 571-578

Needham 1973
Needham,R.M. "Protection - A current research area in operating systems", Proc. International Computing Symposium, 1973, pp 123-126.

Needham 1974(a)
Needham,R.M., Wilkes,M.V. "Domains of protection and the management of processes", Computer Journal, vol. 17, no. 2, May 1974, pp 117-120.

Needham 1974(b)
Needham,R.M., Walker,R.D.H. "Protection and process management in the "CAP" computer", Proc. International Workshop on Protection in Operating Systems, IRIA, August 1974, pp 155-160

Needham 1977(a)
Needham,R.M. "The CAP Project: an interim evaluation", Proc. 6th Symposium on Operating Systems Principles, Purdue University, 16-18 November 1977, pp 17-22.

Needham 1977(b)
Needham,R.M., Birrell,A.D. "The CAP filing system", Proc. 6th Symposium on Operating Systems Principles, Purdue University, 16-18 November 1977, pp 11-16.

Needham 1977(c)
Needham,R.M., Walker,R.D.H. "The Cambridge CAP computer and its protection system", Proc. 6th Symposium on Operating Systems Principles, Purdue University, 16-18 November 1977, pp 1-10.

Pinkerton 1969
Pinkerton,T.B. "Performance monitoring in a time-sharing system", CACM, vol. 12, no. 11, November 1969, pp 608-610.

Randell 1977
Randell,B., Lee,P.A., Treleaven,P.C. "Reliable Computing Systems", University of Newcastle Computing Laboratory Technical Report No. 102, May 1977.

Redell 1974
Redell,D.D. "Naming and protection in extendible operating systems", PhD Thesis, University of California, Berkeley, September 1974.

Rosin 1969
Rosin,R.F. "Contemporary concepts of microprogramming and emulation", Computing Surveys, vol. 1, no. 4, December 1969, pp 197-212.

Rosin 1974
Rosin,R.F. "The significance of microprogramming", SIGMICRO Newsletter, vol. 4, no. 4, January 1974, pp 24-39.

Saal 1972
Saal,H.J., Shustek,L.J. "Microprogrammed implementation of computer measurement techniques", Proc. ACM Annual Workshop on Microprogramming, Illinois, September 1972, pp 42-50.

Saal 1975
Saal,H.J., Shustek,L.J. "On measuring computer systems by microprogramming", Infotech State of the Art Report no. 23, 1975, pp 473-489.

Saltzer 1970
Saltzer,J.H., Gintell,J.W. "The instrumentation of Multics", CACM, vol. 13, no. 8, August 1970, pp 495-500.

Saltzer 1974(a)
Saltzer,J.H. "Protection and the control of information sharing in Multics", CACM, vol. 17, no. 7, July 1974, pp 388-402.

Saltzer 1974(b)
Saltzer,J.H. "Ongoing research and development on information protection", Operating Systems Review, vol. 8, no. 3, July 1974, pp 8-24.

Saltzer 1975
Saltzer,J.H., Schroeder,M.D. "The protection of information in computer systems", Proceedings of the IEEE, vol. 63, no. 9, September 1975, pp 1278-1308.

Schroeder 1971
Schroeder,M.D. "Performance of the GE-645 associative memory while Multics is in operation", Proc. SIGOPS Workshop on System Performance Evaluation, April 1971, pp 227-245.

Schroeder 1972(a)
Schroeder,M.D., Saltzer,J.H. "A hardware architecture for implementing protection rings", CACM, vol. 15, no. 3, March 1972, pp 157-170.

Schroeder 1972(b)
Schroeder,M.D. "Cooperation of mutually suspicious subsystems in a computer utility", PhD Thesis, MIT, September 1972.

Schroeder 1975
Schroeder,M.D. "Engineering a security kernel for Multics", Proc. 5th Symposium on Operating System Principles, November 1975, pp 25-32.

Sketler 1974
Sketler,A.C. "Controlled testing for computer performance

evaluation", Proc. AFIPS National Computer Conference, vol. 43, 1974, pp 693-699.

Slinn 1977
Slinn,C.J. "Aspects of a capability based operating system", PhD Thesis, Cambridge University, February 1977.

Sturgis 1973
Sturgis,H.E. "A postmortem for a time sharing system", PhD Thesis, University of California, Berkeley, 1973.

Waite 1976
Waite,W.M. Editorial in Operating Systems Review, vol. 10, no. 3, July 1976.

Walker 1973
Walker,R.D.H. "The structure of a well protected computer", PhD Thesis, Cambridge University, December 1973.

Weissman 1969
Weissman,C. "Security controls in the ADEPT-50 time sharing system", Proc. AFIPS FJCC, vol. 35, 1969, pp 119-133.

Weissman 1972
Weissman,C. "Trade-off considerations in security system design", Data Management, April 1972, pp 14-19.

Wilkes 1975
Wilkes,M.V. "Time-sharing computer systems", 3rd edition, Macdonald/American Elsevier Monographs no. 5, London 1975.

Williams 1972
Williams,T. "Computer systems measurement and evaluation", Computer Bulletin, vol. 16, no. 2, February 1972, pp 100-104.

Wilner 1972
Wilner,W.T. "Design of the Burroughs B1700", Proc. AFIPS FJCC, vol. 41, 1972, pp 489-497.

Wulf 1974
Wulf,W., Cohen,E., Corwin,W., Jones,A., Levin,R., Pierson,C., Pollack,F. "HYDRA: the kernel of a multi-processor operating system", CACM, vol. 17, no. 6, June 1974, pp 337-345.

Wulf 1975
Wulf,W., Levin,R., Pierson,C. "Overview of the Hydra operating system development", Proc. 5th Symposium on Operating System Principles, November 1975, pp 122-131.

Wyeth 1976
Wyeth,D. "On the comparison of protection systems", PhD Thesis, Newcastle University, July 1976.

Appendix A

This appendix details the assumptions made in carrying out the audit. It provides information supplementary to that given in Chapter 6.

1. The minimum privilege set, M, was taken to be composed of the functions actually called (including those called via library procedures) plus the functions made accessible (but not used) in a domain by the creation of rights to be returned to the caller of the protected procedure.
2. Possession of an ENTER capability for a protected procedure other than ECPROC or SETUP made accessible all the functions in its repertoire. For ECPROC and SETUP certain functions were only accessible if the appropriate software capability was held as well as the ENTER capability.
3. Possession of a send-message capability for a channel to a process made accessible all the functions in the process's repertoire for that channel.
4. Possession of a P-store capability for a peripheral device made accessible the functions corresponding to the services (i.e. peripheral device operations) provided by that peripheral. In some cases a single P-store capability made more than one function accessible. The normal mechanism for acquiring a P-store capability was to call ECPROC's claim-device function. During the early development of the operating system this mechanism was not implemented and some processes

were given a P-store capability in capability segment 0 at system generation. The audit assumed that these P-store capabilities, which made the peripheral device functions accessible in all domains of the process, were not included at system generation but that a suitably restricted P-store capability was obtained instead by the normal mechanism and kept in a non-global capability segment.

5. The audit took account of functions made accessible in a domain by capabilities passed to it as the arguments for, or returned to it as the results from, a call.

6. Capabilities in the N capability segment of a domain, put there as the arguments for or results from a call to a protected procedure, will, unless explicit action is taken to destroy them, survive until another protected procedure is called from the first domain (see Section 8.1). The domain of the second protected procedure called could, therefore, have rights which it was not intended to have. This possibility was ignored in the audit.

7. Fault handling in the CAP Operating System gives the faulted procedure a chance to sort things out. In Algol 68C this was done by automatically calling the run time error procedure. Any functions called from the run time error procedure were included in the minimum privilege set. In other words, it was assumed that blunders could occur at any time.

8. ECPROC's* code for handling faults was logically divided into three parts to deal with blunders, virtual memory faults and the linking of outform ENTER capabilities. These were treated as three ECPROC services handle-blunders, resolve-VM-faults and link-outform-ENTER called implicitly. The (ECPROC, handle-blunders) function was assumed to be always accessible and always needed. It was, therefore, always included in both sets I and M. The (ECPROC, resolve-VM-faults) function was assumed to be both accessible and required unless the core residence constraints imposed made virtual memory faults impossible. The (RSM,, segment-outform-at-mc-level) function was similarly assumed to be both accessible and needed in the ECPROC domain unless virtual memory faults were ruled out. The (ECPROC, link-outform-ENTER) function was assumed to be both accessible and required only when there was a call to a protected procedure whose (outform) ENTER capability had been retrieved from the filing system [Needham 1977(b)].
9. The PRLGARB protected procedure was assumed to be entered (with a request for its sole service garbage-collect-PRL) whenever the (ECPROC, create-PRL-entry) function was called and the ECPROC domain had an ENTER capability for PRLGARB.
10. The CAP Operating System was still under development while the audit was in progress. Until near the end of the audit the data was adjusted to take account of changes: the system was

---

* The fault handling code is in the FAULTPROC procedure. As the ECPROC and FAULTPROC procedures are in the same domain they are referred to jointly as ECPROC for simplicity.

then frozen as far as the audit was concerned. The audit relates to the operating system as it was at the beginning of December 1976.

11. The message-channel-access capabilities for presentation to the SETUP protected procedure were designed to incorporate bits indicating what access (i.e. send, receive, or send and receive) the caller is entitled to have for that channel. These bits were assumed to be set as appropriate although at the time of the audit the system generation program did not deal with them.

12. All CAP Operating System processes have a number of ENTER capabilities (e.g. those for ECPROC and SETUP) in their G capability segments. These ENTER capabilities are, therefore, present in all domains in which that process runs, including the domains which they are used to enter. Thus there is the opportunity for some protected procedures to enter themselves recursively. Such a recursive call would not change the privileges which the protected procedure and its caller have but would almost certainly cause the protected procedure to behave incorrectly. The ability of a protected procedure to enter itself recursively was counted as making a single additional function (not the procedure's full repertoire) accessible in the domain corresponding to that protected procedure. When the functions were divided into categories this function was counted as in category W.

13. The programs of typical protected procedures and processes

in the CAP Operating System start with initialisation code which, because a kind of coroutine mechanism has been implemented, is executed once only [Needham 1977(a)]. Consequently, the function corresponding to the initialise service of procedures and processes was assumed to be accessible only until it had been used. In some cases the first call of the STOREMAN or SINMAN protected procedure, and thus the invocation of the function corresponding to that procedure's initialise service, could have been in the course of performing one of a number of services in the calling domain. In this situation the (STOREMAN, initialise) or (SINMAN, initialise) function was assumed to be invoked for each of the services in the calling domain which could make the first call.

Specific checks in the programs made certain that some other functions (e.g. (DIRMAN, initialise-directory)) too are accessible for a single use only and the audit took account of this.

The initialisation code of the SINMAN protected procedure included code which was only executed if a zero argument was passed by the caller. This was treated as a distinct SINMAN service, which was called start-up, only accessible at the same time as the (SINMAN, initialise) function.

14. The services of the DIRMAN protected procedure are protected by access matrices kept in the filing system [Needham 1977(b)]: the functions corresponding to these services were

all, therefore, counted as being in category W.

15. The functions corresponding to the ECPROC send-.....-message and send-.....-message-wait-event pairs of services were treated as being the same function. The functions corresponding to the return-.....-message and return-.....-message-wait-event pairs of services were treated likewise.

16. No attempt was made to determine the objects potentially accessible in a given domain (see Section 5.2.3). In particular, objects which could be made accessible by retrieving a capability from the filing system were only included in the audit if the appropriate capability was in fact retrieved.

Appendix B

As mentioned in Chapter 6, in the process of doing the audit a number of points were noted about the CAP Operating System. They are detailed in this appendix.

1. Certain services (e.g. RSM's initialise-2nd-part, SINMAN's start-up and MODULE's restart) are only used for system initialisation. The code which is executed in the performance of these services should ensure that the services can only be performed once for each run of the operating system. A subsequent attempt to request these services should be rejected, and probably faulted. Also, the rights to request these services should only be in the domain in which they are required and should be destroyed as soon as they have been used. This belt-and-braces strategy is justified because of the havoc that would be caused by these services being performed after the system has been initialised. At the time of the audit the position was that:-

   (a) RSM had code to ensure that its initialise-2nd-part service was performed once only: subsequent requests for the service would not cause the service to be performed but a success code would, nevertheless, be returned.

   (b) a request for MODULE's restart service caused MODULE to poll its second message channel via which it was passed a segment. The only capability for this second channel was acquired by the SINMAN protected procedure while it

was performing its start-up service and was disposed of after it had been used (by calling the freeslot library procedure: but see 6 below). If there had been a subsequent request for MODULE to perform its restart service MODULE would have looped waiting for a message which could never be sent because no send-message capability for the second message channel to MODULE existed.

(c) the code for SINMAN's start-up service was part of the SINMAN program's initialisation code but was only executed if a zero argument was passed on the first entry to an instance of the SINMAN protected procedure. Thus the (SINMAN, start-up) function, which dealt with the initialisation of global system data structures, was accessible as often as the (SINMAN, initialise) function, which only did the initialisation work for a particular instance of SINMAN. The program did not check for attempts to initialise the system data structures more than once.

2. As mentioned in Appendix A item 4, until ECPROC's claim-device service was implemented processes were given P-store capabilities at system generation. The code of two processes, CDCDISC and RESTART, still referenced these capabilities rather than the ones obtained by their calls of (ECPROC,claim-device). The system generation P-store capabilities were held in capability segment 0 and were thus accessible in all domains which the process entered: in the

case of RESTART this globally accessible P-store capability was for the whole P-store! It is easy to forget to remove short-cuts such as these taken during the development of a system.

Another left-over was that the TESTER program included a request for DIRMAN's update service although that service had been withdrawn.

3. The initmap and restore-map services were still included in MODULE's repertoire although any use of them would have had disastrous consequences for the filing system! They were subsequently withdrawn.

4. The Master Coordinator did not check to ensure that a given process was only created once. Such a check could have been incorporated but, as the create-process service was only available in two highly trusted domains, it would probably not have been worth while.

5. The peripheral permission software capability, presented to ECPROC when requesting a P-store capability for a particular device, could be used to support a request for any P-store capability. It need not have been so general and could have had built into it the device(s) for which it was valid.

6. The Algol 68C libraries included procedures getslot and freeslot which were used to manage the allocation of slots in the I capability segment. The freeslot procedure made a slot available for re-allocation but did not destroy the capability in the slot that had been freed. (A comment in the SINMAN pro-

gram implied that the capability was destroyed.) The freeslot procedure has since been modified to invalidate the capability in the released slot.

7. The services provided by certain protected procedures were made generally available in a process by putting the ENTER capabilities for these procedures in the G capability segment. This mechanism was used to provide public facilities. However, it had the side effect of making it possible for these protected procedures to be entered recursively (see Appendix A item 12).

Appendix C

Categorisation of CAP Operating System functions (see Section 6.1.1). The tables below give the categories for the functions corresponding to the services listed.

1. Services provided by processes

Process	Service	Cat.	Service	Cat.
BRSDISC	initialise BRSDISC	W	read map from disc	V
	write map to disc	P		
CDCDISC	initialise CDCDISC	W	read disc	V
	write disc	P		
CLOCK	initialise CLOCK	W	read date and time	W
ENSURER	initialise ENSURER	W	ensure file directory	W
	ensure and remove	P	delete	P
LPRDESPOOL	initialise LPRDESPOOL	W	spooled	P
MC	wait event	W	create message channel	P
	stop system	P	reserve segment	P
	release segment	P	create process	P
	inform segment	P	outform segment	P
	claim device	P	release device	P
	wake up from send/return	P	wait condition	P
	signal condition	P	wait for message channel	P
	trace	W		
MODULE	initialise MODULE	W	reserve space	U
	extend chain	P	contract chain	P
	delete chain	P	bad block	P
	disc address	W	ensure map on disc	W
	checkmap	W	restore map	P
	initmap	P	free blocks	W
	chain length	W	restart	P
PRINTER	initialise PRINTER	W	print line	W
	print document	W		
PTPDESPOOL	initialise PTPDESPOOL	W	spooled	P
PUNCH	initialise PUNCH	W	punch line	W
	punch document	W		
READER	initialise READER	W	read line	W
	read document	W		
RESTART	restart	P	allocate USER process	W
	USER process finished	P		
RSM	initialise RSM	P	segment outform at mc level	P
	ensure mcaddress ok	W	outform	W
	change segment size	P	delete	P
	move window	P	initialise	P
	new thing	P		
TELETYPE	initialise TELETYPE	W	output line	W
	input line	W	output document	W
	input document	W		

USER	initialise USER	W	run user process	W
VSM	initialise VSM	P	sin to mcaddr	P
	mcaddr to sin	V	ensure sin	W
	disc addr to mcaddr	P	close window	P
	delete mcaddr	P		

2. Services provided by protected procedures

Procedure	Service	Cat.	Service	Cat.
DIRMAN	initialise DIRMAN	W	initialise directory	W
	retrieve	W	remove	W
	retain	W	alter access	W
	examine	W	file details	W
	reduce directory status	W	file examine	W
ECPROC	wait event	W	send null message	U
	send data message	U	send segment message	U
	send full message	U	receive null message	W
	receive data message	W	receive segment message	W
	receive full message	W	receive reply data message	W
	receive reply segment message	W	receive reply full message	W
	return data message	U	return (segment) message	U
	return data (full) message	U	create message channel	P
	stop system	P	reserve for reading	W
	reserve for writing	W	release reservation	W
	clear fault	W	cause fault	W
	return fault	W	create process	P
	delete process	P	create PRL entry	P
	create capability	P	delete capability	P
	update capability	P	read PRL entry	V
	read capability	V	inform segment	P
	outform segment	P	claim device	P
	release device	P	process info	V
	handle blunders	P	resolve VM faults	P
	link outform ENTER	P		
IOC	request spooled stream	W	request interactive stream	W
	despool to printer	W	despool to tape punch	W
	send document to tape reader	W	send document to tape punch	W
	send document to printer	W	send document to tty1	W
	send document to tty2	W	send document to tty3	W
	send document to tty4	W		
o/p ISPP	initialise ISPP	W	write buffer	P
	close	P	state	W
	reset state	P		
i/p ISPP	initialise ISPP	W	read buffer	P
	close	P	endof	W
	state	W	reset state	P
LINKER	link PCB	W		
LOGON	validate user	W		
MAKEENTER	make cap seg	W	make ENTER	W
PRLGAR	garbage collect PRL	W		
SET	set up reply	U	set up reply with store	U

	set up receive	P	set up send	P
	set up send with reply	P		
SINMAN	initialise SINMAN	P	start up	P
	dynamic increment	U	dynamic decrement	P
	newseg:access change	U	newseg:new	U
	seg from sin	P	cap from sin	P
	retain	W	remove	P
	sininf	V	sin of MFD	V
	restart	P	usecount	V
	capinf	V	change size	P
	make swc	P		
STOREMAN	initialise STOREMAN	W	ensure ok	W
	outform	W	change size	W
	newseg:access change	U	newseg:new	U
	capinf	V	get size access	W
	open window	W	move window	W
	close window	W	cleanse	W
	details	V		

3. Services provided by peripheral devices

Service	Cat.	Service	Cat.
read Burroughs	V	write Burroughs	P
read CDC	V	write CDC	P
read date	W	read time	W
write to printer	P	write to punch	P
read from p/tape reader	P	read from teletype	P
write to teletype	P	read clock	W
set clock interval	P		

Appendix D

This appendix gives the frequency counts of P category functions (excluding the functions corresponding to the services provided by the Master Coordinator). The table number quoted for each set of results is the table in Chapter 6 to which that set of results corresponds. The results corresponding to Table 6.6 show only the figures that are different from those for Table 6.5. The results corresponding to Table 6.7 are the same as those shown for Table 6.6 except in the case of o/p ISPP's write-buffer service for which the results are the same as for Table 6.5. The results are for the functions corresponding to the services listed.

		Table 6.3		Table 6.4		Table 6.5		Table 6.6	
Module	Service	times accessible	% not needed	times accessible	% not needed	times accessible	% not needed	times accessible	% not needed
BRSDISC	write map to disc	15	67	15	67	15	67		
CDCDISC	write disc	13	8	13	8	13	8		
ENSURER	ensure & remove	38	55	21	19	17	0		
	delete	25	80	16	69	5	0		
LPRDESPOOL	spooled	0	0	0	0	0	0		
MODULE	extend chain	46	57	31	35	23	0		
	contract chain	46	57	31	35	23	0		
	delete chain	46	65	19	16	19	16		
	bad block	46	67	15	0	15	0		
	restore map	46	67	15	0	15	0		
	initmap	46	67	15	0	15	0		
	restart	2	0	2	0	2	0		
PTPDESPOOL	spooled	0	0	0	0	0	0		
RESTART	restart	1	0	1	0	1	0		
	USER process finished	1	0	1	0	1	0		
RSM	initialise RSM	1	0	1	0	1	0		
	seg outform at mc level	65	0	65	0	65	0		
	change seg size	53	60	32	34	21	0		
	delete	53	66	23	22	23	22		
	move window	53	60	27	22	21	0		
	initialise:2nd part	2	50	2	50	1	0		
	new thing	53	66	23	22	23	22		
VSM	initialise VSM	1	0	1	0	1	0		
	sin to mcaddr	45	69	24	0	24	0		
	disc addr to mcaddr	45	67	30	50	15	0		
	close window	45	58	25	24	19	0		
	delete mcaddr	45	36	29	0	29	0		
ECPROC	create message channel	35	0	35	0	35	0		
	stop system	134	0	134	0	134	0		
	create process	12	83	2	0	2	0		
	delete process	0	0	0	0	0	0		
	create PRE set	91	32	67	7	62	0		
	create c[illegible]	91	73	34	26	25	0		

	update capability	91	91	24	67	14	43		
	delete capability	91	100	0	0	0	0		
	inform segment	9	44	9	44	9	44		
	outform segment	9	0	9	0	9	0		
	claim device	28	71	8	0	8	0		
	release device	28	96	1	0	1	0		
	handle blunders	238	0	238	0	238	0		
	resolve VM faults	214	0	214	0	214	0		
	link outform ENTER	1	0	1	0	1	0		
o/p ISPP	write buffer	20	55	20	55	20	55	11	18
	close	20	65	16	56	16	56	7	0
	reset state	20	75	15	67	15	67	5	0
i/p ISPP	read buffer	11	82	11	82	11	82	2	0
	close	11	82	11	82	11	82	2	0
	reset state	11	91	11	91	11	91	1	0
SETUP	set up receive	107	88	13	0	13	0		
	set up send	78	78	21	19	18	6		
	set up send with reply	78	45	44	5	44	5		
SINMAN	initialise SINMAN	13	0	13	0	13	0		
	start up	13	92	1	0	1	0		
	dynamic decrement	45	93	7	57	7	57		
	seg from sin	45	80	8	0	8	0		
	cap from sin	45	82	17	53	8	0		
	remove	45	98	22	50	14	21		
	restart	45	98	1	0	1	0		
	change size	45	89	12	33	8	0		
	make swc	45	98	9	89	1	0		
Peripherals									
	write Burroughs	3	67	3	67	3	67		
	write CDC	3	67	3	67	3	67		
	write to printer	4	25	4	25	4	25		
	write to punch	3	33	3	33	3	33		
	set clock interval	2	50	2	50	2	50		
	read from p/tape reader	3	33	3	33	3	33		
	read from teletype	8	13	8	13	8	13		
	write to teletype	9	11	9	11	9	11		

Appendix E

This appendix contains the full results of the audit (see Section 6.1). The table number quoted for each set of results is the table in Chapter 6 to which that set of results corresponds. For Tables 6.5, 6.6 and 6.7 the only figures listed are those which differ from the figures for Tables 6.4, 6.5 and 6.6 respectively. The figures given for process PRINTER* are those for the PRINTER process without the use of the STOREMAN protected procedure as a gate-keeper (see Section 6.1.6). (Note: The figures for PTPDESPOOL are as for LPRDESPOOL and those for PUNCH and READER are as for PRINTER.)

1. Results corresponding to Table 6.3

Process	Service	Domain	|I|	|M|	d	|I| p	|M| p	d p	|I| u	|M| u	d u	|I| v	|M| v	d v	|I| w	|M| w	d w
BRSDISC	initialise BRSDISC	MAIN	20	6	.70	7	5	.29	3	1	.67	2	0	1.00	8	0	1.00
		SETUP	18	9	.50	7	4	.43	1	1	0.00	1	0	1.00	9	4	.55
		ECPROC	20	14	.30	16	12	.25	2	0	1.00	0	0	0.00	2	2	0.00
	read map	MAIN	20	7	.65	7	3	.57	3	1	.67	2	1	.50	8	2	.75
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	write map	MAIN	20	7	.65	7	4	.43	3	1	.67	2	0	1.00	8	2	.75
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
CDCDISC	initialise CDCDISC	MAIN	19	6	.68	6	4	.33	3	1	.67	2	1	.50	8	0	1.00
		SETUP	18	9	.50	7	4	.43	1	1	0.00	1	0	1.00	9	4	.55
		ECPROC	21	13	.38	15	11	.27	2	0	1.00	0	0	0.00	4	2	.50
	read disc	MAIN	19	6	.68	6	2	.67	3	1	.67	2	1	.50	8	2	.75
		ECPROC	20	7	.65	14	5	.64	2	0	1.00	0	0	0.00	4	2	.50
	write disc	MAIN	19	6	.68	6	3	.50	3	1	.67	2	0	1.00	8	2	.75
		ECPROC	20	7	.65	14	5	.64	2	0	1.00	0	0	0.00	4	2	.50
CLOCK	initialise CLOCK	MAIN	22	8	.64	7	6	.14	3	1	.67	1	0	1.00	11	1	.91
		SETUP	19	9	.53	7	4	.43	1	1	0.00	1	0	1.00	10	4	.60
		ECPROC	21	14	.33	16	12	.25	2	0	1.00	0	0	0.00	3	2	.33
	read date & time	MAIN	22	9	.59	7	3	.57	3	1	.67	1	0	1.00	11	5	.55
		ECPROC	21	11	.48	16	9	.44	2	0	1.00	0	0	0.00	3	2	.33
ENSURER	initialise ENSURER	MAIN	50	5	.90	22	5	.77	7	0	1.00	6	0	1.00	15	0	1.00
		SETUP	35	26	.26	16	13	.19	2	2	0.00	2	1	.50	15	10	.33
		ECPROC	21	13	.38	16	11	.31	2	0	1.00	0	0	0.00	3	2	.33
	ensure file directory	MAIN	50	7	.86	22	3	.86	7	1	.86	6	0	1.00	15	3	.80
		ECPROC	21	11	.48	16	9	.44	2	0	1.00	0	0	0.00	3	2	.33
	ensure & remove	MAIN	52	9	.83	24	5	.79	7	1	.86	6	0	1.00	15	3	.80
		SETUP	45	36	.20	23	20	.13	2	2	0.00	2	1	.50	18	13	.33
		ECPROC	21	13	.38	16	11	.31	2	0	1.00	0	0	0.00	3	2	.33
		SINMAN	51	8	.84	26	7	.73	4	1	.75	4	0	1.00	17	0	1.00
	delete	MAIN	50	7	.86	22	4	.82	7	1	.86	6	0	1.00	15	2	.87
		ECPROC	21	11	.48	16	9	.44	2	0	1.00	0	0	0.00	3	2	.33
LPRDESPOOL	initialise LPRDESPOOL	MAIN	80	14	.83	16	6	.69	7	0	1.00	7	1	.86	50	7	.86

		SETUP	87	53	.39	32	29	.09	5	3	.40	5	2	.60	45	19	.58
		ECPROC	47	14	.70	16	11	.31	4	0	1.00	2	0	1.00	25	3	.88
		PRLGARB	46	8	.83	8	5	.38	5	1	.80	4	1	.75	29	1	.97
		STOREMAN	64	5	.92	24	4	.83	6	1	.83	8	0	1.00	26	0	1.00
		MAKEENTER	66	28	.58	10	7	.30	4	0	1.00	4	1	.75	48	20	.58
		IOC	55	12	.78	16	7	.56	8	1	.88	7	0	1.00	24	4	.83
		SINMAN	94	31	.67	26	7	.73	6	2	.67	6	1	.83	56	21	.63
		MFD	71	18	.75	15	7	.53	8	0	1.00	7	0	1.00	41	11	.73
		SPOOLDIR	62	12	.81	15	7	.53	8	1	.88	7	1	.86	32	2	.94
		o/p ISPP	41	4	.90	3	3	0.00	5	0	1.00	3	0	1.00	30	1	.97
	spooled	MAIN	75	9	.88	14	4	.71	7	0	1.00	7	0	1.00	47	5	.89
		SETUP	44	10	.77	7	4	.43	3	1	.67	3	0	1.00	31	5	.84
		ECPROC	47	14	.70	16	11	.31	4	0	1.00	2	0	1.00	25	3	.88
		PRLGARB	46	8	.83	8	5	.38	5	1	.80	4	1	.75	29	1	.97
		STOREMAN	64	4	.94	24	3	.88	6	0	1.00	8	1	.88	26	0	1.00
		IOC	51	9	.82	13	4	.69	8	1	.88	7	1	.86	23	3	.87
		SINMAN	76	11	.86	26	5	.81	6	1	.83	6	3	.50	38	2	.95
		SPOOLDIR	62	8	.87	15	5	.67	8	1	.88	7	0	1.00	32	2	.94
		o/p ISPP	41	8	.80	3	3	0.00	5	1	.80	3	0	1.00	30	4	.87
MODULE	initialise MODULE	MAIN	24	11	.54	7	5	.29	5	3	.40	2	1	.50	10	2	.80
		SETUP	24	15	.38	8	5	.38	3	3	0.00	2	1	.50	11	6	.45
		ECPROC	20	13	.35	16	11	.31	2	0	1.00	0	0	0.00	2	2	0.00
	reserve space	MAIN	24	6	.75	7	3	.57	5	1	.80	2	0	1.00	10	2	.80
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	extend chain	MAIN	24	9	.63	7	4	.43	5	2	.60	2	0	1.00	10	3	.70
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	contract chain	MAIN	24	6	.75	7	3	.57	5	1	.80	2	0	1.00	10	2	.80
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	delete chain	MAIN	24	6	.75	7	3	.57	5	1	.80	2	0	1.00	10	2	.80
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	bad block	MAIN	24	9	.63	7	4	.43	5	2	.60	2	0	1.00	10	3	.70
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	disc address	MAIN	24	6	.75	7	3	.57	5	1	.80	2	0	1.00	10	2	.80
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	ensure map on disc	MAIN	24	9	.63	7	4	.43	5	2	.60	2	0	1.00	10	3	.70
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	checkmap	MAIN	24	6	.75	7	3	.57	5	1	.80	2	0	1.00	10	2	.80
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	restore map	MAIN	24	9	.63	7	3	.57	5	2	.60	2	1	.50	10	3	.70
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	init map	MAIN	24	9	.63	7	4	.43	5	2	.60	2	0	1.00	10	3	.70
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	free blocks	MAIN	24	6	.75	7	3	.57	5	1	.80	2	0	1.00	10	2	.80
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	chain length	MAIN	24	6	.75	7	3	.57	5	1	.80	2	0	1.00	10	2	.80
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
	restart	MAIN	24	8	.67	7	3	.57	5	2	.60	2	0	1.00	10	3	.70
		ECPROC	20	11	.45	16	9	.44	2	0	1.00	0	0	0.00	2	2	0.00
PRINTER	initialise PRINTER	MAIN	32	6	.81	7	5	.29	5	1	.80	3	0	1.00	17	0	1.00
		SETUP	31	9	.71	7	4	.43	3	1	.67	3	0	1.00	18	4	.78
		ECPROC	34	15	.56	16	12	.25	4	0	1.00	2	0	1.00	12	3	.75
		PRLGARB	34	8	.76	8	5	.38	5	1	.80	5	1	.80	16	1	.94
	print line	MAIN	31	7	.77	7	4	.43	5	1	.80	3	0	1.00	16	2	.88
		ECPROC	33	11	.67	16	9	.44	4	0	1.00	2	0	1.00	11	2	.82

	print document	MAIN	32	13	.59	7	4	.43	5	1	.80	3	0	1.00	17	8	.53
		ECPROC	33	14	.58	16	11	.31	4	0	1.00	2	0	1.00	11	3	.73
		PRLGARB	33	8	.76	8	5	.38	5	1	.80	5	1	.80	15	1	.93
		STOREMAN	53	17	.68	26	10	.62	6	1	.83	8	4	.50	13	2	.85
		SINMAN	63	15	.76	26	9	.65	6	1	.83	6	3	.50	25	2	.92
		SETUP	57	36	.37	24	21	.13	4	2	.50	4	1	.75	25	12	.52
RESTART	restart	MAIN	110	36	.67	32	15	.53	9	1	.89	7	2	.71	62	18	.71
		SETUP	83	48	.42	24	21	.13	5	3	.40	4	1	.75	50	23	.54
		ECPROC	49	17	.65	16	14	.13	4	0	1.00	2	0	1.00	27	3	.89
		PRLGARB	47	8	.83	8	5	.38	5	1	.80	4	1	.75	30	1	.97
		STOREMAN	65	9	.86	24	4	.83	6	1	.83	7	1	.86	28	3	.89
		SINMAN	82	15	.82	28	10	.64	7	2	.71	6	0	1.00	41	3	.93
	allocate USER process	MAIN	102	15	.85	26	3	.88	9	1	.89	7	0	1.00	60	11	.82
		ECPROC	49	11	.78	16	9	.44	4	0	1.00	2	0	1.00	27	2	.93
	user process finished	MAIN	102	9	.91	26	3	.88	9	1	.89	7	0	1.00	60	5	.92
		ECPROC	49	11	.78	16	9	.44	4	0	1.00	2	0	1.00	27	2	.93
RSM	initialise RSM	MAIN	24	9	.63	12	8	.33	3	1	.67	1	0	1.00	8	0	1.00
		SETUP	18	9	.50	7	4	.43	1	1	0.00	1	0	1.00	9	4	.56
		ECPROC	20	15	.25	15	13	.13	2	0	1.00	0	0	0.00	3	2	.33
	segment outform at mc level	MAIN	41	13	.68	19	7	.63	6	2	.67	2	1	.50	14	3	.79
		SETUP	20	11	.45	8	5	.38	1	1	0.00	2	1	.50	9	4	.56
		ECPROC	20	14	.30	15	12	.20	2	0	1.00	0	0	0.00	3	2	.33
	ensure mcaddr ok	MAIN	41	11	.73	19	6	.68	6	2	.67	2	0	1.00	14	3	.79
		SETUP	20	11	.45	8	5	.38	1	1	0.00	2	1	.50	9	4	.56
		ECPROC	20	14	.30	15	12	.20	2	0	1.00	0	0	0.00	3	2	.33
	outform	MAIN	41	10	.76	19	5	.74	6	2	.67	2	0	1.00	14	3	.79
		SETUP	20	11	.45	8	5	.38	1	1	0.00	2	1	.50	9	4	.56
		ECPROC	20	13	.35	15	11	.27	2	0	1.00	0	0	0.00	3	2	.33
	change segment size	MAIN	41	11	.73	19	6	.68	6	2	.67	2	0	1.00	14	3	.79
		SETUP	20	11	.45	8	5	.38	1	1	0.00	2	1	.50	9	4	.56
		ECPROC	20	14	.30	15	12	.20	2	0	1.00	0	0	0.00	3	2	.33
	delete	MAIN	41	10	.76	19	5	.74	6	2	.67	2	0	1.00	14	3	.79
		SETUP	20	11	.45	8	5	.38	1	1	0.00	2	1	.50	9	4	.56
		ECPROC	20	13	.35	15	11	.27	2	0	1.00	0	0	0.00	3	2	.33
	move window	MAIN	41	12	.71	19	6	.68	6	2	.67	2	1	.50	14	3	.79
		SETUP	20	11	.45	8	5	.38	1	1	0.00	2	1	.50	9	4	.56
		ECPROC	20	14	.30	15	12	.20	2	0	1.00	0	0	0.00	3	2	.33
	initialise:2nd part	MAIN	38	8	.79	18	5	.72	5	2	.60	1	0	1.00	14	1	.93
		SETUP	30	21	.30	13	10	.23	2	2	0.00	1	0	1.00	14	9	.36
		ECPROC	20	13	.35	15	11	.27	2	0	1.00	0	0	0.00	3	2	.33
	new thing	MAIN	41	6	.85	19	3	.84	6	1	.83	2	0	1.00	14	2	.86
		ECPROC	19	8	.58	14	6	.57	2	0	1.00	0	0	0.00	3	2	.33
TELETYPE	initialise TELETYPE	MAIN	35	10	.71	8	5	.38	6	3	.50	3	0	1.00	18	2	.89
		SETUP	33	11	.67	7	4	.43	4	2	.50	3	0	1.00	19	5	.74
		ECPROC	34	15	.56	16	12	.25	4	0	1.00	2	0	1.00	12	3	.75
		PRLGARB	34	8	.76	8	5	.38	5	1	.80	4	1	.75	17	1	.94
	output line	MAIN	34	8	.76	8	5	.38	6	1	.83	3	0	1.00	17	2	.88
		ECPROC	33	11	.67	16	9	.44	4	0	1.00	2	0	1.00	11	2	.82
	input line	MAIN	34	8	.76	8	5	.38	6	1	.83	3	0	1.00	17	2	.88
		ECPROC	33	11	.67	16	9	.44	4	0	1.00	2	0	1.00	11	2	.82
	output document	MAIN	35	14	.60	8	5	.38	6	1	.83	3	0	1.00	18	8	.56
		ECPROC	33	14	.58	16	11	.31	4	0	1.00	2	0	1.00	11	3	.73
		PRLGARB	33	8	.76	8	5	.38	5	1	.80	5	1	.80	15	1	.93
		STOREMAN	53	17	.68	26	10	.62	6	1	.83	8	4	.50	13	2	.85

		SINMAN	63	15	.76	26	9	.65	6	1	.83	6	3	.50	25	2	.92
		SETUP	57	36	.37	24	21	.13	4	2	.50	4	1	.75	25	12	.52
	input document	MAIN	35	14	.60	8	5	.38	6	1	.83	3	0	1.00	18	8	.56
		ECPROC	33	14	.58	16	11	.31	4	0	1.00	2	0	1.00	11	3	.73
		PRLGARB	33	8	.76	8	5	.38	5	1	.80	5	1	.80	15	1	.93
		STOREMAN	53	17	.68	26	10	.62	6	1	.83	8	4	.50	13	2	.85
		SINMAN	63	15	.76	26	9	.65	6	1	.83	6	3	.50	25	2	.92
		SETUP	57	36	.37	24	21	.13	4	2	.50	4	1	.75	25	12	.52
	logon ok	MAIN	34	8	.76	8	5	.38	6	1	.83	3	0	1.00	17	2	.88
		ECPROC	33	11	.67	16	9	.44	4	0	1.00	2	0	1.00	11	2	.82
	logon system full	MAIN	34	8	.76	8	5	.38	6	1	.83	3	0	1.00	17	2	.88
		ECPROC	33	11	.67	16	9	.44	4	0	1.00	2	0	1.00	11	2	.82
	input when idle	MAIN	34	8	.76	8	5	.38	6	1	.83	3	0	1.00	17	2	.88
		ECPROC	33	11	.67	16	9	.44	4	0	1.00	2	0	1.00	11	2	.82
USER	initialise USER process	MAIN	76	12	.84	16	8	.50	9	1	.89	7	1	.86	44	2	.95
		SETUP	75	39	.48	23	20	.13	5	3	.40	4	1	.75	43	15	.65
		ECPROC	50	14	.72	16	11	.31	4	0	1.00	2	0	1.00	28	3	.89
		PRLGARB	48	8	.83	8	5	.38	5	1	.80	5	1	.80	30	1	.97
		SINMAN	88	23	.74	26	7	.73	6	2	.67	6	1	.83	50	13	.74
		MFD	65	10	.85	15	7	.53	8	0	1.00	7	0	1.00	35	3	.91
		MAKEENTER	54	14	.74	7	4	.43	4	0	1.00	4	1	.75	39	9	.77
	run user process	MAIN	95	24	.75	22	8	.64	9	2	.78	7	2	.71	57	12	.79
		SETUP	68	24	.65	20	11	.45	4	2	.50	4	1	.75	40	10	.75
		ECPROC	58	15	.74	22	11	.50	4	0	1.00	2	0	1.00	30	4	.87
		PRLGARB	56	8	.86	14	5	.64	5	1	.80	5	1	.80	32	1	.97
		SINMAN	96	23	.76	32	5	.84	6	2	.67	6	3	.50	52	13	.75
		MFD	82	17	.79	21	5	.76	8	0	1.00	7	0	1.00	46	12	.74
		STOREMAN	83	17	.80	30	5	.83	6	1	.83	8	2	.75	39	9	.77
		MAKEENTER	62	14	.77	13	4	.69	4	0	1.00	4	1	.75	41	9	.78
		IOC	62	22	.65	19	10	.47	8	2	.75	7	0	1.00	28	10	.64
		o/p ISPP	48	8	.83	6	3	.50	5	1	.80	3	0	1.00	34	4	.88
		i/p ISPP	48	8	.83	6	3	.50	5	1	.80	3	0	1.00	34	4	.88
		LINKER	63	10	.84	16	5	.69	7	3	.57	7	0	1.00	33	2	.94
		LOGON	47	8	.83	8	7	.13	4	0	1.00	3	0	1.00	32	1	.97
VSM	initialise VSM	MAIN	41	6	.85	17	5	.71	7	1	.86	5	0	1.00	12	0	1.00
		SETUP	26	17	.35	11	8	.27	2	2	0.00	1	0	1.00	12	7	.42
		ECPROC	21	13	.38	16	11	.31	2	0	1.00	0	0	0.00	3	2	.33
	sin to mcaddr	MAIN	43	11	.74	19	6	.68	7	3	.57	5	0	1.00	12	2	.83
		SETUP	45	36	.20	23	20	.13	2	2	0.00	2	1	.50	18	13	.28
		ECPROC	21	13	.38	16	11	.31	2	0	1.00	0	0	0.00	3	2	.33
		SINMAN	51	7	.86	26	7	.73	4	0	1.00	4	0	1.00	17	0	1.00
	mcaddr to sin	MAIN	41	6	.85	17	3	.82	7	1	.86	5	0	1.00	12	2	.83
		ECPROC	21	11	.48	16	9	.44	2	0	1.00	0	0	0.00	3	2	.33
	ensure sin	MAIN	41	9	.78	17	3	.82	7	2	.71	5	0	1.00	12	4	.67
		ECPROC	21	11	.48	16	9	.44	2	0	1.00	0	0	0.00	3	2	.33
	disc addr to mcaddr	MAIN	41	9	.78	17	4	.76	7	2	.71	5	0	1.00	12	3	.75
		ECPROC	21	11	.48	16	9	.44	2	0	1.00	0	0	0.00	3	2	.33
	close window	MAIN	43	12	.72	19	6	.68	7	2	.71	5	1	.80	12	3	.75
		SETUP	45	36	.20	23	20	.13	2	2	0.00	2	1	.50	18	13	.28
		ECPROC	21	13	.38	16	11	.31	2	0	1.00	0	0	0.00	3	2	.33
		SINMAN	51	8	.84	26	7	.73	4	1	.75	4	0	1.00	17	0	1.00
	delete mcaddr	MAIN	43	10	.77	19	6	.68	7	1	.86	5	1	.80	12	2	.83
		SETUP	45	36	.20	23	20	.13	2	2	0.00	2	1	.50	18	13	.28
		ECPROC	21	13	.38	16	11	.31	2	0	1.00	0	0	0.00	3	2	.33

		SINMAN	51	8	.84	26	7	.73	4	1		4	0	1.00	17	0	1.00
PRINTER*	initialise PRINTER*	MAIN	57	7	.88	29	6	.79	7	1		8	0	1.00	13	0	1.00
		SETUP	31	22	.29	14	11	.21	2	2	0	2	1	.50	13	8	.38
		ECPROC	22	15	.32	16	12	.25	2	0	1.	0	0	0.00	4	3	.25
		PRLGARB	21	8	.62	8	5	.38	3	1		1	1	0.00	9	1	.89
	print line	MAIN	55	7	.87	27	4	.85	7	1	.	8	0	1.00	13	2	.85
		ECPROC	22	11	.50	16	9	.44	2	0	1.	0	0	0.00	4	2	.50
	print document	MAIN	57	19	.67	29	10	.66	7	2	.7	8	4	.50	13	3	.77
		SETUP	45	26	.42	23	20	.13	2	2	0.00	2	1	.50	18	3	.83
		ECPROC	22	12	.45	16	11	.31	2	0	1.00	0	0	0.00	4	1	.75
		PRLGARB	21	8	.62	8	5	.38	3	1	.67	1	1	0.00	9	1	.89
		SINMAN	51	15	.71	26	9	.65	4	1	.75	4	1	.75	17	4	.76

2. Results corresponding to Tables 6.4 - 6.7

			Table 6.4			Table 6.5			Table 6.6			Table 6.7		
Process	Service	Domain	I	M	d p	I	M	d p	I	M	d p	I	M	d p
BRSDISC	initialise BRSDISC	MAIN	6	5	.17									
		SETUP	4	4	0.00									
		ECPROC	16	12	.25									
	read map	MAIN	4	3	.25									
		ECPROC	16	9	.44									
	write map	MAIN	4	4	0.00									
		ECPROC	16	9	.44									
CDCDISC	initialise CDCDISC	MAIN	5	4	.20									
		SETUP	4	4	0.00									
		ECPROC	15	11	.27									
	read disc	MAIN	3	2	.33									
		ECPROC	14	5	.64									
	write disc	MAIN	3	3	0.00									
		ECPROC	14	5	.64									
CLOCK	initialise CLOCK	MAIN	6	6	0.00									
		SETUP	4	4	0.00									
		ECPROC	16	12	.25									
	read date & time	MAIN	4	3	.25									
		ECPROC	16	9	.44									
ENSURER	initialise ENSURER	MAIN	7	5	.29									
		SETUP	13	13	0.00									
		ECPROC	16	11	.31									
	ensure file directory	MAIN	5	3	.40									
		ECPROC	16	9	.44									
	ensure & remove	MAIN	6	5	.17									
		SETUP	20	20	0.00									
		ECPROC	16	11	.31									
		SINMAN	12	7	.42	7	7	0.00						
	delete	MAIN	5	4	.20									
		ECPROC	16	9	.44									
LPRDESPOOL	initialise	MAIN	7	6	.14									

		SETUP	29	29	0.00		
		ECPROC	16	11	.31		
		PRLGARB	5	5	0.00		
		STOREMAN	9	4	.56	4	4 0.00
		MAKEENTER	8	7	.13	7	7 0.00
		IOC	9	7	.22	7	7 0.00
		SINMAN	12	7	.42	7	7 0.00
		MFD	10	7	.30	7	7 0.00
		SPOOLDIR	10	7	.30	7	7 0.00
	spooled	o/p ISPP	3	3	0.00		
		MAIN	4	4	0.00		
		SETUP	4	4	0.00		
		ECPROC	16	11	.31		
		PRLGARB	5	5	0.00		
		STOREMAN	8	3	.63	3	3 0.00
		IOC	6	4	.33	4	4 0.00
		SINMAN	10	5	.50	5	5 0.00
		SPOOLDIR	8	5	.38	5	5 0.00
MODULE	initialise MODULE	o/p ISPP	3	3	0.00		
		MAIN	6	5	.17		
		SETUP	5	5	0.00		
		ECPROC	16	11	.31		
	reserve space	MAIN	4	3	.25		
		ECPROC	16	9	.44		
	extend chain	MAIN	4	4	0.00		
		ECPROC	16	9	.44		
	contract chain	MAIN	4	3	.25		
		ECPROC	16	9	.44		
	delete chain	MAIN	4	3	.25		
		ECPROC	16	9	.44		
	bad block	MAIN	4	4	0.00		
		ECPROC	16	9	.44		
	disc address	MAIN	4	3	.25		
		ECPROC	16	9	.44		
	ensure map on disc	MAIN	4	4	0.00		
		ECPROC	16	9	.44		
	checkmap	MAIN	4	3	.25		
		ECPROC	16	9	.44		
	restore map	MAIN	4	3	.25		
		ECPROC	16	9	.44		
	init map	MAIN	4	4	0.00		
		ECPROC	16	9	.44		
	free blocks	MAIN	4	3	.25		
		ECPROC	16	9	.44		
	chain length	MAIN	4	3	.25		
		ECPROC	16	9	.44		
	restart	MAIN	4	3	.25		
		ECPROC	16	9	.44		
PRINTER	initialise PRINTER	MAIN	6	5	.17		
		SETUP	4	4	0.00		
		ECPROC	16	12	.25		
	print line	PRLGARB	5	5	0.00		
		MAIN	4	4	0.00		

	print document	MAIN	4	4	0.00	
		ECPROC	16	11	.31	
		PRLGARB	5	5	0.00	
		STOREMAN	10	10	0.00	
		SINMAN	12	9	.25	9 9 0.00
		SETUP	21	21	0.00	
RESTART	restart	MAIN	15	15	0.00	
		SETUP	21	21	0.00	
		ECPROC	16	14	.13	
		PRLGARB	5	5	0.00	
		STOREMAN	9	4	.56	4 4 0.00
		SINMAN	14	10	.29	10 10 0.00
	allocate USER process	MAIN	3	3	0.00	
		ECPROC	16	9	.44	
	user process finished	MAIN	3	3	0.00	
		ECPROC	16	9	.44	
RSM	initialise RSM	MAIN	9	8	.11	
		SETUP	4	4	0.00	
		ECPROC	15	13	.13	
	segment outform at mc level	MAIN	7	7	0.00	
		SETUP	5	5	0.00	
		ECPROC	15	12	.20	
	ensure mcaddr ok	MAIN	7	6	.14	
		SETUP	5	5	0.00	
		ECPROC	15	12	.42	
	outform	MAIN	7	5	.29	
		SETUP	5	5	0.00	
		ECPROC	15	11	.27	
	change segment size	MAIN	7	6	.14	
		SETUP	5	5	0.00	
		ECPROC	15	12	.42	
	delete	MAIN	7	5	.29	
		SETUP	5	5	0.00	
		ECPROC	15	11	.27	
	move window	MAIN	7	6	.14	
		SETUP	5	5	0.00	
		ECPROC	15	12	.42	
	initialise:2nd part	MAIN	6	5	.17	
		SETUP	10	10	0.00	
		ECPROC	15	11	.27	
	new thing	MAIN	7	3	.57	
		ECPROC	14	6	.57	
TELETYPE	initialise TELETYPE	MAIN	7	5	.29	
		SETUP	4	4	0.00	
		ECPROC	16	12	.25	
		PRLGARB	5	5	0.00	
	output line	MAIN	5	5	0.00	
		ECPROC	16	9	.44	
	input line	MAIN	5	5	0.00	
		ECPROC	16	9	.44	
	output document	MAIN	5	5	0.00	
		ECPROC	16	11	.31	
		PRLGARB	5	5	0.00	
		STOREMAN	10	10	0.00	

		SINMAN	12	9	.25	9	9	0.00					
		SETUP	21	21	0.00								
	input document	MAIN	5	5	0.00								
		ECPROC	16	11	.31								
		PRLGARB	5	5	0.00								
		STOREMAN	10	10	0.00								
		SINMAN	12	9	.25	9	9	0.00					
		SETUP	21	21	0.00								
	logon ok	MAIN	5	5	0.00								
		ECPROC	16	9	.44								
	logon system full	MAIN	5	5	0.00								
		ECPROC	16	9	.44								
	input when idle	MAIN	5	5	0.00								
		ECPROC	16	9	.44								
USER	initialise USER process	MAIN	8	8	0.00								
		SETUP	20	20	0.00								
		ECPROC	16	11	.31								
		PRLGARB	5	5	0.00								
		SINMAN	12	7	.42	7	7	0.00					
		MFD	10	7	.30	7	7	0.00					
		MAKEENTER	5	4	.20	4	4	0.00					
	run user process	MAIN	10	8	.20				8	8	0.00		
		SETUP	17	11	.35				11	11	0.00	12	11 .08
		ECPROC	22	11	.50				16	11	.31	17	11 .35
		PRLGARB	11	5	.55				5	5	0.00	6	5 .17
		SINMAN	16	5	.69	11	5	.55	5	5	0.00	6	5 .17
		MFD	14	5	.64	11	5	.55	5	5	0.00	6	5 .17
		STOREMAN	15	5	.67	11	5	.55	5	5	0.00	6	5 .17
		MAKEENTER	11	4	.64	10	4	.60	4	4	0.00	5	4 .20
		IOC	12	10	.17	11	10	.09					
		o/p ISPP	6	3	.50				3	3	0.00		
		i/p ISPP	6	3	.50				3	3	0.00	4	3 .25
		LINKER	11	5	.55				5	5	0.00	6	5 .17
		LOGON	7	7	0.00								
VSM	initialise VSM	MAIN	8	5	.38								
		SETUP	8	8	0.00								
		ECPROC	16	11	.31								
	sin to mcaddr	MAIN	7	6	.17								
		SETUP	20	20	0.00								
		ECPROC	16	11	.31								
		SINMAN	12	7	.42	7	7	0.00					
	mcaddr to sin	MAIN	6	3	.50								
		ECPROC	16	9	.44								
	ensure sin	MAIN	6	3	.50								
		ECPROC	16	9	.44								
	disc addr to mcaddr	MAIN	6	4	.33								
		ECPROC	16	9	.44								
	close window	MAIN	7	6	.17								
		SETUP	20	20	0.00								
		ECPROC	16	11	.31								
		SINMAN	12	7	.42	7	7	0.00					
	delete mcaddr	MAIN	7	6	.17								
		SETUP	20	20	0.00								
		ECPROC	16	11	.31								

PRINTER*	initialise PRINTER*	SINMAN	12	7	.42	7	7	0.00
		MAIN	14	6	.57			
		SETUP	11	11	0.00			
		ECPROC	16	12	.25			
		PRLGARB	5	5	0.00			
	print line	MAIN	11	4	.64			
		ECPROC	16	9	.44			
	print document	MAIN	11	10	.09			
		SETUP	20	20	0.00			
		ECPROC	16	11	.31			
		PRLGARB	5	5	0.00			
		SINMAN	12	9	.25	9	9	0.00

Appendix F

This appendix lists the CAP Operating System primary services (see Section 7.2.1). Further information about these services can be found in the CAP Operating System Manual [Herbert 1978(c)].

Protected procedure	Primary service
DIRMAN	retrieve
	remove
	preserve
	alter access
	examine
	file details
	file examine
ECPROC	wait event
	send message without segment
	send message with segment
	receive message without segment
	receive message with segment
	receive message with reply
	return message
	reserve segment
	release segment
	clear fault
	cause fault
	return fault
	process information

FAULT	fault message
IOC	stream request
	despool request
	send document
ISPP	write buffer
	read buffer
	close
	endof
	state
	reset state
MAKEENTER	make capability segment
	make enter capability
SETUP	set up receive
	set up send
	set up send with reply
	set up reply
	set up reply with store
SSPP	write buffer
	read buffer
	close
	endof
	state
	reset state
	backspace buffer

STOREMAN	ensure ok
	outform
	change size
	new segment
	capability information
	get size amd access
	open window
	move window
	close window
	details

Appendix G

The full results of the experiments described in Chapter 7 are given in this appendix. The programs used for the instruction timing experiments are also included.

G.1 Counting domain switches and messages

The results given below are the averages over five runs.

G.1.1 Results from runs of the Algol 68C compiler

Protected procedure	Primary service	Times called
DIRMAN	retrieve	6
	preserve	1
IOC	stream request	6
ISPP	write buffer	7
	read buffer	1
	endof	2
	reset state	2
SSPP	write buffer	4697
	read buffer	7055
	close	6
	endof	14109
STOREMAN	change size	2
	new segment	2

Total primary service calls	=	25896
Consequential ENTERs	=	2810 (includes 53 simulated ENTERs)
Consequential ENTERs/primary service call	=	0.11
Total ENTERs	=	29780
Source program size	=	1750 lines
Object code segment size	=	3772 words
Virtual memory faults	=	73
Messages sent	=	307
Consequential messages	=	380
Consequential messages/primary service call	=	0.01

G.1.2 Results from runs of Genesis

Protected procedure	Primary service	Times called
DIRMAN	retrieve	48
	preserve	1
ECPROC	process information	1
IOC	stream request	2
	despool request	1
ISPP	write buffer	1
	read buffer	1
	endof	2
	reset state	2

SSPP	write buffer	44
	read buffer	1403
	close	2
	endof	2805
STOREMAN	change size	3
	new segment	19
	get size and access	44
	open window	58
	move window	45
	close window	58

Total primary service calls	=	4540
Consequential ENTERs	=	6135 (includes 127 simulated ENTERs)
Consequential ENTERs/primary service call	=	1.35
Total ENTERs	=	10706
Virtual memory faults	=	117
Messages sent	=	774
Consequential messages	=	891
Consequential messages/primary service call	=	0.20

G.2 Instruction timing

G.2.1 System used for experiments

The timing experiments were run using a simple system the programs of which were written in the CAP's assembly language. The Master Co-ordinator of the system is listed in Figure G.1.

```
|       Master Coordinator for timing experiments
        BS      R1 M15    G0-.-1
        ESP     R2 M1                 |enter level-1 process
        JNLT    R2 M15    -2          |loop if fault detected
        EC      0                     |go into wait state
G0:     #X1                           |general address of G0 word 0
```

Figure G.1

G.2.2 Load from store (BS), register to store (SB) and test and count (TCS) instructions

The programs for the level-1 process for timing the BS,SB and TCS instructions are given in Figures G.2, G.3 and G.4 respectively.

```
|       Program for timing BS instruction
        MODNS     M15     COUNT-.-1
        BN        R1                  |load count
        BS        R2 M15  G0-.-1
LOOP:   BS        R2 M15  G0-.-1
        TCS       R1 M15  LOOP-.-1    |repeat count times
        EC                            |enter Master Coordinator
G0:     #X1
COUNT:9999999
```

Figure G.2

```
|       Program for timing SB instruction
        MODNS       M15     COUNT-.-1
        BN      R1                      |load count
        BS      R2 M15      GO-.-1
LOOP:   SB      R2 M15      GO-.-1
        TCS     R1 M15      LOOP-.-1    |repeat count times
        EC                              |enter Master Coordinator
GO:     #X1
COUNT:9999999
```

Figure G.3

```
|       Program for timing balance of BS and SB programs
        MODNS       M15     COUNT-.-1
        BN      R1                      |load count
        BS      R2 M15      GO-.-1
LOOP:   TCS     R1 M15      LOOP-.-1    |repeat count times
        EC                              |enter Master Coordinator
GO:     #X1
COUNT:9999999                           |set to 999999 and 1999999
                                        |for timing TCS
```

Figure G.4

The results of these timing experiments, averaged over ten runs, are given below:

10 million iterations	BS program	SB program	balance program	BS - balance	SB - balance
time secs.	52.80	49.92	31.44	21.36	18.48
core reads	1	1	1	0	0
core writes	61	62	61	0	1

Time for 1*BS instruction = 2.136 = 2.14μs
====
Time for 1*SB instruction = 1.848 = 1.85μs
====

The single core write during 10 million iterations of the SB instruction could have been removed by a simple modification to the SB and balance programs. However, as it was insignificant compared to the 10 million writes to the slave stores it was ignored.

Time for 1 million * balance program = 6.96 seconds

Time for 2 million * balance program = 9.76 seconds

Time for 1 * TCS instruction = 2.80μs
====

G.2.3 ENTER + RETURN

For measuring the time taken to enter and return from a protected procedure the program of the level-1 process was as given in Figure G.5. The program of the protected procedure which was repeatedly entered consisted of the single instruction, RETURN.

```
!       Program for timing ENTER + RETURN
        MODNS       M15     COUNT-.-1
        BN      R1                      !load count
        BS      R2 M15      G0-.-1
LOOP:   ENTER       M2                  !enter protected procedure
        TCS     R1 M15      LOOP-.-1    !repeat count times
        EC                              !enter Master Coordinator
G0:     #X1
COUNT:  99999                           !and 199999
```

Figure G.5

To measure the time taken to enter a protected procedure with one capability argument and to return from it with one capability result the main domain program was as in Figure G.6 and that of the protected procedure was as given in Figure G.7.

```
|       Program for timing ENTER + RETURN with capability argument
|        and result
        MAKEIND  R3 1                 |create N capability segment
        MODNS       M15    COUNT-.-1
        BN       R1                   |load count
        BS       R2 M15    G0-.-1
        BS       R3 M15    I0-.-1
        BS       R4 M15    N0-.-1
        BS       R5 M15    I0-.-1
        BS       R6 M15    A0-.-1
        BS       R7 M15    R0-.-1
        BS       R8 M15    R0-.-1
LOOP:   MOVECAP  R3 M4                |capability argument to N0
        ENTER       M2                |enter protected procedure
        MOVECAP  R4 M8                |capability result from N0
        TCS      R1 M15    LOOP-.-1   |repeat count times
        EC                            |enter Master Coordinator
G0:     #X1
A0:     #X2
N0:     #X3
I0:     #X5
R0:     #X6
COUNT:99999                           |and 199999
```

Figure G.6

```
|       Program for protected procedure of level-1 process
        MOVECAP  R6 M5              |capability argument from A0
        MOVECAP  R7 M6              |capability result to A0
        RETURN
```

Figure G.7

The results, averaged over ten runs, of these experiments are given below.

(a) ENTER + RETURN

	time secs	core reads	core writes
200000 iterations	53.58	0	84
100000 iterations	28.82	0	84
200000 - 100000	24.76	0	0

(b) ENTER + RETURN + capability argument + capability result

	time secs	core reads	core writes	store writes
200000 iterations	99.52	0	800289	6000575
100000 iterations	51.76	0	400319	3000434
200000 - 100000	47.76	0	399970	3000141

Time for ENTER+RETURN+TCS = 247.6μs

Time for TCS = 2.8μs

Time for ENTER+RETURN = 244.8μs = 0.24ms

Time for ENTER+RETURN+cap. arg.+cap. result+TCS = 477.6μs

Time for ENTER+RETURN+cap. arg.+cap. result = 474.8μs = 0.47ms

Percentage of writes to core = 399970*100/3000141 = 13.3% = 13%

G.3 Slaving effectiveness of Capability Unit

The full results of these experiments are given in Section 7.3.3 except for the calculation of the ratio of the saving in elapsed time to the number of reset cycles saved. The full results of these calculations are given below: the figures are averages over five runs.

(a) Algol 68C

	time secs.	extra time vs slaving	extra resets vs slaving	extra time/ extra resets
No slaving across process or domain switches	324.4	98.0secs.	375751	260.8μs
Slaving across domain switches only	235.9	9.5secs.	42569	223.2μs
Slaving across process switches only	306.9	80.5secs.	341411	235.8μs
Slaving across process and domain switches	226.4	-	-	-

(b) Genesis

	time secs.	extra time vs slaving	extra resets vs slaving	extra time/ extra resets
No slaving across process or domain switches	179.6	38.0secs.	150514	252.5μs
Slaving across domain switches only	155.9	14.3secs.	65727	217.6μs
Slaving across process switches only	170.0	28.4secs.	110048	258.1μs
Slaving across process and domain switches	141.6	-	-	-

Average over Algol 68C and Genesis runs 241.3μs

= 112.8*BS