Technical Report

UCAM-CL-TR-27
ISSN 1476-2986

Number 27

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Exception handling in domain based systems

Martyn Alan Johnson

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 September 1981 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

ISSN 1476-2986

Contents

List of illustrations

Summary

Modern computer operating systems allow the creation of **protection domains**; these enable subsystems to cooperate whilst being protected from each other. This creates a number of problems in the handling of exceptions such as the expiry of time limits or the receipt of console 'quit' signals. Particular problems arise when parts of the operating system are implemented as protection domains which cannot easily be distinguished from user programs by the underlying protection system.

The dissertation surveys some traditional methods of dealing with such problems, and explains why they are inadequate in a domain based system. In addition, work done on related topics in the operating system for the Cambridge CAP computer is described.

The major part of the research described is concerned with a class of exception not usually recognized by operating system designers. This arises from the observation that protection domains which implement subsystems can retain internal state information between invocations, and care needs to be taken to ensure that domains are given an opportunity to keep their private data structures in a consistent state. In particular, domains which fall into disuse need to be notified of the fact so that they can tidy up the data structures they manage before they are destroyed. An intuitively simple solution to the problem is discussed, and its limitations and implementation difficulties are noted. Refinements of the mechanism are proposed which provide an improved treatment of the problem; and it is suggested that the moderate run time overhead which these revisions impose can be minimized by providing hardware or microprogram support for the mechanism.

Preface

I would like to thank my supervisor, Dr R. D. H. Walker, for his advice and encouragement, and for providing me with information about the CTL E4 executive not readily obtainable elsewhere. I am also grateful to Dr R. M. Needham for the interest he has shown in my work and for many helpful discussions. Thanks are due to Professor M. V. Wilkes, Dr A. J. Herbert and Mrs M. S. Atkins for their constructive criticism of this thesis. Particular thanks must go to Caroline Blackmun for her continued encouragement and careful proofreading.

I am grateful to the Science Research Council for their financial support of my research.

This dissertation is not substantially the same as any I have submitted for a degree, diploma or other qualification at any other university. No part of it has been or is being concurrently submitted for any such degree, diploma or qualification.

Except where specific reference is made to the work of others, this dissertation is my own work. It includes nothing which is the outcome of work done in collaboration.

1. Introduction

1.1 Motivation

Recent work on protection in operating systems has enabled systems to be built which adhere closely to the principle of minimum privilege. This requires that each part of a computation should have access only to those resources necessary for it to do its job. The work described in this dissertation assumes the existence of such a protection system, and uses it as the basis for the discussion of a wider class of issues. These include the proper handling of exceptional conditions, and the mechanisms which are needed to ensure that programs can maintain the integrity of the resources they manage, regardless of the behaviour of other programs.

There are a number of motivations for studying these issues. Firstly, proper treatment of such topics can make the task of writing the operating system itself much easier. Well protected operating systems frequently do not display a sharp distinction between system programs and user programs. Instead, a set of basic mechanisms is provided to support both. By creating separate <u>protection domains</u>, programs can be protected from each other, and each can be given access to the resources and facilities it requires. In such systems, the integrity of the operating system may depend on the proper internal functioning of a protection domain regardless of the actions of domains with which it communicates.

Secondly, there is considerable interest in making computer systems highly reliable. To do this requires a combination of many techniques in both hardware and software. One of these techniques is to provide good mechanisms for detecting that a computation is in an erroneous state, and correcting or circumventing the fault which caused it (e.g. the failure of some program to adhere to its specification). No operating system can be expected to put right an incorrect program, but it can attempt to limit the effects that such a program can have on other programs.

Thirdly, many of these issues are handled badly or not at all in existing operating systems. For example, multi-user operating systems frequently provide protection for themselves from their users, and protect the users from each other, but do not allow proper exception handling within a user's computation. In such systems, attempts to construct robust subsystems may be difficult or impossible.

Finally, many computations are nowadays handled by distributed operating systems. These are frequently much more complex than single machine systems, since genuine concurrency is present as opposed to multiprogramming. The possibility of independent failure of the components of a distributed system adds to the complexity, but also allows one to consider recovery from events which would lead to a catastrophic failure in a simpler system. If this possibility is to be realized, it is important to have clean recovery mechanisms which programs can invoke.

1.2 Requirements and limitations

There are a number of conflicting requirements of the mechanisms to be provided. It may be necessary, for example, for one program to be able to cause forcible termination of another if the latter is believed to be engaged in unproductive work, for example, in an infinite loop. This action is based on the implicit assumption that it is the former program which is correct, and the latter which is wrong. This behaviour characterizes the operation of many conventional operating systems; the 'system program' is assumed to be correct, and the 'user program' to be wrong. Whilst this may often be a reasonable assumption, it tends to lead to a system in which it is difficult or impossible to write subsystems which behave in a well defined manner under all circumstances.

This thesis puts forward the view that one should try to avoid building into operating systems the assumption that certain programs have unrestricted control over others. As far as possible, all programs which run within and under an operating system should do so in an environment in which they can handle exceptional events on equal terms. The operating system designer should be prepared to define what this environment is, and to identify the constraints within which a program must remain if it is to react properly to exceptions. Only if the program violates these well defined constraints is it reasonable for the operating system to take action to correct the error in a manner which may override the original intentions of the programmer. In other words, it is the job of the operating system writer to provide an environment in which proper exception handling can be done; actually doing so is the job of the applications programmer, no doubt aided by well designed language systems.

All exception handling mechanisms will have their limitations. It is unrealistic to suppose that all possible faults can be handled in a uniform manner. For example, it will not generally be possible to cope with complete failure of the hardware on which the fault handling mechanisms are implemented. An attempt should be made, however, to do as much as possible in a well defined manner, and to be aware of the limitations of what has been done. Specifications of exception handling are frequently written in positive terms such as '... the fault routine will be entered if event x occurs'. Clearly there may be circumstances in which it is impossible to meet this specification. What is really meant is '... if event x occurs, then event y will not occur before the fault routine has been entered'. ('Event y' may, for example, be the termination of the program). This point of view emphasizes that no attempt is being made to achieve the impossible. Failure of both hardware and software can and will happen. Under such circumstances, specifications become suspect, and it is often difficult to predict precisely what will happen after such a failure. One can, however, make assertions about what should not happen, and attempt to take corrective action if these constraints are broken.

This dissertation does not attempt to address all of these topics. It is concerned with what can be done within an operating system to ensure that sensible fault and exception handling is done. The possibility of failure of the hardware or software implementing the exception handling is admitted; the recovery from such events is beyond the scope of this work. It is intended, though, that by keeping the underlying mechanisms small and simple, the probability of their failure can at least be reduced.

1.3 Overview

Chapter 2 is a general description of the area in which this work falls. It is an informal introduction to the subject, and is deliberately couched in rather vague terms. A more formal description, including assumptions made about underlying protection systems, follows in chapter 3.

Chapter 4 describes work done by others. Partly it is a review of the relevant areas in other operating systems (whose designers were not necessarily attempting to solve the problems addressed in this work). It goes on to describe some work which has been done in programming languages to address the issues concerned. Finally, it discusses more formal proposals made by other researchers.

In chapter 5 some practical implementation done by the author is described. This work was done on the operating system for the Cambridge CAP computer, a system which was developed primarily for research into capability based memory protection. This work left a number of problems unsolved, and provided the stimulus for the more formal treatment in the remainder of the dissertation.

Chapter 6 presents a conceptually simple solution to the problems raised in chapters 2 and 3, and explains how it might be implemented. Chapter 7 exposes the weaknesses of this solution, and proposes two refinements. It is suggested that these refinements are easier to implement than the simple solution is, and have properties closer to those required. Chapters 5, 6 and 7 constitute the main body of the research described.

Finally, in chapter 8, the work done is summarized. Possible applications are discussed, together with their limitations. Suggestions are given about areas in which there is more work to be done.

2. Exception handling

2.1 A user's view

A user of an interactive operating system experiences its exception handling very early on. Beginners frequently make mistakes which cause programs to execute infinite loops, or to produce copious incorrect output. Some of these mistakes will be detected for the user by the operating system, and will generate simple error messages, such as 'CPU TIME LIMIT EXCEEDED'. Other errors, such as the production of incorrect output, are detected by the user himself. When this happens, the user is frequently directed towards a key on his console labelled **BREAK** or **ATTN**, and told that if he presses this, the output will stop and he will be able to correct his program and start again.

From the user's point of view, this is a very simple action, with a very simple result. When the appropriate key is pressed, the current output stops, and a prompt is given for further interactive input. If the user is using a good debugging system, it will probably present him with a brief summary of what the program was doing at the time it was interrupted, and may even present him with an opportunity to interrogate its state interactively. When the program stops, the user should observe that any files that the program may have been using are properly closed, and that any other actions necessary to restore his environment to a standard state have been taken.

The actions necessary in the operating system to achieve these effects are unlikely to be so simple. It is unlikely, for example, that the program interacting with the terminal user will be allowed to read the BREAK key as a character. This might be possible in a single user system, or in an environment in which only a few, trusted, subsystems could be run. But in a general purpose operating system, the fact that the user has pressed BREAK must be interpreted within the operating system, so that it can enforce the intended effect.

Assuming that this is done, action must be taken in the operating system to terminate the user program. Even an abrupt termination may not be easy. There should be little difficulty if the user process is executing the user's own code at the time of interruption, but it is quite likely that it is in fact executing some operating system routine. A typical example would be a supervisor call which performs input and output on the user's behalf. Such routines are frequently written on the assumption that they will not be interrupted - e.g. an output routine may initiate some action on a peripheral device and then set a flag in a system control block to indicate that it has done so. If the operating system data structures are to be internally consistent, then it is required that both of the actions are done or neither of them is. This creates a requirement that the supervisor routine is not stopped at an arbitrary point, or that the part of the operating system which arranges interruption of the user program is aware of such consistency constraints.

In practice it is often inappropriate to terminate the user program abruptly. If a user breaks his program out of an infinite loop, it ought at the very least to be possible to find out where in the program the loop was. It is, however, of little use to the FORTRAN programmer to be told by the operating system that his program was executing at address 156345 when he interrupted it - he needs a statement number or execution backtrace. This creates a requirement for the program to be able to discover that the user has interrupted it, so that it can present information in useful terms.

This is not the only case in which a subsystem requires knowledge of the use of a BREAK function. For example, when editing his program, the user may well issue a command instructing the editor to type out 1000 lines. If he decides that he did not mean this, he will expect the BREAK key to stop the output and let him carry on with his editing. He will not expect the editor itself to be aborted, possibly losing the work he has already done. On the other hand, the programmer working on the development of editor itself <u>does</u> want to be able to abort execution of the editor if he introduces a bug which puts it into an infinite loop.

2.2 Conventional systems

Within conventional operating systems, termination of programs is dealt with in a variety of ways. Some specific examples are given in section 4.1. Many of the solutions are, however, based on the assumption that the operating system is monolithic, in the sense that it is always possible to distinguish 'system program' from 'user program'. If this is so, and the operating system designer is prepared to provide the user program with only limited BREAK handling facilities, then there are few conceptual difficulties in handling such events. The use of the BREAK key can be trapped by the process responsible for dealing with terminal input and output, and action can be taken within the operating system to cause the user process to stop its current activity. The problem of knowing exactly <u>what</u> to stop can be solved by asking the user at the terminal to specify what he wants to have happen. The operating system has knowledge of all its internal data structures and the consistency constraints on them. Programming the actions necessary to maintain those data structures is likely to require considerable care, but should not be inherently difficult. In a monolithic system, it is therefore relatively easy to provide the facilities required in such a way that the terminal user sees a clean interface.

Even in this simple case though, care must be taken to provide reasonable treatment of BREAK within the user process. Operating systems frequently provide good mechanisms for the trapping of faults such as division by zero, but have more difficulty in coping with asynchronous events such as BREAK. It is convenient for the applications programmer if the same basic exception handling mechanism can be used for all faults. There are two main difficulties with this. The first is that it is much more difficult to deal with truly asynchronous events than those which are directly provoked by the program; this is particularly so if an attempt is made to provide a high level language interface. The second problem is that different exceptions provide different potential for recovery. The example given above of division by zero is easy to recover from; at its crudest, recovery can consist of ignoring the error and continuing regardless. At the other extreme, the expiry of some externally imposed CPU time limit may provide little scope for recovery; there would be little point in having a fault such as this if it were possible to ignore it.

2.3 Non-monolithic systems

Modern operating systems are rarely monolithic as assumed above. This creates a number of awkward problems in the handling of events such as BREAK. It not always possible in non-monolithic systems to define exactly what is part of the operating system and what is not - the system may consist of many modules with varying privileges. Particularly in an operating system with good internal protection mechanisms, one module may have little or no knowledge of the workings of another. Interfaces between modules are, or ought to be, narrow and well defined. Such operating system structures present additional problems for the implementor of break handling. No longer can a single program be in charge of the termination of user programs, doing all the work itself, without violating the principles by which the system is built. Since an event such as BREAK occurs asynchronously, the module which happens to be executing when it occurs is unlikely to be the only one which needs to be involved in handling it. Instead, each module must be able to tidy up the data structures which it manages. Organizing such activity can be a difficult task, since there is no single program in charge.

In section 2.1, a routine to perform input and output on a user's behalf was given as an example of part of an operating system with consistency constraints on its private data structures. In a non-monolithic system, a routine to do this would probably be a separate module. The part of the system which handled BREAK would have no knowledge of the consistency constraints on the status information maintained by that module.

2.4 Non-hierarchic systems

If the operating system allows a flow of control more complicated than simple procedural call and return then even more problems arise. A particular case, which will be discussed in more depth later, is that of a system structure based on coroutines rather than subroutines. The essential difference here is that coroutines can remember state internally between activations. If they do this, the processing of an exception becomes much more complicated, since the set of modules which may need to be informed of the occurrence of the exception is not limited to the modules currently on a call stack. Instead there may be many modules active as coroutines, with no easy way of determining any hierarchical relationship between them.

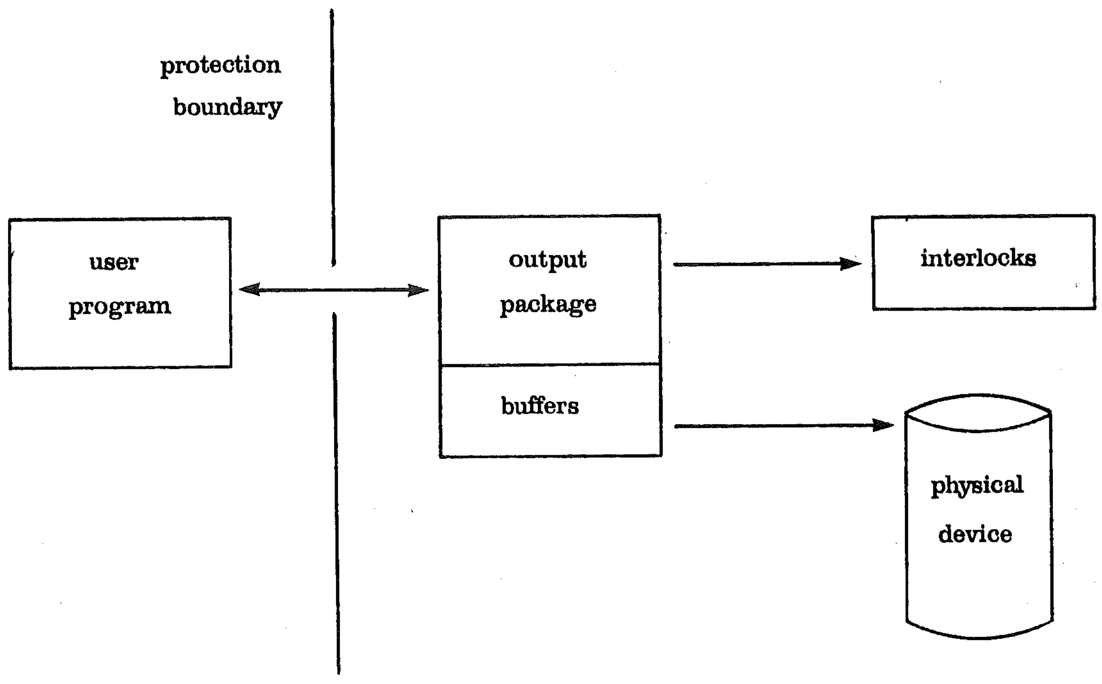

Figure 1: Example of non-hierarchic domains

Figure 1 shows an example of this. A user program communicates with an output package. Data is transferred across the protection boundary between them by coroutine calls; the output package maintains private buffers and sends blocks of data to a physical device when necessary. In such a system, any BREAK signal which causes the user program to stop ought to be communicated to the output package even if control happens to be in the user program at the time, so that the output package can arrange to flush its buffers, release device interlocks etc.

The treatment of this problem, and variants of it, occupies the bulk of chapters 6 and 7. In fact, problems arise even in the absence of events such as BREAK signals, and the discussion of the problem is therefore more general than the remarks above would imply.

3. Protection domains and the problems they introduce

3.1 Basic assumptions

The protection structure in which these problems will be considered is based on capabilities. The model has much in common with some real capability systems, but is not intended necessarily to correspond exactly with any of them. Properties of real systems will be considered later, when implementation problems are dealt with.

The fundamental notions in this protection scheme are the capability and the object. A capability identifies (or names) some object, and possession of the capability allows some access to the object. The degree of access may be encoded in some manner within the capability. In general, objects contain capabilities (for more objects) and 'data'. The latter normally consists of bit patterns which can be manipulated with relative freedom (e.g. arithmetic may be performed on them). The operations which may be performed on capabilities are restricted in such a manner that they cannot be forged, although they may be copied (access controls permitting).

Objects have a property known as their type, and it is assumed that a capability contains (or points to) an indication of the type of the object it represents. Whether there are a fixed number of primitive types or an indefinite number of types is a property of a particular implementation. One type which is assumed always to be available is the 'data segment' - an object which contains data but no capabilities. Further variations may be found in particular systems, such as capabilities which do not name a particular object, but allow some action to be performed (such as allowing access to the internal representation of an object).

In any protected operating system, there is a need to protect programs from one another (the term mutually suspicious subsystems has been coined in this connection [Schroeder 72a, Graham 72]). A central notion of this thesis is that a programmer may write a subsystem to perform some function, and present his users with a single capability which represents that subsystem. This may be done quite easily by providing an object which contains (at least) the code of the subsystem (or a capability for it), and giving the user a capability for this object. The operating system must provide some means by which the code bound within this object can be executed; the object is then said to represent a protection domain. For simplicity, it is assumed that a protection domain consists entirely of capabilities, at least one of which is a capability for a segment containing code.

At any particular moment, the capabilities addressable by a program will be restricted to those available in some current protection domain. Some special instruction or system call will be made available to transfer control between protection domains in a well defined manner [1]. Capability based protection schemes can be made powerful enough to enable a completely disjoint set of capabilities to be made available on a change of protection domain. In practice, such an extreme change is rare; some capabilities may be globally available, and it is frequently necessary to pass capabilities between protection domains as parameters.

----

[1] Not only must the capabilities available be changed, but the protection system must ensure that the new domain begins executing code at a valid entry point.

3.2 Shared data structures

In general, a protection domain will contain capabilities other than those for its executable code. For example, most programs require workspace, and this will normally consist of one or more private data segments. It is important to note that although a protection domain can be isolated from all other domains, it need not be, and in the typical case it is not. For instance, it is highly likely that it will have capabilities allowing transfer to other protection domains (even if only to those providing 'supervisor services').

From the point of view of this dissertation the possible existence of capabilities for private shared data structures is more important. By 'shared' is meant that other protection domains, accessed via different capabilities, may have capabilities for the same object. For example, a mail handling subsystem will have a capability for a data structure containing messages and other information; this structure may be shared with other instances of the subsystem existing in different places or at different times. There may also be sharing with different subsystems which may require different access rights to the object. For example, a program to provide statistics about the mail system could run in a protection domain which shared the data structure with the main subsystem, but had only read access to it.

A number of observations may be made about this sharing. It is important that the protection system should provide the subsystem writer with sufficient facilities to enable him to share access to objects in such a manner that he can maintain their integrity. If the protection system permits more than one instance of the subsystem to run at the same time (in different processes or on different processors) then some means of interlocking accesses to the shared object will almost always be needed.

Another factor to be considered is the relative permanence of the data structure being shared. Some objects may contain information relevant only to the current invocation of the subsystem, and will cease to exist if, for example, the hardware on which the system runs breaks down. Other shared data structures may be more permanent, and may be expected to be preserved across all but the most disastrous failures. If a subsystem has very stringent integrity requirements, then sophisticated techniques will be needed to enable it to recover from total failures. This does not, though, render useless those mechanisms which are provided for the handling of simpler failures, since the full recovery may be very time consuming. For example, failure to handle some minor exceptional condition in a database system might cause a semaphore not to be released, which could in turn lead to a deadlock. Although it would be possible to recover from this deadlock by restarting the system using the mechanisms provided for handling serious failures, this would probably be an expensive operation, which could have been avoided if the original exception had been handled properly. This is an application of the general principle that any error should be handled as soon as possible in order that its effects should not propagate too far.

3.3 Maintaining the integrity of protection domains

It should be evident that with a system as general as that described, very complex control structures are possible. When protection domains interact in a complex manner, care must be taken to ensure that each can maintain the integrity of its own private data structures.

The principle being aimed for can be stated in very simple terms. It is that if a protection domain initiates some action which involves intermediate, non-standard states, then nothing should prevent that domain from returning to a standard state. The definition of 'standard state' is not necessarily a global property of the protection system, but may be a property of each protection domain [2]. An example of a typical 'standard state' for a domain might be: 'interlock on shared data segment is not claimed'. This is a very simple example; a more complex one might make assertions about the contents of the shared segment. It follows from this that restoration of a domain's standard state generally requires execution of code within that domain.

Although the above remarks have been put forward as an ideal, even this would only be satisfactory in an environment in which protection domains can be trusted not to make unreasonable demands of the mechanisms provided to enable them to maintain their integrity. Within the components of an operating system, such cooperation of domains may reasonably be assumed, but the same would not necessarily be true of a user supplied subsystem. Any routines provided by the subsystem writer to tidy up his private data structures are just as likely to be wrongly programmed as any other part of a subsystem, and this must be taken into account.

3.4 Restrictions imposed by the protection system

Another point which needs to be made is that the difficulty of solving the problems depends very much on what the programmer is allowed to do while in a non-standard state without compromising his ability to recover. To take an extreme example, if the domain is not allowed to be in a non-standard state except during the execution of a machine instruction, and the hardware being used guarantees indivisibility of instructions, then the operating system need provide no facilities to enable the programmer to restore a standard state at all. Any de facto inconsistency in the domain's data structures between instructions would be, by definition, a programming

---

[2] A term sometimes used for this is that the domain maintains an invariant: a condition which will be made true at certain well defined times. The term is not used here because of the implication that an invariant must be true whenever control is outside the domain. As will be seen later, this is not necessarily so for the 'standard states' described here.

mistake. The ease with which the problems have been thus removed from the operating system is clearly at the expense of presenting the programmer with a very restrictive set of programming rules. This may make it difficult or impossible for the programmer to provide a subsystem with the properties he desires.

In order to provide the programmer with a more conveniently used environment, it is desirable to relax the constraints on him as much as possible. For example, it would seem reasonable that a programmer should be able to accept and recover from faults such as memory protection violations while in a non-standard state. As will be seen in chapter 4, many existing operating systems allow this to be done.

One of the main issues addressed in this dissertation is that of allowing a protection domain to relinquish control whilst in a non-standard state. If this is to be allowed, then the operating system must be prepared to guarantee to return control to the domain at some later time, regardless of whether or not it is explicitly called.

There are two main ways in which a domain can relinquish control. It may, for example, call another domain as a subroutine, expecting control to return to it when the subroutine returns. The second way in which a domain can relinquish control is simply by returning to its caller.

If a domain relinquishes control while in a non-standard state, then presumably it does so with the expectation of being called again at some later time. When a subroutine call is made, this is implicit: the domain will receive control again when the called domain returns.

If the domain simply returns, however, there is in general no implied indication of the non-standard state existing. It is not immediately obvious that returning in a non-standard state is a reasonable thing for a domain to do. The main application for it comes when a domain provides a user with some service which requires several transfers of control between it and the user. A protection domain providing the user with an interface to an interactive terminal is an example. The non-standard state would be that an interlock on the terminal is claimed - this state would (and ought to) persist when control is (temporarily) returned to the user's domain. Nevertheless, the eventual release of the interlock should not depend on the correctness of the program in the user's protection domain.

3.5 The problem to be solved

It is the aim of this dissertation to describe a mechanism which allows domains both to call others and to return to their callers, without losing the ability to recover their standard states later. As will be seen in chapter 4, techniques to deal with the former case are already known, and the research described in chapters 6 and 7 has therefore concentrated on the latter.

4. Related work

This chapter is a review of previous work which has been done in this field. It falls into three parts: section 4.1 is a discussion of what has been done in operating systems in the past, section 4.2 discusses linguistic approaches to the problem, and section 4.3 reviews more formally proposed solutions.

4.1 Other operating systems

Many operating system designers have attempted to address at least some of the issues set out in chapter 3. No attempt will be made to give a complete survey - instead a number of examples will be chosen to illustrate important features of previous work. The particular case of the operating system for the Cambridge CAP computer is discussed in Chapter 5.

4.1.1 MULTICS

The first system to be considered is MULTICS (Multiplexed Information and Computing Service). This is an example of a protected operating system based on rings of protection. At any particular moment a process is running in one particular ring; it has complete control of material in its own and higher numbered rings, but has restricted or no access to anything in lower numbered rings, other than by way of gates providing entry points into a more privileged ring [Schroeder 72b].

4.1.1.1 Conditions

MULTICS provides mechanisms to enable a program to nominate handlers for a variety of exceptions. These mechanisms are described in terms of the facilities provided in a particular programming language, PL/I [Honeywell 72]. Exceptions in MULTICS are known as conditions and are given condition names. An example of a condition is division by zero, and its condition name is **zerodivide**.

It is possible to nominate an exception handler for a given condition by means of a PL/I **ON** statement. When a condition occurs, it is said to be signalled, and the corresponding handler - the body of the **ON** unit - is called. The ON unit may either take corrective action and return, or it may divert the program by means of a **GOTO** statement. (An example may be found in section 4.2.2.)

The PL/I mechanism uses a number of underlying primitives: [MPM 73, Organick 72] **condition_**, **reversion_**, **signal_** and **find_condition_info_**. These system subroutines may also be called directly, independently of the PL/I **ON** statement, and are thus available to other languages. The **condition_** primitive sets up a handler for a named condition (or replaces an existing handler in the same block activation). The **reversion_** call will remove the most recent handler for a condition. The **signal_** subroutine is used to cause the most recent handler for a condition to be executed; it takes various parameters describing the environment in which the exception occurred. The parameters of the **signal_** call, together with other information about the exception, are made available to the handler by calling the **find_condition_info_** subroutine.

A handler can perform any action necessary to deal with the occurrence of a condition. If it can correct the circumstances which caused the exception, it may cause the program to be resumed at the point of interruption. It does this simply by returning to its caller, **signal_**. It is a programming convention that if a call of **signal_** returns, the operation which provoked the exception should be retried [MPM 73]. If the problem cannot be rectified by the handler, it may perform a non-local jump to some location in the interrupted program or to one of its callers. Only the most recent handler for a given condition will be activated.

A special condition name is provided for handling all conditions. This is the **any_other** condition. The **any_other** condition handler will be executed if there is no specific handler for the condition raised. For each procedure activation, starting from the most recent, a search is made for a handler for the condition raised, or, failing that, a handler for the **any_other** condition. The first suitable handler found is the one activated. If no suitable user supplied handler is found, a default handler is activated: this usually prints a message and terminates the user's program.

During the search for a handler, protection boundaries may be crossed. If no handler can be found in the procedure activations within a given ring, then that ring is abandoned, and the condition is signalled in the calling ring. When nothing remains, the process is terminated. A more detailed treatment of the interaction of protection rings and the signalling of conditions may be found in [Organick 72].

4.1.1.2 Termination of programs

Most conditions detect programming errors and are logically very similar, but some are of special interest:

- The **finish** condition is signalled when the user's process is to be terminated by voluntary or involuntary logout. A small time interval is allowed for the handler to run before the process is actually stopped.
- The **program_interrupt** condition is signalled by a specific user command.
- The **quit** condition is signalled when the user presses a particular key on his terminal.

The last two of these are used to implement attention handling. Both of them can be handled by the user program, but the MULTICS Programmers' Manual [MPM 73] recommends that the user should not attempt to handle the **quit** condition (and that an **any_other** handler should merely pass it on). The intention is that the default handler should be used; this causes a new activation of the command program to be established, without disturbing the user program's stack. From this new command level, the user at the terminal can use the **program_interrupt** command to return to the interrupted program, signalling the **program_interrupt** condition, which the programmer can handle as he chooses. This two stage process has the merit of giving the terminal user control over what is done after a program is interrupted [1], but relies on the cooperation of all programmers [2].

4.1.1.3 Cleanup procedures

A further feature of the MULTICS system is of interest. Both ordinary programs and exception handlers may perform non-local transfers (sometimes known as 'longjumps'). This may cause procedure activations to be destroyed. If a procedure has work to do which it must complete before being terminated, it can establish a cleanup procedure. A subroutine **establish_cleanup_proc_** is provided to do this, and **revert_cleanup_proc_** will remove the record of the cleanup procedure if it is no longer required. Reversion of cleanup procedures on ordinary procedure return is automatic. A cleanup procedure will be invoked if the block activation which established it is being aborted because of a non-local transfer. Cleanup

---

[1] In particular, an interactive debugger may be invoked.

[2] The MULTICS Programmers' Manual points out that a program with a bug in its **quit** handler might be impossible to interrupt. This is in contradiction to the "Beginners' guide to the use of MULTICS", which states that the 'quit' button on a terminal will always enable the terminal user to take control of a 'runaway or incorrect program' [MPM 73 (Chapter 3)]. In fact, it would seem from the more detailed documentation that some programs can be terminated only by forcing a logout (which can be done by disconnecting the terminal); this will signal the **finish** condition.

procedures differ from exception handlers in that a single event can cause more than one to be invoked.

4.1.2 IBM OS/360

In contrast to an operating system designed and developed in a research environment, it is interesting to consider what is provided in a typical manufacturer's operating system. The system to be considered is IBM's OS/360 system for the System 360 and 370 series machines. (The specific system dealt with is OS/MVT release 21, but most of the remarks made apply equally to other OS releases, as well as more recent VS systems). Of particular interest is the fact that some of its facilities have been found to be quite inadequate in a computing service context, and examples will be given of modifications which have been made to this system to support better exception handling.

The main characteristic of exception handling in OS/360 is a lack of uniformity. A variety of ad hoc mechanisms are provided for various exceptions. The exception handling functions are provided by means of the supervisor call instruction (SVC) [IBM 1], and are described in terms of Assembler language macro instructions which generate calling sequences for SVCs. The descriptions are all at a low level (machine registers, system control block pointers etc.) and are intended for the Assembler language programmer.

4.1.2.1 Program interruptions

The first category of exceptions which can be handled by the programmer are those detected by the hardware. These are called program interruptions, and include such errors as use of an invalid instruction, storage protection violation and division by zero. When these errors occur, the hardware passes control to an interrupt routine, which examines various system control blocks to determine what should be done in the user task. If a user program wishes to trap such events, it sets up an exit routine using the **SPIE** macro (Specify Program Interruption Exit) [IBM 2]. The operands to the macro specify which interruptions the programmer wishes to trap. If an interruption occurs, the control program will call the exit routine with various parameters indicating the nature of the interruption and where it happened (except that instruction pipelining and memory caching can have the effect that certain program interruptions cannot be localized to a particular instruction). The exit routine can arrange to alter (arbitrarily) the registers and program counter of the main program and therefore resume execution at any point.

A task may have only one program interruption exit routine set up at any one time. When a programmer issues a **SPIE** macro instruction, the control program returns the address of a control block describing any previously set up exit routine. It is the programmer's responsibility to save this address so that the previous exit routine can be restored when the current one is no longer required. Since there is no check that this is done correctly, it is not safe for a module to trust that its program interruption exit is still set up after calling another module.

4.1.2.2 ABENDs

If there is no program interruption exit routine set up, the control program takes a default action which causes termination of the task. This termination is called an ABEND (abnormal end), and can occur for many other (software detected) reasons. Most ABENDs can themselves be trapped by the programmer, using the **STAE** macro (Specify Task Abnormal Exit) or the **STAI** (Subtask ABEND Intercept) [IBM 2] operand of the **ATTACH** macro, which is used for creating new tasks [3]. An exit routine is provided in the same way as for program interruptions, but the environment within it and what it can do are different. Information is provided giving the nature of the ABEND and where it occurred. The exit routine can examine this information

----

[3] **STAE** is used by a task to trap its <u>own</u> ABENDs; **STAI** is used by a task to trap the ABENDs of the created subtask.

and decide whether termination processing should continue or whether a '**STAE** retry routine' [IBM 3] should be scheduled to cause resumption of the user program. In common with program interruption exits, there may be only one **STAE** exit active at once, but the control program does provide for automatic restoration of a previous **STAE** exit when a program returns to its caller.

The **STAE** mechanism would appear at first sight to provide a general means of handling exceptions, albeit somewhat awkwardly. However there are a number of conditions under which abnormal termination of a task can occur without the **STAE** exit routine being entered. A few examples of these conditions are:

- Operator cancellation of a job, or time limit expiry
- ABEND recursion
- normal termination of a task which has active subtasks
- ABEND of a parent task

It will be clear from this that sub-system writers have severe difficulty in handling even fairly commonplace exceptions. For example, high level language implementations may wish to intercept expiry of the job's CPU time limit in order that they can give a sensible diagnostic when a program is in an infinite loop. The ABEND which occurs when the job time expires cannot be trapped. It is therefore necessary for subsystems to do their own timing, and attempt to anticipate the expiry of the system timer. This is possible, although it is fraught with difficulties which will not be described here. Other, rather similar, exceptions (such as producing too much printer output) cannot be handled at all.

A further, somewhat notorious, problem is that ABENDs can occur while input and output operations are in progress. The operating system attempts to deal with this, but frequently does so in such a manner that it is dangerous to attempt to do further input and output after recovery from the ABEND; to do so risks ABENDing again. This reduces the power of the mechanism considerably, since the programmer cannot be assured of a clean environment after error recovery.

4.1.2.3 Console attentions

The handling of 'console attention' under TSO (time sharing option) is another area in which OS/360 lacks proper exception handling facilities. Superficially the handling of attentions is similar to the other exceptions; the programmer writes a **STAX** (Specify Time Sharing Attention Exit) macro [IBM 4] to set up an exit routine to be called when an attention occurs. It can also specify a line of data to be written to the terminal, and request that a line is read from the terminal before the attention exit routine is entered. The exit routine cannot divert the main program, except by forcing an ABEND, and in most programs its actions are limited to setting a flag which is polled elsewhere. This is, of course, a dangerous programming technique; if a programming error causes the program to execute an infinite loop in an unexpected place, the flag may not be inspected.

The main problem with OS attention routines occurs when more than one is set up at one time. A TSO session, like any other job, may be running several tasks - a typical example being a command program which runs user programs as subtasks. If there is more than one attention exit set up, the one entered is the one which was set up most recently. Since tasks may set up attention exit routines asynchronously, it is not necessarily predictable which exit routine this will be. Moreover, this rule has the effect that an exit routine which a command program sets up to provide a means by which the terminal user can abort a command will be overridden by an exit routine which the command sets up for itself. This is often what is wanted, but it can make it difficult to abort the command when that is what is wanted.

The above description is slightly over-simplified. There is the additional rule that an attention which occurs before processing of the previous one is complete will cause the current attention exit to be abandoned entirely and the next most recent to be called instead. It can therefore be possible for two (or more) attentions in quick succession to have different effects - depending on their exact timing. A commonly observed problem is that programs which set up attention exit routines can be very difficult to break out of, except when the system is overloaded and running slowly, in which case two attentions in quick succession will work. It is clearly undesirable that the ability to escape from a program should depend on the system loading.

4.1.3 IBM OS modifications

Exception handling in OS/360 can be seen to leave much to be desired. To some extent it is possible to improve on what has been provided in the standard operating system. This section describes some improvements which have been made to OS by the staff of the University of Cambridge Computing Service [Larmouth 76, Harrison 79].

4.1.3.1 ABEND handling

The first category of change has been the provision of an 'early warning' of most of the common exceptions which cannot be trapped by the **STAE** mechanism. For example, half a second before the CPU time limit for a job or command is due to expire, an ABEND with a locally defined code number is issued. This ABEND can be handled using the normal **STAE** mechanism, and a retry routine can be scheduled to perform tidying up operations in the remaining half second. If the time limit expires again, then the untrappable ABEND will be given as before. The practical effect of this is that language systems can give useful diagnostics (such as subroutine tracebacks) when programs exceed their time limit without the contortions required in standard OS. Similar trappable ABENDs have been added for most other errors caused by excessive use of resources, although there remains no corresponding facility for such cases as cancellation of a job when the system is being closed down.

4.1.3.2 Console attentions

The second area in which changes have been made is in the attention handling. A minor improvement is that multiple exit routines entered as a result of multiple attentions are properly nested - that is, the first attention routine is resumed when the second is completed instead of being abandoned. The main change, however, is that it is possible for the user at the terminal to direct the attention to a particular exit routine (and hence, by implication, to a particular program). As part of the handling of the pressing of the terminal break key, the user is prompted for a keyword. An attention exit routine can have a list of keywords associated with it, and the available exit routines are searched until one is found which nominates the key typed by the user. This exit routine is then entered in the normal manner. For example, the key 'Q' is taken by the command program to request termination of any program which is running under it, and 'LOGOFF' is trapped by the terminal monitor program to request termination of the command program. Various utilities can set up their own keys for their own purposes without affecting these. Since the IBM provided strategy of entering the most recent exit routine is often what is in fact wanted, this can be requested by typing a null key.

A curious property of the revised mechanism is that there is no check that duplicate keys are not set up. A user program can, therefore, set up the key 'LOGOFF', and trap attentions that the user intended to terminate the session, since the normal search order for keys is that the most recently set up keys are checked first. If, after doing any local tidying, it were possible to propagate the attention on to the next exit routine with the same key, this would be a useful facility. As it stands, this is not possible, and the effect merely causes confusion. The default search rule enables a program to be written which traps all of the system keys. In order that it should still be possible to escape from such a program, a special syntax is available [4] which requests that the search for exit routines is made in the opposite direction, i.e. starting with the earliest set up.

Although the modified attention handling does much to improve the online system as seen by the user at the terminal, it does little to help the programmer who needs to tidy up after an attention. An exit routine in a command program can cause the user's task to be terminated immediately, and there is nothing that the user can do about it - this is a general property of OS rather than a particular failing of the attention mechanism.

4.1.4 The Titan supervisor

An operating system which was considerably more helpful to the programmer in its fault and break handling was the supervisor for the Titan computer, a prototype of the ICT Atlas 2. A general description of the Titan supervisor may be found in [Wilson 71]. The facilities available to the programmer are described, in low level terms, in the machine code programming manual [UML 69].

4.1.4.1 Faults

Communication between the programmer and the Titan supervisor was by means of extracodes - orders in the format of machine instructions but in fact interpreted by the supervisor rather than by the hardware. A large number of possible exceptions (called, on Titan, faults) could be detected by the hardware and supervisor, and were divided into three categories: direct faults, delayed faults and special faults.

Direct faults were those provoked by some specific program action which could be detected and reported immediately, such as 'negative argument for square root'. A delayed fault was one which could occur at an unpredictable

----

[4] typing '!' before the key

time, such as the failure of a magnetic tape transfer which was proceeding asynchronously. A special fault was a more serious condition, which required termination of the program, such as 'execution time limit expired'.

All direct and delayed faults could be trapped, using the 1150 extracode. This specified the fault number to be trapped, and a trap address. When the particular fault occurred, the trap routine was entered, and some information was provided (in machine registers and via other extracodes) to enable the programmer to determine the cause of the fault and recover. A further aid to the programmer was that delayed faults, being asynchronous anyway, were queued by the supervisor so that the trap routine did not have to handle more than one at once.

Special faults could not be trapped, but normally caused entry to the system monitor routine, which printed out standard diagnostic information. However a facility was provided to enable a user routine to be called instead of (or as well as) the system monitor. This routine was called a private monitor, and was set up by calling the 1112 extracode. The private monitor could handle all faults reported by the supervisor. As its name implies, it was intended to enable the programmer to print out diagnostic information in a format of his own choice, but could also be used to enable a program to tidy up before termination. An example given by Needham [Needham 71] is that of the QUEUEJOB command, which manipulated a sensitive file containing other users' passwords. Its private monitor routine had the job of closing the file properly and ensuring that embarrassing information was not left available to the user after it had terminated.

There were, of course, restrictions on what the private monitor could do. Firstly, entry to the private monitor was not allowed to happen more than once in any one program invocation; any further fault would cause the system monitor to be entered and the program to be terminated. The supervisor did, however, attempt to reduce the probability of a second fault by doing such things as waiting for peripheral transfers to complete before entering the private monitor. In addition, the CPU and execution time limits for the private monitor routine were set to fixed values (five seconds and three minutes respectively), thus allowing the program a limited time to tidy up.

4.1.4.2 Console attentions

Console 'quit' signals were implemented simply by mapping them onto faults. The user could choose three different signals: @I ('console signal') caused a delayed fault, and @Q and @M ('console quit') caused special faults. The only distinction between the latter two was that @M caused the printing of diagnostic information by the system monitor, whereas @Q did not.

All of these faults could be handled by the private monitor; in addition @I could be trapped and used as a signal to the running program. Within the limitations of a rigid two level structure, the Titan supervisor was thus close to ideal in allowing user programs reasonable opportunity to tidy up. The main infelicity in the system was that a 'console quit' when in the private monitor counted as a second fault, and it was thus possible for two 'quit' signals in quick succession to cause embarrassment, by terminating the private monitor routine.

4.1.5 CTL E4 executive

Another manufacturer's system which has features of interest is the E4 executive of the MODUS operating system for the CTL Modular One and the Series 8000. There is no publicly available reference work describing this system; the information in this section is derived from experience with the use of this system to implement a student teaching system [Walker 81].

In E4, the unit of protection is called a <u>sphere</u>, which comprises one or more <u>activities</u> [5], together with other resources such as segments of

----

[5] 'Activity' is the term used in E4 for 'process'.

memory and semaphores. Activities may create and delete resources within the sphere.

In some respects, E4 behaves as a conventional monolithic operating system as described in section 2.2. A sphere is deleted on termination of its last activity, at which point the sphere may still own resources of other kinds. The executive undertakes to delete these resources itself, and to perform any necessary tidying up actions, such as the cancellation of outstanding peripheral transfers.

There is a hierarchic ownership relationship between spheres [6], and the owner of a sphere may terminate it. This is done to implement console attention, CPU time limits etc. Typically, a user program executes in a sphere created and owned by a command interpreter sphere; the command interpreter can terminate, suspend or resume the user program sphere at any time. When this happens, the activities in the user sphere are given no opportunity to tidy up.

E4 does not cause interrupts or traps within activities; synchronization is done by polling for messages. Since this prevents activities from handling their own faults, E4 allows one activity in a sphere to be an error-collector. This activity is informed (by executive message) of faults in other activities in the sphere. The error-collector is normally assumed to do nothing other than handle faults in other activities, and a fault within it will terminate the sphere. The error-collector may read and write the registers of the failing activity by means of the **READCONTEXT** and **WRITECONTEXT** executive functions. While this is going on, the failing activity is held suspended, and can be resumed or terminated by the error-collector.

A great deal of care is taken to ensure that the registers of other activities (as seen via **READCONTEXT** and **WRITECONTEXT**) always reflect the

---

[6] This does not imply a hierarchic protection relationship.

state of the user program registers, even if the activity concerned is executing in an executive routine. In addition, any actions on other activities (such as suspension or termination) are pended, if necessary, until the activity is back in user code. This is done in order to ensure that executive routines can run to completion without interference, and these routines are programmed on that assumption.

There is one particular aspect of E4 in which it does not behave as a monolithic system. Suitably trusted user level programs may add facilities to the executive by providing User operations (UOPs). Functions provided by UOPs may be called like any other executive function. UOPs can be used, for example, to enhance the input and output facilities of the executive by providing a disc filing system and a spooling system.

A UOP runs in a protected environment like any other user program, so a call to a UOP is in some respects a change of protection domain. Since the executive has no knowledge of what UOPs are doing, it cannot arrange to tidy up their data structures when a sphere is terminated. Instead, a UOP can nominate a second entry point, at which it will be called just before termination of each sphere. Each UOP can thus tidy up its own data structures (for example, the filing system can write directories to disc). Only when this has been done are the resources of the sphere actually deleted.

This means that UOPs can relinquish control safely after doing work on behalf of a sphere, knowing that the sphere will not be deleted before they have had a chance to restore a standard internal state. The mechanism is analogous to those which a general system must have to satisfy the requirements expressed in section 3.5. It is not itself a complete solution for two main reasons:

- All UOPs are called on all sphere terminations regardless of whether they have done any work for the sphere concerned. This is acceptable only if there is a small number of them.

- Only specially trusted programs can make use of the facility.

4.1.6 UNIX

UNIX [7] is a popular operating system developed at Bell Telephone Laboratories, New Jersey [Ritchie 74]. It is a multi-user system, and allows each user to create multiple processes in a single session.

Exception handling facilities are provided as part of the operating system [UNIX 79]. In UNIX, one process can send a signal to another. There is a fixed number of possible signals [8]. The default action of all signals is to stop the process. Examples of such signals are **SIGINT** and **SIGQUIT,** which are sent to the process running a command when particular keys are typed on the terminal. The **SIGINT** signal is the normal means of breaking out of a command.

A program can specify different treatment of signals by using the **signal** procedure. For each signal it can specify either that the signal should be totally ignored, or that a handler routine provided in the program should be called. If a program ignores the **SIGINT** signal, then the ordinary console attention mechanism will not work for that program; if a handler routine is specified, it can have arbitrarily chosen effects. There are no limits on the number of times a signal may be ignored or handled, and no execution time limits on the process after a signal has been received.

---

[7] UNIX is a trademark of Bell Telephone Laboratories.

[8] The number of possible signals is about 15, but varies in different versions of UNIX.

In order that it should be possible to kill a process which ignores signals, a signal **SIGKILL** is provided which can be neither ignored nor handled. This signal can be sent using the **KILL** command. This signal will terminate a process immediately, giving it no opportunity to tidy up at all.

The system primitive underlying the **KILL** command, **kill**, requires two arguments: the number of the signal and the number of the target process. Process numbers are simple integers allocated in sequence as the system runs. To prevent a user from arbitrarily killing processes belonging to a different user, **kill** requires the user identifiers associated with the source and the target processes to be the same.

An unfortunate flaw in this scheme is that it has an unfavourable interaction with the 'set-userid' feature. The UNIX filing system records the user identifier of the creator of each file, and the creator of a program can specify that when an object module is executed, the user identifier of the process should be set to that of the creator of the file rather than that of the person invoking it. The intended use of this feature is to enable programs to access files which are not normally available to the person running the program, and is therefore in some sense a change of protection domain. This temporary change of user identifier also means that the person who started execution of such a program may be totally unable to stop it, since attempts to use the **KILL** command or the underlying system primitive will be rejected. On the other hand, the person who created the program can kill it arbitrarily if he can discover (or guess) the process number [9].

The exception handling in UNIX can thus be seen to fulfil neither of the main requirements of a satisfactory system: it is not possible to trap all exceptions, and it is not possible for an ordinary user to force eventual termination of all programs.

----

[9] Such programs could also be stopped by the 'super-user'. This privilege would normally be available only to a restricted group of system programmers.

4.2 Linguistic models

Some researchers have attempted to solve the problems of exceptional condition handling by defining linguistic models, some of which have been wholly or partially implemented in programming languages. Much attention has recently been given to the design of programming languages suited to the writing of operating systems and of large, complex applications programs. Since all programs are written in some language, it is important to consider how exception handling facilities are to be mapped onto language primitives; on the other hand it is equally important to be aware of the limitations of a purely linguistic approach. This section surveys some actual and proposed linguistic models of exception handling.

4.2.1 History

Early programming languages had no facilities explicitly provided for the handling of exceptions. Yet exceptions in some wide sense have always been something programmers have had to deal with, even if they did not use that terminology.

Consider, for example, a free store package, providing functions for claiming blocks of store and releasing them again. The subroutine for getting a block of store will return a store address as a result. But it must do something if the required store cannot be obtained, and a typical solution has been to return some special, otherwise invalid, value such as zero. The caller is expected to test the result and react appropriately.

A similar technique is to modify the flow of control in some way. A practice particularly common in assembly language programming is to provide more than one possible return point from a subroutine. In higher level languages, subroutines may simply execute non-local jumps when some exceptional event occurs. In the example given above, the 'get store' routine may just jump out to a particular label if the call fails.

Programming techniques such as these are so widespread that it is rare for them to be regarded as exceptional condition handling mechanisms at all.

More powerful techniques emerged later. Some languages allow procedure variables which can be updated to refer to routines of the user's choice. For example, in BCPL [Richards 79] the global procedure **ABORT** [10] is called on most serious errors. A program may assign its own routine to the global variable **ABORT**, and this routine will then be called instead of the one provided by default. It can be noted here that when changing the procedure variable, it is possible to remember the previous value. After the local exception handling is complete, it is therefore possible to call the previous exception handler, but there is no way that doing this can be enforced. A further example of the use of procedure variables for exception handling may be found in the description of attention handling in the CAP operating system (see section 5.3.4).

4.2.2 PL/I

Some programming languages have explicit support for exception handling. An early example is PL/I [IBM 5], which has already been mentioned in connection with the MULTICS system (see section 4.1.1). The **ON** statement allows a handler (the body of the **ON** unit) to be associated with a named exception. Exceptions such as division by zero are raised by the runtime system when they occur; it is also possible to raise exceptions explicitly using the **SIGNAL** statement. The latter facility allows programmers to specify their own exceptions, which are not explicitly known to the language system.

----

[10] Strictly speaking, this is a property of the implementation dependent runtime system, since the language specification says little about the runtime system. However, **ABORT** is sufficiently standard to be regarded for practical purposes as part of the language.

The following skeleton program shows examples of these constructions:

```
/* a, p, r and x are assumed to
   be declared at this point.
*/

declare usererror condition; /* declares a programmer defined
                                condition name
                             */

...

on zerodivide begin; /* ON unit to handle division by zero */
              if a = 0 then signal usererror;
              else goto fail;
              end;

...

x := p/r; /* system will signal zerodivide if r is zero */

fail: /* ON unit can jump here */

...
```

The exception handling mechanism of PL/I is, however, rather restrictive, since the only communication between the signaller and the handler is by means of global variables. **ON** units can exit by using the **GOTO** statement to perform a non-local jump. If the **ON** unit returns, control returns to just after the **SIGNAL** statement; for exceptions raised by the runtime system, some default action (specified separately for each condition) is then taken. A critical evaluation of the exception handling facilities in PL/I may be found in [MacLaren 77].

4.2.3 MESA

A number of later languages have attempted to improve on this. A good example of exception handling may be found in the language MESA [Geschke 77, Mitchell 79]. MESA uses the term signal to refer to an exception, and the term catch phrase to denote an exception handler.

Signals are declared in the same way as procedures, by means of a type constructor. This specifies the parameters to be passed from the signaller to the catch phrase, and any results to be returned, e.g. [11]

```
BadParameter: SIGNAL [reason:CARDINAL]
                     RETURNS [fatal:BOOLEAN];
```

Generating a signal looks like a procedure call except that the word **SIGNAL** or **ERROR** [12] is present. There is also a construct known as the special error, consisting of the single word **ERROR**. This allows generation of an unspecified error signal; it merely asserts that something has gone wrong. Examples of these are:

```
DontContinue <- SIGNAL BadParameter[n];


ERROR;
```

A program which needs to handle an exception must provide a catch phrase. Catch phrases may be attached to a number of constructs, the main examples being procedure calls and **BEGIN ... END** blocks. The catch phrase must name the particular signals it is to intercept, or it may have the label **ANY**, which, as its name implies, will catch any signal, including a special error. An example of a catch phrase on a procedure call is:

```
Procedure[argument ! BadParameter =>
  BEGIN
    -- the body of the catch phrase
    IF reason < 0 THEN RESUME[FALSE] -- returns a result
                                     -- to be interpreted
                                     -- in the procedure
    ELSE ERROR; -- a recursive signal
  END ];
```

---

[11] In the examples, words entirely in upper case are MESA reserved words; words in lower or mixed case are identifiers. '--' introduces a comment.

[12] The difference is explained later.

When a signal is generated, the MESA runtime system searches for enabled catch phrases, beginning at the most recently activated procedure, going down the call hierarchy. Any catch phrase which matches the signal name will be called, with the arguments which were passed to the signal being available to it. It can also refer to any variables it can access by virtue of its textual position in accordance with the normal scope rules of the language. It is important to note that even though the first catch phrase which is called may be several levels down the call hierarchy, the environment of the signaller and all the intermediate levels still exists. This enables the catch phrase to cause the signaller to be resumed if it can correct the condition which provoked the signal or if it returns information to the signaller. In the example, the procedure '**Procedure**' is assumed to signal the '**BadParameter**' signal if the argument to the procedure is wrong. The catch phrase may resume the signaller, passing back information about whether it is sensible to continue or not. If the procedure generating the signal cannot sensibly be resumed, then it can indicate this by using **ERROR** instead of **SIGNAL**. The MESA system will then fault any attempt to resume the signaller; as might be guessed, it does this by means of a recursive signal.

Another possibility is that the catch phrase will reject the signal, which will cause a search for another suitable catch phrase. Rejection is indicated by falling out of the code associated with the catch phrase.

A catch phrase can also cause the statement which provoked the signal to be retried by using the **RETRY** statement, and can cause continuation at the statement after the one which provoked the signal by using **CONTINUE**.

A final possibility is that the catch phrase will exit by means of a non-local jump. If this is done, the jump does not occur immediately; instead the runtime system passes the signal **UNWIND** to any intermediate procedure activations which are about to be destroyed. Procedures which need to tidy up before being destroyed must include a catch phrase for **UNWIND**, which is a pre-declared **ERROR**.

MESA allows catch phrases to generate signals, so that the whole mechanism may be invoked recursively. However, special care is taken in the case of a catch phrase signalling the same signal as an earlier one. When handling the second signal, any procedure activations which have already had a chance to handle the first signal are missed out of the search for a catch phrase, thus avoiding recursive loops.

MESA provides explicit support for parallel processes and monitors. A limitation of the signalling mechanism is that a signal cannot be propagated from a process to the process which created it; any signal not caught within a process will be passed to the debugger. Care must also be taken within monitors which can provoke signals. The monitor lock is not released when a signal is generated from within a monitor (since the invariant being maintained by the monitor might not be satisfied at the time of the signal). Since a catch phrase can jump out of an entry procedure of a monitor, the programmer must provide an **UNWIND** catch phrase. This catch phrase can restore the monitor invariant, and even if this is otherwise a null operation, it will have the side effect of releasing the monitor lock.

4.2.4 A BCPL exception mechanism

In order to demonstrate that special language support is not essential to provide reasonable exception handling, a mechanism invented for the language BCPL [Richards 79] can be considered. It is implemented not by changing the language definition, but by providing a set of library routines. The mechanism is the work of the Rainbow Group in the University of Cambridge Computer Laboratory, and is described fully in local documentation [Wilkes 80, Singer 80].

Since the mechanism is provided as a library package rather than as part of the language, it is necessary to initialize the system explicitly, by a call of the form: [13]

```
EX.init()
```

Once this has been done, handlers may be associated with an exception by a call of the form:

```
EX.on(@exception, handler.function, arg1, arg2)
```

and the call

```
EX.off(@exception)
```

removes the most recent handler for an exception.

An exception is represented by a BCPL word, and is identified by its address (the BCPL @ operator gives the address of a word).

To signal an exception, the call is:

```
EX.signal(@exception, type, arg3, arg4)
```

The handler is called with parameters 'arg1' and 'arg2' from the point of handler set up, and 'arg3' and 'arg4' from the signaller. The 'type' parameter indicates whether the signaller is prepared to receive control back after the exception has been handled.

An exception handler exits by calling:

```
EX.exit(how, result1, result2)
```

The 'how' parameter indicates what should be done next, e.g. call the next handler, return to the signaller etc.

----

[13] Some of the argument lists in these examples have been slightly simplified in the interests of clarity.

If no handler for an exception can be found, then the **EX.default** exception is raised, and a default handler, which prints out an error message and terminates the program, is provided for this exception. In addition, the exception **EX.finish** is signalled when the program is about to stop because of a call of the BCPL function **STOP()**; this allows packages to tidy up before termination.

This exception handling mechanism has much of the power of more complicated mechanisms built into languages, and is simple to implement as a library package without altering the compiler. The current implementation is about 300 lines of BCPL. It does, of course, require discipline in its use. It provides no support at all for mutually suspicious packages, and care must be taken to avoid breaking scope rules (as it always must be when addresses are handled in BCPL).

An unfortunate limitation in the current scheme is the lack of an **EX.unwind** exception to indicate that a stack frame is about to be abolished (perhaps because another exception handler has terminated by requesting a non-local jump). There is no fundamental reason for not providing this; the author has suggested that the facility should be provided, and it is possible that it will be implemented in the next version of the package.

4.2.5 Levin's proposal

Levin has proposed a somewhat more general mechanism than those already described [Levin 77]. One of the most important properties of his proposal is that it addresses a situation that other mechanisms do not, namely that the best place to handle an exception may not be on the current call stack. The terminology Levin employs for this is that existing exception handling mechanisms look for exception handlers by following a 'calls' relation, whereas they ought to be following a 'uses' relation; the distinction is often forgotten because the two relationships coincide in simple systems. A context 'uses' another context if it can invoke the functions it provides. In the presence of shared abstractions, the 'calls' hierarchy and the 'uses' hierarchy do not necessarily coincide.

Levin distinguishes clearly between exceptions which are raised on structure conditions and those that are raised on flow conditions. The difference is that structure conditions are raised relative to the data hidden behind an abstraction, whereas flow conditions are raised on the call of a particular function provided by the abstraction [14]. The practical effect of this is to assist in determining which handlers can be called when a condition is raised. Such handlers are said to be eligible [15]. For a flow condition, eligible handlers may be found in only one context (the caller). For structure conditions, handlers may be found in many contexts - i.e. in all of the places where the relevant instance of the abstraction can be used. Earlier exception handling mechanisms did not make this distinction.

One of the problems that arises when more than one exception handler is eligible to handle a condition is that of deciding which to invoke (and if more than one is to be invoked, in what order). The algorithm for determining which of the eligible handlers to invoke is termed a selection policy. Levin describes a number of possible selection policies, and assumes that an implementation would provide a few fixed policies, a particular policy being applied (statically) to each condition.

Unlike most exception handling mechanisms, Levin's does not permit an exception handler to abort the signalling routine. It is therefore safe for

----

[14] The example given is that of manager of a particular format of file. The exception **file_inconsistent** is a structure condition raised on a particular instance of the file; the exception **file_read_only** is a flow condition raised on a call of the function **file_write**. The former is of concern to all users of the file; the latter matters only to the programmer attempting to perform a write operation.

[15] The definition of 'eligible' also includes rules for restricting the scope of handlers by lexical nesting. For full details, see [Levin 77] section 4.6.1.

a module to raise an exception while its private data structures are in an inconsistent state. Provided that the handler does not enter an infinite loop, control is guaranteed to return to the signaller. A handler can, however, specify a local transfer of control which is to occur when the context of the exception handler is later resumed.

4.2.6 Limitations of the linguistic approach

Although the author considers the linguistic approach to be a useful contribution to the handling of exceptions, it is not seen as providing a full solution. Good language mechanisms (in general) enable programmers to write programs which are clear and comprehensible, and possibly provable. Languages provide a set of abstractions which enable a programmer to think in high level terms. But it must not be forgotten that the low level implementation, although largely hidden from the programmer, is still present and is the source of many of the exceptional conditions that the language system is to handle.

High level languages are now used for the implementation of many operating systems, as well as for the production of applications programs. However it is rare for the whole of an operating system, together with the applications programs which run under it, to be written within a single language system. When considering the interactions between separate programs, possibly written in different languages, the description of those interactions have to be expressed in low level terms. Once the operating system has informed a program of some exceptional condition, a high level language system can help the programmer to deal with it cleanly. But the operating system writer still has to make the decisions about when to raise exceptions, and must provide the basic facilities which allow programs to handle them.

An obvious remark which is nevertheless worth making is that the exception handling mechanisms of a language can only help if they are used. Morris has pointed out [Morris 78] that the use of MESA signals can be very dangerous. Any call on another procedure can raise a signal, and is therefore a potential exit from the current procedure. Although it is possible (in MESA) to provide catch phrases to avoid losing control, it is rare for programmers to be disciplined enough to do so. Bugs caused by errors of this nature are likely not to be noticed for some considerable time.

A further limitation of the linguistic approach is that it provides little scope for the enforcement of constraints additional to the basic exception handling, such as the imposition of real time limits on the handling of certain exceptions. Such constraints are essential in real operating systems; language designs rarely allow this topic to be addressed.

A final danger with providing a purely linguistic basis of exception handling is that one is relying on the compiler and runtime system of the language to implement correctly the abstractions it claims to implement. By providing lower level mechanisms, and regarding the language system as simply a tool, it is possible to design an operating system whose exception handling will continue to work even if the language system used for user programs is incorrectly implemented.

4.3 Proposals made in previous research

The need for mechanisms to enable domains to protect their integrity has been recognized for many years. An early paper by Lampson [Lampson 69] considers the problem of handling traps and interrupts in a capability based protection system. A simple exception handling scheme is proposed, which allows domains to handle their own exceptions, propagating the exception to the caller if there is no suitable handler. Time limits and attention signals are dealt with by converting them into traps. The need for a domain to perform indivisible action is catered for by allowing processes to become non-interruptible for a short time.

The more stringent requirements placed on operating systems by complex program structures are expressed in a paper by Needham [Needham 71]. One of the main points made in this paper is that it should be possible for all faults to be handled within the user's program; it is not sufficient for the operating system to force conventional error handling on the user. The Titan supervisor 'private monitor' (see section 4.1.4) is quoted as an example of the approach required. A more general mechanism is also proposed to cope with there being more than one program on the call stack, each of which needs to do its own fault handling.

These requirements have also been pointed out by Parnas and Würges [Parnas 75]. This paper considers the treatment of exceptions (called 'undesired events') in a hierarchically structured system, and offers general techniques for the handling of exceptions and their propagation across levels of abstraction. The relevance of exception handling techniques to the more general area of software reliability has been pointed out by Melliar-Smith and Randell [Melliar-Smith 77].

A paper by Goodenough [Goodenough 75] presents a survey of exception handling, but is predominantly a discussion at the programming language level rather than the system level. The work of Levin in this area [Levin 77] has already been discussed (see section 4.2.5).

A detailed exposition of techniques for handling exceptions may be found in [Lindsay 77]. Lindsay explicitly defines an exception as 'the reported failure of an operation invocation to produce the specified effects of the operation'. Mechanisms are provided, at the system level rather than the language level, to report exceptions across subsystem boundaries.

An interesting feature of the scheme proposed is that it attempts to treat all exceptions uniformly. In many systems, low level exceptions such as virtual memory faults are dealt with specially. Lindsay sees such things as exceptions reported by the 'basic processor', and suggests that they should be handled using the same mechanisms as any other exception. In order to do this there must be restrictions on the setting up of handlers for certain exceptions; in particular it must be possible for handlers for some exceptions to be imposed. This ensures that the user cannot trap the exception before the operating system has had a chance to handle it. This uniform treatment of exceptions has been taken as far as to allow the exception processor to deal with exceptions which occur within itself.

The limitations of Lindsay's approach come from the assumptions made. Calls of subsystems are seen as isolated events which either succeed or signal an exception. The possibility that subsystems might retain state information between invocations has not been considered. The treatment of asynchronous events, particularly in systems which support a non-hierarchical organization of subsystems, is left for future research.

The problems of resource management and maintaining the integrity of protection domains have been considered by Taylor [Taylor 78]. He views execution time (whether measured by instruction cycles or by real time) as a resource, for which a domain may have a capability. Enforcing CPU time limits, console attention etc. is done by revoking the CPU time resource. He considers the problem of allowing protection domains to maintain their integrity in the face of such revocation, by arranging to allow the domain certain time guarantees. The time guaranteed is not assumed to be a fixed amount, but may be varied (within limits) by negotiation with the operating system. A domain is expected to ensure that the time guaranteed to it is sufficient to allow a standard state to be restored. Taylor's scheme allows for the possibility that domains may have to call others in order to restore their standard state, but does not consider the possibility that a domain may need to return in a non-standard state.

The problem of allowing protection domains to relinquish control while in a non-standard state has been largely neglected in previous work. As will be seen in chapter 6, part of the problem is concerned with the loss of access to objects by the destruction of capabilities. Some work has been done in this area, notably in the Hydra system [Cohen 75]. The problem is dealt with in this system by allowing type managers to retrieve capabilities for objects of the appropriate type which have become unreferenced. This subject will be returned to briefly in section 7.5.

5. The CAP operating system

5.1 Background

The practical work done by the author has been in the operating system for the CAP computer, a machine built in the University of Cambridge Computer Laboratory for research into capability based memory protection. The hardware is fully described in [CAP 1], the microprogram in [CAP 2, Walker 74], and the operating system in [CAP 3, Slinn 77]. In addition, an account of the whole of the CAP project, with bibliography, may be found in [Wilkes 79]. Since considerable reference material is readily available, only a brief summary of the relevant parts of the CAP system will be given.

5.1.1 Addressing and protection

The machine itself is a 32-bit word addressed computer with (at the time the work described was being done) 192K words of core store. It is controlled by a microprogram, which resides in 4K 16-bit words of fast, volatile memory. The microprogram provides the user with a fairly conventional instruction set, together with the support of a capability based addressing and protection scheme.

5.1.1.1 Addressing

The addressing is worth discussing in more detail, since it lays the foundation for the support of the protection domains which are so important in the operating system.

The format of a virtual address is shown in figure 2.

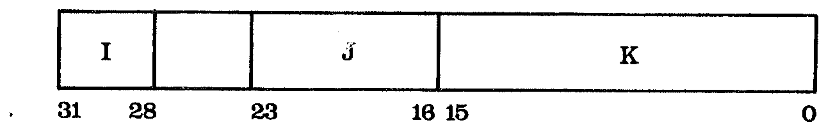

Figure 2: CAP virtual address format

I is a 4 bit number which specifies one of 16 <u>capability segments</u> which may be accessible at any moment.

J is an 8 bit index into that capability segment, and is used to select a particular capability.

K is a 16 bit word offset into the segment specified by the capability.

The more significant part of the address, (I, J), is called the <u>segment specifier</u>.

Capabilities do not themselves contain the absolute addresses of the store segments they represent, but instead contain a pointer to an entry in a <u>resource list</u>. The capability segments are also specified by entries in the resource list. Figure 3 illustrates the relationship between the resource list and the capability segments.

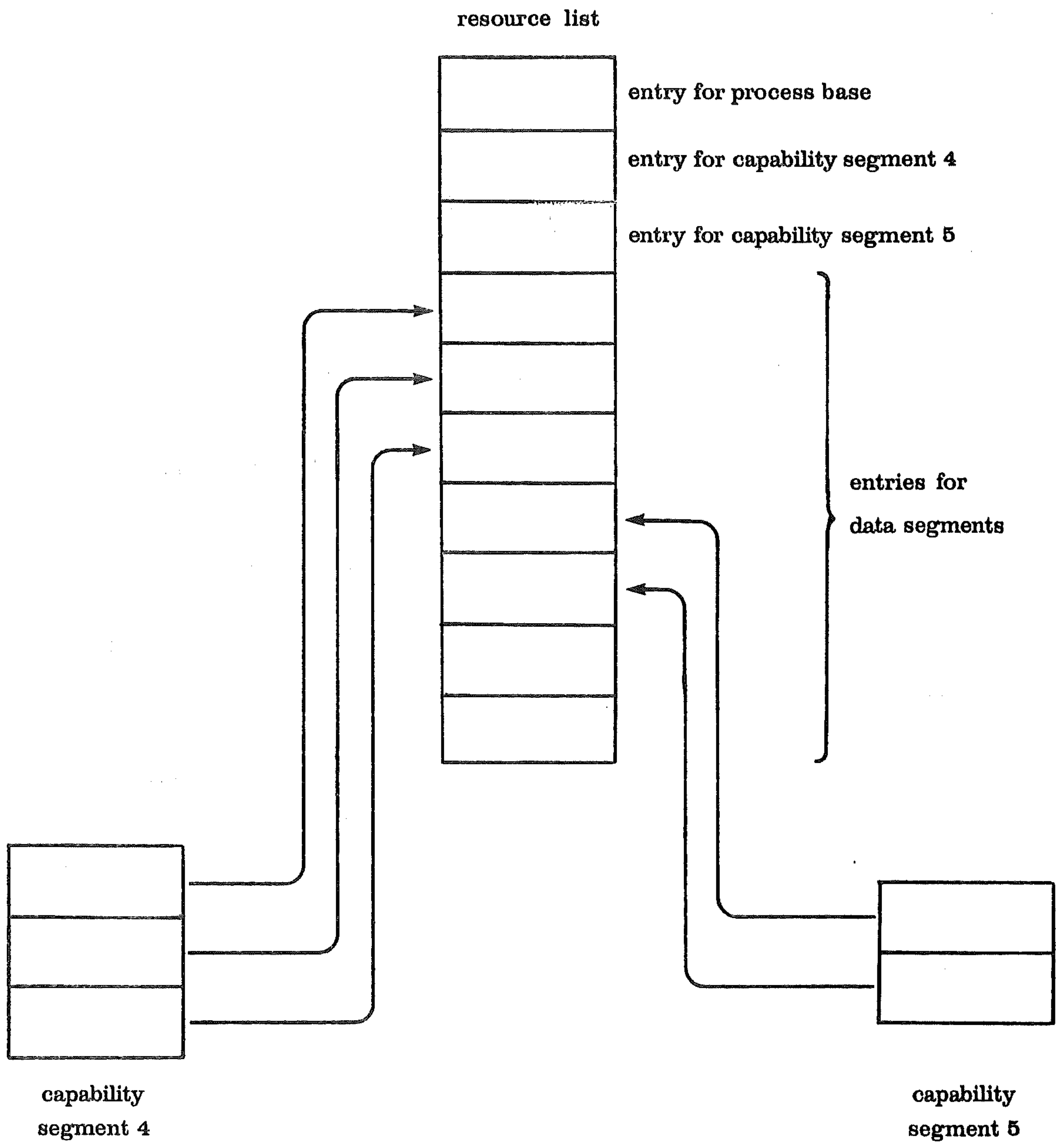
resource list
entry for process base
entry for capability segment 4
entry for capability segment 5
entries for data segments
capability segment 4
capability segment 5

Figure 3: CAP resource list and capability segments

5.1.1.2 Process structure and process resource lists

In order to describe what the resource lists contain, the process structure must be considered. The CAP microprogram supports a hierarchy of processes; any process can set up sub-processes and coordinate them. When the microprogram is loaded, a single process is created; this is called the Master Coordinator. Part of the bootstrapping operation specifies the absolute location of its resource list, which is called the Master Resource List, or MRL. The identities of the current capability segments of a process (as resource list offsets) are stored in its process base. The process base of the master coordinator is identified to the microprogram as an MRL offset during initial loading. MRL entries all contain the absolute addresses of the segments they represent. Both the capability for an object and its MRL entry contain size and access information, the capability acting as a refinement of the MRL entry.

A sub-process is created by executing the enter-subprocess instruction (**ESP**), quoting two arguments: a capability for a segment which is to become the resource list of the sub-process, and a number which specifies which entry in this new resource list is to be the process base segment. This new resource list is called a Process Resource List, or PRL. From the point of view of the coordinator, the process base and PRL of a sub-process are just data segments. When a sub-process is executing, its process base and PRL are not usually made available as data segments within its own virtual address space, and are thus highly protected data structures.

The PRL contains entries for the segments available to the process; each entry gives the virtual address of that segment in the virtual address space of the coordinator of the process. Hence to find the absolute address of a segment, a chain of capabilities and resource list entries must be followed; this chain is represented by the upward pointing arrows in figure 4. Every capability or resource list entry can provide size and access refinement.

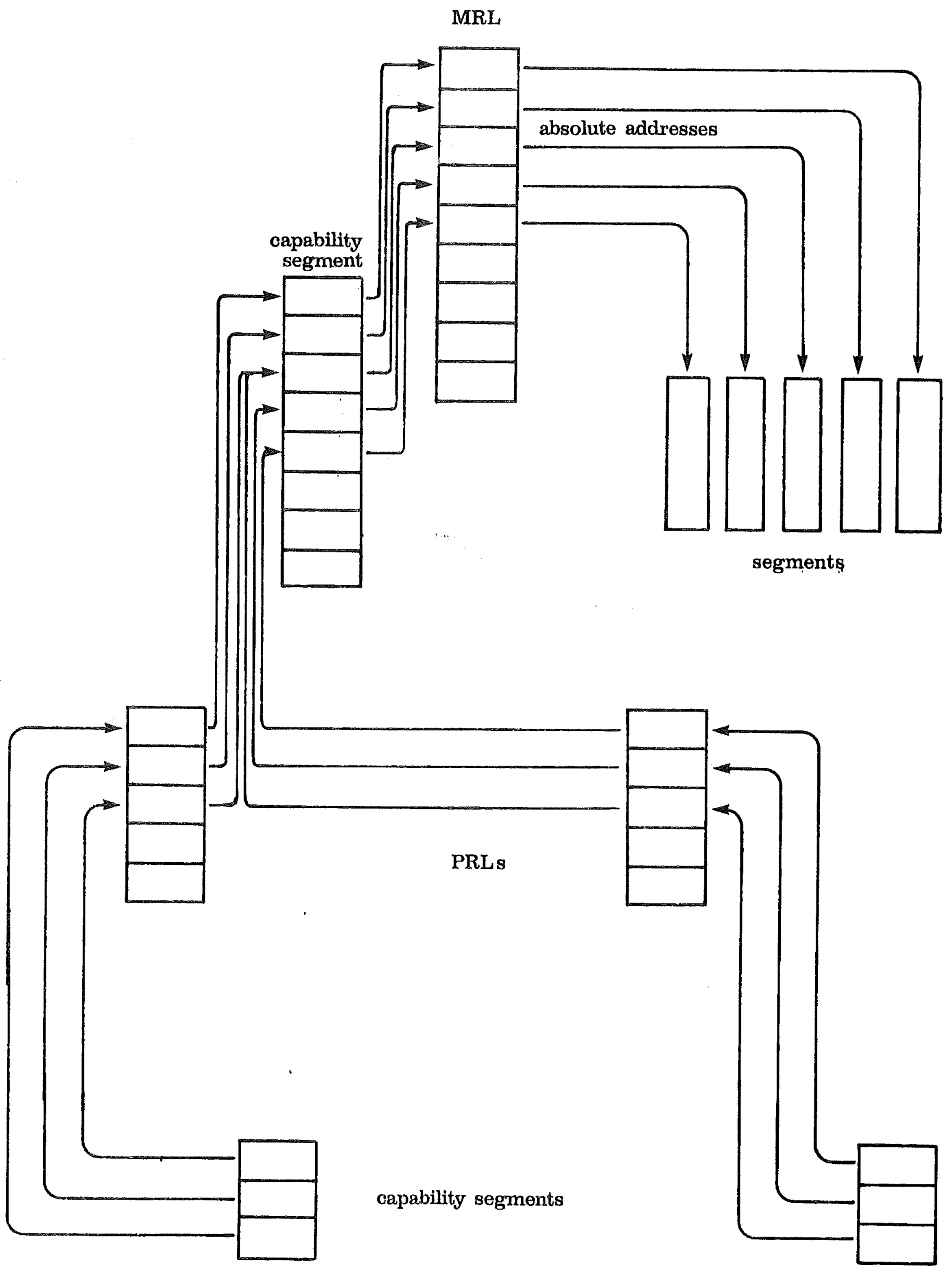

Figure 4: CAP process structure

Although not strictly relevant for the purposes of this dissertation, it should be pointed out that the chain of capabilities and resource list entries is not followed every time a virtual address has to be translated into an absolute address to put onto the store bus. CAP has a set of 64 capability registers, which acts as a cache for frequently used capabilities. If a capability is in the cache, the address translation is done entirely by the CAP hardware. If it is not, the microprogram performs the full address translation computation and loads the result into the cache. This microprogram operation is called the reset cycle: its operation will not be further described. It is sufficient for the present purposes to note that apart from the increase in speed that it offers, it is transparent to the programmer.

5.1.1.3 Changing protection domains

Up to now, it has been assumed that there is only a single address space in each process, which would imply a hierarchic protection structure corresponding to the process structure. This is not in fact true; it is possible to change protection domains within a process. A protection domain is represented by a capability called an **ENTER** capability. (This is not recognized by the hardware, only the microprogram.) The PRL entry for an **ENTER** capability contains three numbers, which are PRL offsets of capability segments. When a new protection domain is entered (using the **ENTER** instruction), the process base entries specifying the PRL offsets of capability segments 4, 5 and 6 are saved on a stack, and new ones are loaded from the **ENTER** capability. The stack is called the <u>C-stack</u> [1], and is quite separate from any stack a language system uses. Thus the meaning of all virtual addresses with I = 4, 5 or 6 (see fig. 2) is changed. The old program counter is also saved on the stack, and the domain is forced to start executing at a fixed virtual address in one of the new segments. The

----

[1] C-stack stands for capability stack.

RETURN instruction undoes all of this, causing the calling domain to resume execution from where it left off.

In addition, a mechanism is provided to allow capability arguments to be passed between protection domains. (Numeric arguments are simply passed in the machine registers.) The microprogram provides an instruction to create a new capability segment, number 3, (on the C-stack), and capabilities can be moved into it. After an **ENTER**, these capabilities appear in capability segment number 2 in the new address space. If capability results are to be returned, they are moved into capability segment number 2 before the **RETURN** is executed, and will then become available to the caller in capability segment number 3.

These domains, hidden behind **ENTER** capabilities, are called protected procedures, and are extensively used in the CAP operating system, to provide both protection and modularity. It can be seen that the interface between protected procedures consists of a (usually) small, well defined set of numbers and capabilities. This narrow interface (compared, say, with the typical procedure interface in a programming language) encourages careful definition of the interfaces between the modules of the operating system.

An **ENTER** instruction causes three capability segments to change their identities (rather than just one) to enable controlled sharing of capabilities between protected procedures. The three capability segments concerned, 4, 5 and 6, are called the P, I and R [2] segments. As an example of controlled sharing, two **ENTER** capabilities may refer to the same P capability segment, but different I capability segments. The former can contain capabilities for objects to be shared between the domains (such as code segments); the latter can contain capabilities for unshared objects (such as workspace segments). This has allowed the implementation of a rudimentary form of protected object - the representation of the object,

----

[2] P, I and R stand for procedure, interface and resource respectively.

together with the code to manipulate it, being hidden within the protection domain.

5.1.2 Operating system structure

In order to set the scene for the remaining sections, an overview of the relevant parts of the operating system structure will be given. Much more detail may be found in [Wilkes 79]. The operating system employs just two levels of the process hierarchy: the master coordinator schedules a number (about 20) of sub-processes. Some of these are system processes; others form a pool of processes in which user programs are run. The master coordinator is a single protection domain; each sub-process consists of several different domains.

Each protection domain of the operating system is a separate program - most are written in Algol68C [Bourne 75] - and is independently compiled. The data structures described in the previous sections are constructed when the various program binaries are bound together to form a system image.

The operating system manipulates the capability structure (in appropriately privileged protection domains) to implement a virtual memory system and filing system. Dynamic linking of new protection domains from the filing system is supported. This requires the creation of new entries in process resource lists, and the construction of capabilities to refer to them.

Much of this dissertation is concerned with what happens when capabilities for protection domains are deleted. In the CAP system, capabilities can be deleted simply by overwriting them with a capability for something else. This is purely a microprogram function; it does not involve any operating system action at all. The effect of this is that entries in the process resource list become unreferenced. In order that the PRL should not become unreasonably large [3], it is necessary to garbage collect it from time to time.

The PRL garbage collector was written by the author, and is a protected procedure which is called whenever a new PRL entry is wanted but none is immediately available. It is necessarily a highly privileged domain, since it must be able to scan all the capabilities which can be used by any domain in the process, and compute a map of the PRL entries which are referenced. When it has done this, those entries in the PRL that can never be accessed again will have been identified. Before the entry can be reused, certain housekeeping operations may have to be performed (e.g. the entry may have referred to an object on disc which can now be deleted). The PRL garbage collector makes use of other modules in the operating system to do this, and then makes the freed PRL entries available for reuse. The garbage collection of the PRL occurs as a side effect of a system call (such as requesting the creation of a new segment) and its operation is transparent to the user.

5.1.3 The use of protected procedures

A few words are appropriate about the way in which protected procedures are used in practice in the CAP operating system, since the style of use provided much of the motivation for the work described in this dissertation. As described above, protected procedures appear as subroutines; when they are called, they begin executing at a fixed virtual address, and the microprogram retains no state information about the domain when it returns.

For many operating system procedures, a subroutine style of use is not particularly appropriate. Many have initialization work to do (such as setting up of communication channels) and it is important for efficiency reasons that such initialization is done only once. It was observed, however, that a protection domain can retain state information between

---

[3] There are, in fact, implementation limits on its size anyway.

calls in its private data segments. Although each entry to the domain initially goes to the same place (offset zero in the main code segment), the program may examine data structures left by the previous call and make further decisions on the basis of what it finds. If a test for whether the domain is already initialized can be devised, it is easy to avoid repeating the initialization on subsequent calls.

In fact, programs on the CAP do not rely on ad hoc tests to organize this. Instead, a convenient mechanism has been incorporated into the Algol68C [4] runtime system to enable such decisions to be made easily. The language libraries provide a procedure **enter** which is used to transfer control to a domain. The first time a domain is entered, it begins executing at the beginning of its main program - each protected procedure is mapped onto a complete program, rather than a language procedure. If the program terminates in the normal way (by control reaching the end of the main program), the protection domain returns, using the **RETURN** order. A somewhat different effect can be achieved by using the library procedure **return**. When **return** is called (in the Algol68C sense) the runtime system sets a special marker on the Algol68C stack, which is a private data segment of the protected procedure. It then returns to the calling protection domain using the **RETURN** order. When the domain is **ENTER**ed again, the marker on the stack is noted, and the Algol68C environment is restored to the state it was in just before the **RETURN** order. The called program will then resume from immediately after the call of the **return** procedure. Returning to the caller thus appears as a subroutine call. The relationship between the two domains is thus very similar to a coroutine structure.

----

[4] and also BCPL [Richards 79], though this is not used in the operating system

The gross structure of many programs in the CAP operating system is:

```
BEGIN

     ... initialization code ...


     DO
        CASE first argument
        IN

               ... services provided ...

        ESAC;

        return (results ...)
     OD

END
```

It can thus be seen that the CAP protection system, and the manner in which it is used, closely resembles the formal model presented in chapter 3.

5.2 The inter-process communication system

This section describes some work done in the CAP operating system which illustrates some of the integrity problems which arise in a domain based system and how they are typically solved. The CAP operating system has an inter-process communication mechanism which is accessed by means of capabilities. A protected procedure, available to anybody, exists for setting up message channels. This procedure performs two functions - checking software capabilities [5] for message channels, and establishing the communications path between processes. The result of a successful call to this procedure is one or more capabilities giving access to the more

---

[5] A software capability is a protected bit pattern which will be accepted by a piece of software as evidence that its caller is allowed to invoke some function. Software capabilities are sometimes called <u>permission capabilities</u>.

primitive message passing facilities (e.g. **send data message**). The construction of capabilities for a message channel involves the allocation of workspace. The messages themselves are held in dynamically allocated buffers, but some additional space is required to hold data describing the state of particular communication channels. This workspace must be in an address space accessible from all processes, and is in fact taken from the master coordinator's workspace (in Algol68C terms, a HEAP generator is used). The workspace is referred to from the sub-process by means of PRL entries and capabilities which are created as part of the setup operation.

5.2.1 A practical problem: reclaiming store

In early versions of the operating system, there was little attempt to manage this workspace - it was assumed that message channels would be set up between system processes at system initialization time and that there would be little need for such operations once the system was running. However, it was later found extremely convenient to enable programs which were dynamically linked from the filing system to engage in communication with system processes. The particular case which caused most embarrassment was that of setting up a channel to enable a linked program to send a message to a system process and receive a reply. A data structure about 5 words long has to be made for each channel to handle the replies to messages. Although this data structure is not large, it is clear that if the space obtained when a channel is created is not recovered, the coordinator will quickly run out of workspace entirely (it typically has only 2K words available).

A problem arises here because the master coordinator is not truly the master of its own resources. Much of the message system code is part of a protected procedure called **ECPROC** which runs within a sub-process. This part of the message system interacts intimately with the coordinator. The coordinator cannot manage its storage without considering references to that storage from other programs - hence a simple garbage collector will not do (nor would it be desirable for other, essentially independent, reasons). If coordinator space is to be recovered, one must rely on it being explicitly returned by sub-processes.

If all programs which set up message channels could be relied upon to perform a 'release channel' operation when a channel was no longer required, then this space could be recovered. This is not really satisfactory for several reasons. Firstly, it is not always possible to guarantee that a program can regain control before it is abolished (this is precisely the problem to be addressed in chapters 6 and 7). Secondly, it would restrict the availability of the message system to those programs which could be guaranteed to do the necessary tidying up operations. It is reasonable to expect programs to look after their own resources, but not those belonging to other protection domains. Thirdly, it would give no protection against a program which appears to perform correctly (and under normal circumstances actually does), but which has a bug in it which causes it not to tidy up correctly under certain error conditions. This would give a very slow drain on system resources, probably causing it to crash after a very long run.

5.2.2 The solution

The decision was therefore taken that the operating system should arrange to recover the coordinator heap space whenever it found that a message channel had fallen into disuse. The problem divides itself into two parts - discovering what can be deleted, and arranging for the recovery of the space.

The latter operation is straightforward. The data structure on the Master Coordinator's heap is called a channel. Since the Algol68C system does not allow objects to be returned to the heap, the coordinator was changed to keep a chain of free channels. When a new channel is required, it tries to obtain one from the free chain, and only if this is empty does it obtain one from the heap. A new function was added, called **delete message channel**, to put an existing channel on the free chain.

The work of deciding which channels ought to be deleted is done from within the sub-processes. Capabilities for message channels are of three types - send, receive and reply. Send and receive capabilities refer to PRL entries in the usual manner; a reply capability is a temporary repository for information associated with messages which require replies, and will refer to a PRL entry if a reply is outstanding. Disused message capabilities are detected in the PRL garbage collector.

The PRL garbage collector had some knowledge of message capabilities from the outset, since it must be aware of all references to PRL entries. It was modified so that it attempted to delete the message channels which belonged to those PRL entries it had found to be unreferenced. This is not as simple an operation as might appear - since the message channel has presumably been used to send messages, there may still be messages in transit. If a program sends a message which requires a reply, and dies before receiving the reply, then the message will still be in the system when the message channel capabilities are garbage collected. It is therefore necessary to clear out any pending messages, otherwise the system will be liable to run out of message blocks. As **ECPROC** is the only program which touches message blocks, the PRL garbage collector passes all unreferenced message channels to **ECPROC**. This program decides whether or not the channel can in fact be disposed of (there are some cases in which **ECPROC** may detect that asynchronous activity associated with the channel may be going on in other processes). If it can, it deletes any pending messages, calls the Master Coordinator to recover the channel space, and informs the PRL garbage collector that the PRL entry can now be reused. If the channel cannot be deleted yet, the garbage collector simply refrains from reusing the PRL entry. It will be detected as garbage again next time the PRL becomes full, and the deletion tried again. Since the asynchronous activity is likely to have completed by this time, this retry will usually succeed.

5.2.3 Evaluation

In practice, this garbage collection technique works well. The garbage collection is being done anyway, to recover PRL entries, and it is very easy to perform extra operations on entries of particular types before they are reused. A number of similar problems are also dealt with by taking action when their capabilities are deleted. It is interesting to note, however, that this is a typical example of an operating system providing an ad hoc mechanism for its own benefit. The structure of the system has been disturbed because the PRL garbage collector needs to contain code which is logically part of the inter-process communication system. It would have been preferable to implement message channels as protection domains or protected objects, and provide the more general techniques to be presented in chapters 6 and 7 to deal with channels which became disused.

5.3 Attention handling

One problem which has to be considered in any multi-user operating system is that of arranging some means of terminating computations which are no longer desirable. This usually takes the form of a 'console quit' or 'break' key for online work, and a CPU time or real time limit for offline work. Facilities are usually provided to enable operators to cancel sessions and jobs. In this section, the term 'attention' is used to cover all of these.

It is also desirable to allow programs to be able to handle attentions. In the author's opinion, it should be possible for a program to be able to do at least a small amount of computation after any attention, even if the aim of the attention is to stop the program. On the other hand, it should not be possible for the program to ignore such an event entirely, since that would defeat the purpose of the mechanism.

It must also be noted that attentions are used for a variety of distinct purposes, which should not be confused. For example, an attention may be used to give asynchronous communication with a program such as a text editor, or it may be sent to a program in order to make it stop. The fact that a program may be handling attentions itself must not interfere with the latter use.

The remainder of this section describes how the author dealt with these issues in the CAP operating system. No claim is made that all of the problems have been solved; in particular, for reasons which will be given later, the implementation has not been carried out in its entirety. Nevertheless, sufficient is present to show that the general ideas do work in a real operating system.

5.3.1 Requirements of the attention mechanism

During the author's first year of work on the CAP operating system, most of the major internal components of the operating system were completed, and it became possible to run 'user programs'. At that time, it was considered quite important to get a usable user level system working fairly quickly, in order that the CAP system could be developed under itself, and the temporary operating environment, OS6, could be abandoned [Slinn 77].

Almost from the outset the CAP operating system was a multi-user one, and it soon became apparent that the casual attitude taken to the termination of undesirable computations on OS6 (i.e. restart the system) could not continue. Accordingly the author undertook to provide an attention handling mechanism.

A variety of mechanisms were considered, each with its advantages and disadvantages. The essential issues were recognized early on; the following quotation from an internally circulated document [Birrell 76] summarizes the initial aims:

"We require that the mechanism can achieve the following:

1. It must be capable of forcing return(s) from the offending procedures, albeit after a considerable delay.
2. It must be such that procedures managing resources and data structures are allowed to tidy up.
3. For efficiency, both in code and time, it should be such that the various system procedures need not take any special action.
4. There must be a limit to the enforcement of the event (e.g. 'break' need only take the process back to its command program, not to **STARTOP** [6]), but this limit must be flexible (to allow multi-procedure user written command programs), and protected (only **STARTOP** should be able to cancel 'CPU exhausted')."

Point 3 above perhaps requires some further explanation. The CAP architecture allows separate protection domains to exist within one process; control may pass between them using the **ENTER** and **RETURN** orders. Except to the extent that capabilities are explicitly shared or passed as arguments, the domains are fully protected from each other. This makes it much more natural for the operating system designer to provide operating system functions within user processes than it is in many other systems. Thus there are many parts of the CAP operating system which execute as part of an ordinary user process; these parts of the system often update system data structures and ought not to be terminated at arbitrary points. It is a feature of the CAP system that such parts of the operating system are not

----

[6] **STARTOP** is the initial protection domain of a user process, responsible for setting up the command program's environment and handling any faults which it might return.

marked in any way which would make them easily distinguishable from ordinary user programs - the only differences are that they have capabilities which are not made available to the general public.

At the time the attention handling was introduced, the many user and system programs in existence took no explicit measures to deal with attentions, since no means of doing so had been defined. It was considered important that some default attention handling should be provided for both system programs and user programs, despite the fact that their requirements are different and the operating system cannot distinguish between them. The means used to achieve the desired effect is described in section 5.3.4.

5.3.2 Implementation issues

In the early days of the project, it had been tentatively assumed that the existing fault handling mechanisms would be suitable for dealing with asynchronous events as well as with errors like protection violations or invalid parameters being passed to system procedures. A fault caused a jump to a particular location in the user's virtual address space, with sufficient information being provided to enable diagnosis and possible correction of the fault. This correction was termed <u>clearing</u> the fault, and a software capability was required to do it. Every fault had a number, and a 'fault capability' contained the largest fault number it could be used to clear; this was done so that the clearing of certain faults could be made privileged.

For a variety of reasons, this idea was abandoned. To start with, it seemed wrong to require that all faults should have to be arranged in some order of severity, especially as the number of fault codes used by the operating system is of the order of hundreds rather than tens.

Secondly, there is a fundamental difference between faults and attentions. Faults are almost always caused by something that the program has done; it is impossible to get a 'limit violation' fault without attempting to access a segment. On the other hand, attentions are genuinely asynchronous - since they originate from outside the target process. The fault handling mechanism imposed the restriction that a program could have only one fault outstanding at once; making attentions into faults would mean that a program would not be able to deal with an attention if it happened to be dealing with a fault, which would have been an unreasonable constraint.

Finally, a fault is regarded as being a property of a particular protection domain. Even though faults which have not been dealt with are passed back to the calling domain, the fault 'limit violation' is distinct from 'called domain suffered a limit violation and took no corrective action'. Attentions, on the other hand, are the property of a process: at the time the attention is issued it is not necessarily known which protection domain the target process is in. Since an attention is asynchronous anyway, the distinction between an attention which occurred while a particular domain was executing and an attention which was passed back from another domain is not so important.

For these reasons, it was decided to treat faults and attentions separately. Given a separate attention mechanism for enforcing external events, the reason for the existence of 'fault capabilities' would go away, and in fact, the ability to clear faults was subsequently made unprivileged.

The notification of an attention to a process was considered next. An attention usually originates in a terminal process, as a consequence of the user having pressed a 'break' key, possibly followed by some further interaction (the nature of which is not of immediate concern). Attentions may arise from other sources, such as an 'operator cancel' program, but in all cases, the attention originates in a different process from the target process. In principle, the attention could be communicated to the target process using the ordinary inter-process communication system provided in the operating system (the message system is discussed in a different context in section 5.2).

Unfortunately the message system does not have the right properties to enable it to be used for this purpose. Of the means available for notifying a process of an attention, the simplest is for the process to poll for them, and the message system would provide just this. However, although polling may be convenient for those programs which want to accept attentions at certain well defined times, most programs are not of this nature. Great emphasis is given to the fact that programs must be able to trap attentions, but it must be admitted that the vast majority of programs in ordinary use do not need to do so, and in these cases an attention should terminate the program as soon as possible. The programmer should not to have to make any effort to ensure that this happens.

The only case in which the use of the message system would have been appropriate is for the notification of non-enforced attentions, such as may be given to a text editor. Effectively this would have been entirely separate from the general attention handling, and was not done on the grounds of economy of mechanism.

The next simplest means of notifying an attention to a program is the enforced jump, and this is in fact the way attentions are notified in the CAP system. In these days of high level languages and structured programming this may seem rather unpleasant, but as will be seen later, the machine level details of the jump can be effectively hidden from the programmer.

It was initially proposed that the programmer should be able to choose whether or not the program was to be notified of attentions. If he chose to have them notified, then he would be guaranteed a certain tidy up time after such notification; otherwise the time limit would be liable to expire and his program be terminated without ceremony. The reasoning behind this proposal was to ensure that the system procedures mentioned earlier would not have to be changed in any way; it was assumed that their actions were always of short duration and that they could always ignore attentions. It also meant that a program could avoid suffering an attention while it was still initializing itself (when, for example, a high level language runtime system might not have a stack available).

The main drawback of such a scheme is that it has the wrong effect for user programs. If they take no action to have attentions notified to them, they will terminate, but only after an unnecessary delay. Although default action can be taken by language systems, it seemed wrong as a matter of principle that system programs should have to take genuinely no action but user programs should have to make a special request to the operating system in order to ensure that sensible default action occurred.

5.3.3 Implementation decisions

The author therefore decided that at the machine code level at least, it would not be possible to ignore attentions, and that the program must be prepared for control to be forcibly diverted at all times. In practice, this is no great hardship. When an attention occurs, the jump is always to a fixed virtual address [7]. At the machine code level, very little of the programmer's environment is disturbed by an attention; all the registers except the program counter itself are unchanged. The old value of the program counter is available in a small segment which is accessible to the domain at another fixed virtual address.

In this way, it is thus possible to deal straightforwardly with the simple cases of attention handling. If a program has no interest in handling an attentions at all, and simply wants to stop as soon as possible, the attention jump location need contain only a single order - **RETURN.** The attention will then be passed back to the calling domain, which is notified of it in precisely the same manner. Alternatively, if a short lived system program wishes to ignore attentions entirely, two orders are sufficient to

----

[7] PO,2 - offset 2 in the protected procedure's initial code segment. Offset 0 is the initial entry point and offset 1 is the address to which control is diverted to notify a fault.

reload the program counter register from the resume address in store and effect an immediate continuation of the program as if nothing had happened.

In practice, neither of these simple solutions is used much; largely because CAP programmers do not usually write in machine code. Analogous techniques are, however, used within the high level language interface to be described later.

Signalling an attention to a process is simple, since it consists almost entirely of alterations to the process base of that process. These can be done in the master coordinator. When an attention has been signalled to a process, it is said to be in attention state. Some complications arise from the fact that changes of protection domain while a process is in attention state must be treated specially, in order to ensure that the attention is notified to each domain. The microprogram provides a facility to enable a system routine to take control on domain changes [8]. The remainder of the work is done in the same protection domain as fault handling (because the same capabilities are needed to do the two jobs). The system function which signals an attention is protected by means of a software capability.

The operating system guarantees that while control remains within the current protection domain, further uses of the **signal attention** function will not cause a second enforced jump. This avoids the need for the operating system to maintain a stack of resume addresses, and means that the user's attention handling routines do not have to be designed in such a way that they can be called recursively. The attention routine may do one of two things: it may tidy up the current domain and **RETURN** to the calling domain, or it may attempt to 'clear' the attention.

---

[8] Specifically, the facility is that a coordinator entry may be forced whenever the current domain **RETURN**s - this entry is called a **RETURN** trace. Tracing of **ENTER**s is not provided; this deficiency will be considered further later.

The former case is straightforward, and may often be achieved by jumping to the resume address, thus effectively ignoring the attention. Alternatively, the program may jump to a routine specially provided to tidy up and **RETURN**. The tidying up operation may involve **ENTER**ing other protection domains, which will be in attention state as soon as they are **ENTER**ed. Domains must therefore be prepared to handle attentions before they have executed a single order. In the current implementation, **ENTER**ing a domain is not in fact trapped, and this case does not arise. A full implementation would require this to be done, and minor changes to the CAP microcode would make this easy.

Clearing an attention resets attention state, which makes the current protection domain liable to receive another attention. The operating system does not provide a specific mechanism to enable the program to resume from where it left off, since this can be achieved easily using ordinary instructions. It follows from remarks made earlier that clearing an attention cannot be an entirely unprivileged operation. Attentions are of differing degrees of severity, and different capabilities are required to clear them.

When an attention is signalled, it is given a level, which is simply an integer. A **clear attention** software capability contains a number indicating the maximum level of attention it is able to clear. Note that this is similar to the original mechanism proposed and later rejected for the clearing of faults. This use of a simple hierarchy is acceptable in the case of attention levels since there are orders of magnitude fewer of them, and it is very easy to arrange them into order.

There is a simple optimization in the scheme as implemented: the lowest level of attention, zero, can always be cleared without quoting a clear attention capability. This effectively makes a level zero clear attention capability globally available. It is the level zero attention which is commonly used for giving 'interrupt' signals to unprivileged utility programs, and it is convenient for such programs to be able to call **clear attention** without needing to quote any particular permission capability.

A call of **clear attention** may succeed or fail, depending on the level of the attention and the capability quoted; success or failure is indicated by a return code. In fact, the only way a program can infer the level of an attention is to attempt to clear it, and observe the result. This feature is partly a matter of principle (there is no need to know the level of an attention), and partly to enable multiple attentions to be handled correctly. Although an attention in attention state does not cause a forced jump, it may cause the level of the attention to be increased. If the program could discover the level of an attention without attempting to clear it, it could make a decision which would later become invalid if the attention level were to be increased.

If a call of **clear attention** succeeds, the program can continue, but if it fails, it should return to its caller as soon as possible [9]. This continues until a protection domain is reached which can clear the attention. In the CAP operating system, the protection domain **STARTOP**, the initial domain in every user process, has a capability which enables it to clear any attention. An attention which gets this far has the effect of terminating the user's session or job. The program which is made available to the system operator for cancelling user sessions simply signals an attention which only **STARTOP** can clear to the appropriate process.

A capability for clearing intermediate levels of attention is given to the command program, thus providing users with the ability to break out of commands.

----

[9] Eventual return should be enforced - see section 5.3.5.

5.3.4 High level language interface

The attention scheme described has been mapped fairly cleanly onto high level languages - Algol68C and BCPL in particular. The general techniques used on CAP to provide access to system functions from an ordinary high level language are fully described elsewhere [Birrell 77]. A brief description of the facilities available to the Algol68C programmer is given here for completeness.

The language has not been changed in any way; all of the facilities provided by the operating system are accessed via the runtime libraries. These libraries are shared segments of code which form part of complete Algol68C programs. The programmer may decide whether attentions should interrupt his program or be ignored. The procedures **allow attention** and **ignore attention** switch between these two modes. Ignoring attentions is implemented by jumping back to the resume address in the manner described earlier, except that the Algol68C system notes the occurrence of an attention by setting a flag which can be read using the procedure **attention happened.** Thus programs which ignore attentions for most of the time can still poll for them periodically. If attentions are not being ignored, then the variable procedure **attention routine** will be called. A default routine is provided which tries to clear the attention, gives a backtrace if it succeeds and terminates the program if it does not. A programmer who wishes to have more complicated attention handling simply assigns a procedure of his own to the procedure variable. (This may involve a technical violation of the scope rules of the language; in practice this is harmless). A simple procedure, **clear attention,** is also provided as an interface to the **clear attention** function.

Most programs use the default attention handling. System programs are usually compiled in a simpler environment which has a different default - attentions are 'ignored'. When attention handling was introduced, most programs did not have to be changed - they were simply recompiled to use the new libraries. Note also that the **ignore attention** and **allow attention** routines may be used to bracket short critical sections during which an interruption would be inconvenient. The **allow attention** procedure will call **attention routine** if **attention happened** is set, so that an attention which occurs during a critical section is not lost.

5.3.5 Time limits

Little has yet been said about the provision of time limits in attention state. This is because the high level language interface to the attention mechanism has been so successful in providing sensible defaults that time limits have not been implemented. Other practical reasons have discouraged the implementation of timeouts, not least of which is the fact that the CAP processor itself does not have a real time clock, and only a low resolution clock is available as a peripheral device. Nevertheless, many of the details of timeouts have been worked out on paper, and there do not appear to be any fundamental problems.

The primary requirement to be met is that a program should be allowed a certain time interval in which to deal with the attention, after which it is legitimate for the operating system to terminate its execution abruptly. In practice, the user's computation consists of a number of protection domains, and it is important to ensure that the attention handling of one domain cannot adversely affect that of another. The author considers this to be just as important as ensuring that the domains cannot overwrite each other's memory.

At first sight there would appear to be no problem here, since a separate timer could be maintained for each active domain. This scheme would work if the only domain changes which were allowed during attention handling were **RETURN**s. Each domain would perform some computation, return to its caller (either voluntarily, or forcibly when its timer expired), and the timer would be reset for the benefit of the new domain. If the maximum time allowed in attention state is **t**, then the total time until the attention reaches a system domain which is guaranteed to handle it will be **nt**, where **n** is the number of active user domains. It is worth noting that in the CAP system, the value of **n** is limited to some fairly reasonable value (less than ten) by the size of the protected procedure activation stack (C-stack). This observation is important because the system should provide a good response to the user who wants his program stopped, as well as allowing the program to maintain its integrity.

However, it would not be reasonable to restrict domain changes during attention handling to **RETURN**s. A typical attention handling routine may quite reasonably need to **ENTER** other domains as part of the tidying up operation. For example, in the CAP system, an input or output stream is represented as a protection domain which has bound into it the representation of the stream. Closing the stream is done by **ENTER**ing that protected procedure, which may in turn **ENTER** others to release resources and interlocks etc. Similarly, the user program may need to call some system domain which it has not previously needed to use at all. An example of this might be a call of the store manager to ensure that a segment containing some permanent data structure is up to date on disc.

Suppose that the attention routines timer expires during some such system call. It would be legitimate to terminate the calling domain, since the work is being done on its behalf, and the programmer cannot expect to do an indefinite amount of work. On the other hand, it is not legitimate to terminate the newly entered domain, since its programmer might have been working on the reasonable assumption that he would always have a warning of termination of at least time **t**.

It is, therefore, necessary to arrange that newly entered domains are warned of the attention and are timed. With the current CAP microprogram, this is almost impossible, since the operating system cannot easily trap the execution of an **ENTER** order. It would not be difficult, however, to arrange system tracing of **ENTER**s in a similar way to the existing tracing of **RETURN**s. If this were done, the execution of an **ENTER** could be made to notify the new domain of the attention immediately on entry, and set up a timer on it in the normal manner.

At this point it is worth noting a property of the CAP protection scheme which had a considerable influence on this work, and which different protection structures may be able to improve upon. From the point of view of the protection system, each call of a protection domain is a separate event. It is not possible for the microprogram or the operating system to discover whether or not the calls made on a domain really are independent, or whether the domain is in fact retaining internal state information. This implies that each call of another domain must be treated independently for the purposes of attention handling, and must be given the full time limit in attention state.

The original proposal for attention handling suggested that each domain would be timed separately; each **ENTER** is seen as creating a new domain independent of all others. This scheme does eventually force return from the domain which received an attention. This is because the total time which can be spent in attention state is bounded by the finite size of the C-stack. For practical purposes, however, this algorithm is unsatisfactory, because the upper bound of the time is large. The time which may be spent in attention state is of the order of $\mathbf{tn^k}$, where t is the tidy up time as before, **n** is the number of **ENTER**s which can be done in time **t**, and **k** is the number of C-stack frames available (i.e. the depth to which further domain calls may be nested). Since **k** is likely to be about 8, and **n** could not reasonably be made less than 10, it is clear that it is easy to write a program which will remain in attention state for longer than the life of the hardware.

It seems that the principle of independence of the protection domains has been taken too far. The essential cause of the problem is that the time spent in new domains is not recorded against the caller's time allocation. Provided that the new domain is itself immune from premature termination, there seems to be no reason not to count the time spent in it as time spent in the caller as well. The algorithms necessary to implement this policy are straightforward, and it has reasonably sensible practical effects. Whenever a domain was notified of an attention, the standard time interval **t** would be guaranteed to be available to it. That time may be used as the programmer sees fit; part of it may be spent in other procedures such as the store manager. But if such a procedure is called very close to the expiry time of the caller, this should not adversely affect the new domain. The new domain will be allowed to run to completion (or until its time limit expires), but the caller will be terminated as soon as the called domain returns to it.

With this policy, the maximum time that may be spent in attention state is of the order of **2kt** [10]. Again, it is bounded only because the C-stack is finite, but the bound is now a much more reasonable one. It is believed that such a policy would be satisfactory in practice, and that one could afford to make the interval **t** reasonably large without causing undue delays to the user. It would also be fairly easy to implement (provided that a suitable clock could be made available). Some microprogram changes are needed to enable system intervention on **ENTER** in a similar way to that already provided for **RETURN**. It is also necessary to allocate extra space in the C-stack frame to save timing information.

5.3.6 Evaluation

The attention handling scheme described has been in use on CAP for some time, and has proved to be satisfactory. Although the lack of implementation of time limits in attention state means that it is possible for a user to write a program which cannot be broken out of, this almost never happens.

----

[10] This is a worst case figure. It assumes that the attention is signalled when the current procedure is at depth **k** on the C-stack. Each procedure can spend a time just less than t before returning, giving a time of about **kt**. It is then possible to do **k** further **ENTER**s, and spend a further time **t** in each newly called domain, giving a further time of **kt**.

One deficiency has come to light: this is a problem in the language libraries rather than the basic mechanism. Programmers would frequently like to poll for attentions which they can clear, but do not object to being terminated without warning on attentions which they cannot clear. This would be easy if the language libraries allowed an attention handling routine to return and let the main program resume from where it left off, but they do not allow this. The easiest way to solve this would be for the language libraries to attempt to clear the attention on the user's behalf. If the attempt succeeded, a pollable flag would be set; if the attempt failed, the attention routine would be called.

6. The basis of the proposed solution

6.1 Background

Chapter 3 presented a model in which it is assumed that a computation consists of a number of protection domains, which may contain capabilities for other protection domains. The computation achieves its effect by performing actions upon objects in its environment. As it does so, it goes through many different states, some of which may be erroneous. What is required is a method of detecting and correcting an erroneous state.

It must be made clear that the classification of a state as erroneous depends on the context in which that state is to be interpreted. If some component of a subsystem fails, then that computation may enter a state in which it cannot continue to operate correctly unless some exceptional condition handling is invoked. Yet from the point of view of the operating system, the state of the program may appear to be quite normal. An operating system can detect violation of the constraints it puts upon the programs which run under it - it cannot detect violations of the constraints a program puts upon itself. It can, however, provide sufficient mechanisms to ensure that no protection domain is denied the opportunity to maintain its own integrity because of the actions or inactions of some other domain.

6.2 Simple exceptions

It is fairly easy for a domain to protect itself from the actions of the domains which it calls, provided that the underlying operating system allows it to handle any exception which may be raised by the domain called. For example, a domain may encounter some abnormal condition when it calls another domain to claim an interlock.

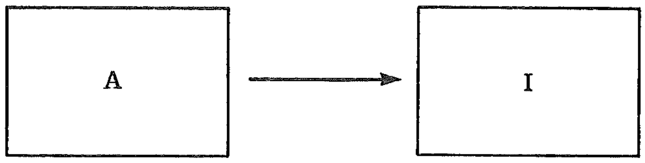

Figure 5: A simple domain call

Figure 5 shows this simple case. Domain **A** calls domain **I**, which is assumed to be some domain responsible for managing interlocks. If I fails, then provided that this failure is passed back to **A** in a clean manner, **A** can keep its internal state consistent by undoing any actions which were done on the assumption that the interlock would be made available. Although **A** might have been temporarily in an inconsistent state when it called **I**, the operating system was prepared to guarantee that control would return to it in due course.

6.3 The 'unfinished business' problem

The case of inaction of another domain is much less straightforward. To continue the above example, suppose that the interlock has been successfully claimed, and the fact of the claim has been noted in A's private data structures. A may at this point need to return to its caller, C, with the expectation of being called again later in order that the interlock may be released.

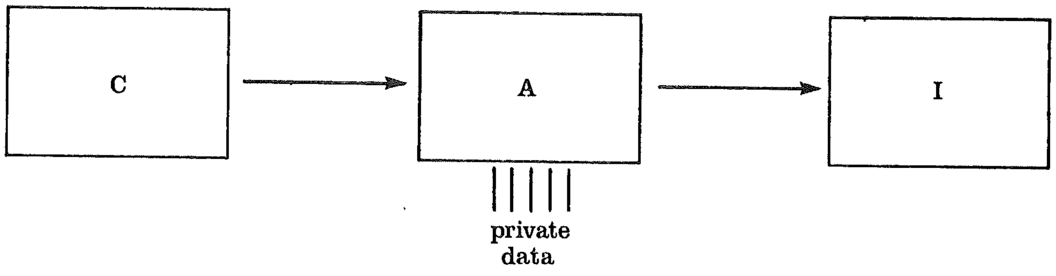

Figure 6: Three interacting domains

Figure 6 illustrates this case. Suppose, however, that the domain C does not, in fact, call A again before it terminates and returns to its own caller [1]. So long as a capability for the domain A exists as part of the computation, the operating system cannot detect that the call will not be made and it cannot reasonably intervene. As soon as the capability is overwritten, however, it is known that no further call can be made upon it,

---

[1] The reason for its failure to do so is irrelevant. A likely cause is that it has stopped as a result of some console 'quit' signal, as discussed in section 2.4. It could equally well be a simple programming error.

and what was previously a valid state becomes an erroneous one [2]. The domain A may at this point need to be warned that it is about to be abolished, so that it can correct any erroneous state which would affect other domains.

The purpose of doing this is to ensure that the remainder of the computation can continue in a self-consistent state. If the operating system did not call the domain with the interlock, but merely forced the interlock to be released, the requirements of the operating system to maintain its own integrity might be fully met, but the computation running under it might find that it could not continue to function. It is inadequate to rely on the domains remaining in the computation detecting the inconsistency. Until the final capability for the domain with the interlock is overwritten there is nothing wrong. Even if the other domains could detect the overwriting of capabilities, there is still, in general, nothing they can do to correct the state of the computation. The information necessary to do this was locked within the domain which has been lost; only it could know what was needed to restore a valid state. It must be emphasized that the operating system cannot be held responsible for ensuring that the correct actions are taken even if the module is warned that it is about to be deleted, any more than it can ensure the correctness of programs in general. It is sufficient to allow the module an opportunity to restore its environment in the way that its author sees fit. If this is done, it will be possible for correctly programmed components of subsystems to behave correctly even if other components do not.

There are two interrelated aspects to the handling of domains which are to be deleted: detecting that a domain is in fact inaccessible, and then arranging that it can be disposed of in a controlled manner. The situation is complicated by the fact that a domain which can no longer be called from

----

[2] The erroneous state is in this case to be interpreted in the context of the user computation rather than the operating system (see section 6.1). The fact that the operating system cannot detect that the state is erroneous is a problem to be considered in chapter 7.

the main body of the computation is not necessarily isolated from that computation (indeed, if it was, there would be no point in letting it tidy up its state, since such action would have no effect on the computation as a whole). An inaccessible domain will generally possess capabilities for other domains, which may or may not be accessible from elsewhere. It may also possess capabilities for other shared resources, such as segments of store. If the aid provided by an operating system to enable domains to maintain self-consistent states is to be useful, the system designer must be prepared to define how the overwriting of capabilities is to be handled, and what each protection domain may assume about its environment.

6.4 A provisional model of ideal behaviour

It is useful to consider how the handling of inaccessible domains might be expected to happen in an ideal world, and then consider whether this behaviour can be implemented in a real system. Later it will be demonstrated that there are circumstances in which this behaviour has the wrong practical effect, and some modified proposals will be made.

The computation is modelled as a directed graph; the nodes are protection domains, and the arcs are the capabilities one domain has for another. Whether the domains are separate processes or different protection environments within a single process is not of immediate concern. As the computation proceeds, control passes between domains by some mechanism such as a procedure call or a message passing system. It is assumed that domains can and do retain internal state information between calls - this state may be entirely private to the domain or it may be partially shared with others. New domains may be created and passed as arguments from one domain to another, and references to domains may be overwritten. Thus the graph of links between domains may be constantly changing.

Sometimes, as a consequence of the destruction of an arc of the graph, one or more nodes may become inaccessible from the remainder. The nodes which have become detached from the remainder correspond to protection domains which can never be called again. At this point, an exception should be reported to those domains, in order that they may perform any final tidying up operations before the representation of the domain is finally dismantled and the operating system resources they occupy are recovered. The term '**finish** call' or '**finish** exception' will be used to denote this call. The reporting of the **finish** exception should ideally be done as soon as it is logically possible to do so. This implies that the **finish** call should happen as a side effect of overwriting the last capability for the domain concerned. There are a number of good reasons for this.

First, if the deleted domain does have any tidying up work to do, this will usually consist of the release of interlocks and other resources. As a matter of good general programming practice, such actions should be done as early as possible, in order that the remainder of the computation may continue unimpeded. Furthermore, the operating system resources occupied by the domain cannot be released until it has finished its work, and it is usually advantageous to get this done as soon as possible.

Secondly, reporting the exception as early as possible has the advantage that the order in which events occur can be made well defined. Overwriting a single capability might make a number of domains inaccessible at the same instant, and it is not immediately obvious how the operating system should decide on the order in which to deal with them. Suppose, however, that the decision is made to report the exception immediately to the domain whose capability was actually overwritten. As it processes the exception, it can be regarded as a separate computation, with its own graph of domain references. If, as it computes, capabilities are overwritten which make other domains inaccessible from both it and the main computation, the same algorithm may be applied recursively to report the exception to them. Any capabilities which remain in the domain's address space after it has deemed its work to be complete will be deleted by the protection system as part of the work of deleting the domain. As this is done, further domains may become inaccessible, and these will be dealt with in precisely the same way. Finally, control will return to the main computation. Apart from possible real time effects and side effects of the exception handling, the main computation has simply deleted a capability, which it sees as an indivisible action.

As far as the programmer is concerned, there are only two differences between programming in this environment and in a more conventional one in which deletion of capabilities is not treated specially. The first is that the operation of deleting a capability for another protection domain is liable to cause an implicit call of that domain. The second is that the programmer can write on the assumption the domain will not be deleted until the fact of its inaccessibility has been reported to it. It is therefore safe for the domain to retain internal state information about resources, interlocks etc. between calls, because one can guarantee, subject to the assumption that the entire operating system does not fail, that the domain will get an opportunity to tidy up its environment before it is deleted.

The method by which the protection system makes the termination call to a domain will be system dependent. At its simplest, it need consist only of an ordinary call with some conventional parameter indicating the reason for the call. Such a parameter is frequently present anyway, to distinguish which of several services provided by a domain is required. Such a scheme would have the property that a **finish** call would not absolutely guarantee that the domain would never be called again, since the ability to make a **finish** call would not be protected. This would merely require a little extra care in programming to ensure that the domain was not confused by such a call.

It would probably be better, however, to report the event to the domain in a manner different from that used for an ordinary call, by raising an exception. Any domain which had necessary work to do before its termination would supply a handler for the exception, and it would be this code which would be called before the domain's deletion. If no exception handler were provided, the protection system could assume that the domain could be deleted immediately, without calling it again. If the system being used already provides comprehensive exception handling techniques, then programmers concerned about integrity of their protection domains are likely to find that providing an exception handler is the natural way of dealing with domain deletion. This technique is also applicable to high level languages which provide exception handling facilities.

Despite its conceptual simplicity, the technique described above is plagued with practical problems. The main difficulty is liable to be arranging that the inaccessibility of a protection domain can be detected as soon as it happens. In a capability based system, it is usually possible to copy capabilities, move them around freely, and pass them as arguments between domains. It is unusual for such a protection system to perform any management activity when a capability is copied or moved; the capability is simply a protected bit pattern. Although the entire graph of domain references is in principle available for inspection, it would require a significant computation to discover that the deletion of a particular arc had isolated one or more nodes. This is a standard problem, common to all systems which can be modelled as directed graphs.

6.5 Domain management

If the situations described above are to be handled correctly by the protection system, it will be necessary to have some overall system management of capabilities. Ideally, one would prefer operations on capabilities to involve no more high level work than loading and storing numbers in main memory. This ideal is, however, unrealistic, even if one ignores the problems discussed in this dissertation. Capabilities usually represent some real resource, such as a segment of memory or space on disc, which it is the job of the operating system to manage. This implies that the operating system must be prepared to keep track of capabilities in order that it can administer real resources. It is therefore not unreasonable to accept a modest amount of extra management activity to detect unreferenced protection domains.

A characteristic of many capability based architectures is that the capabilities do not refer directly to the objects they represent, but instead indirect through some other data structure, such as an object map. If this is the case, then the problem of detecting loss of access to a domain is somewhat simplified, since it is much easier to deal with a single, centralized data structure than one which is distributed. A central table provides a convenient means of remembering information which refers to all instances of a capability, without restricting the freedom to copy capabilities.

6.5.1 Use counts

Assuming the existence of such a table, one could, for example, consider keeping a use count for all capabilities. Each entry in the central table would simply record the number of references to it, as in figure 7:

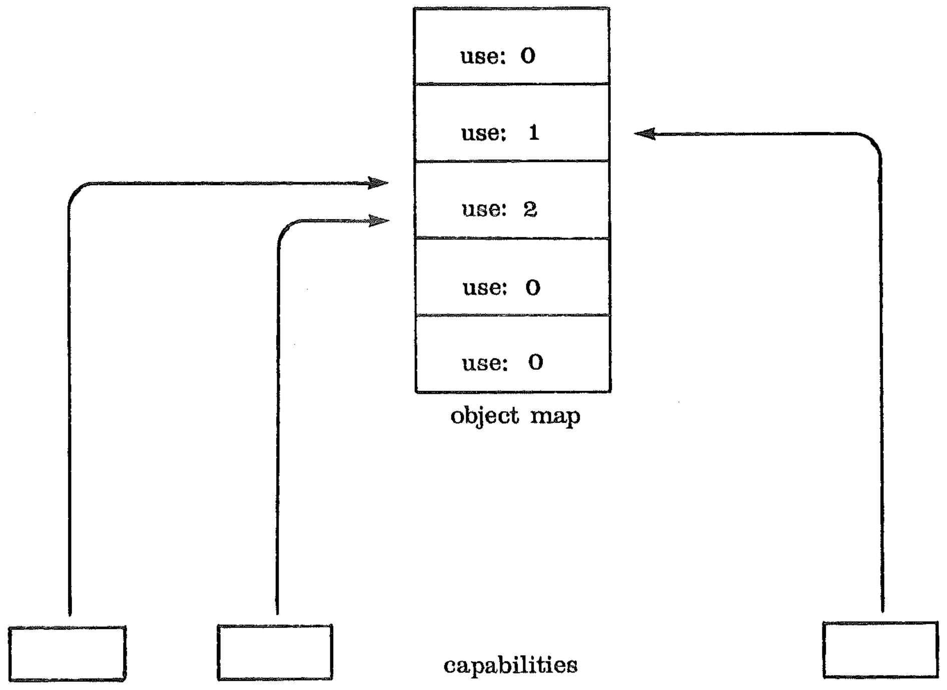

Figure 7: An object map structure with use counts

Maintaining these use counts would be a fairly simple operation. Copying a capability would cause its use count to be incremented, and deleting a capability would decrement its use count. Such a scheme is easy to implement in, say, a microprogrammed protection kernel. Whenever a use count was decremented to the value zero, indicating that the last capability for an object had been overwritten, it would be possible to signal some low level exception to the protection system to inform it that the object represented by the object map entry concerned could no longer be referenced.

For objects such as segments of memory or backing store, the operating system would use this information simply as an indication that the relevant resource could now be released for reuse. If the lost capability is for an entire protection domain, the situation is more interesting. The first point to note is that a protection domain is a capability containing object, and that action must be taken to ensure that the use counts for the capabilities it contains are decremented. This may be a job for either the low level protection system or the higher level software. If the protection system is not concerned with enabling domains to maintain the integrity of their private data structures, then its actions will be confined to this.

The detection of loss of access to a domain does, however, give the protection system the opportunity to refrain from deleting the contents of the domain until it has been given a chance to tidy its internal state. It can do this simply by calling the domain, raising some exception to indicate that the call is a final warning that the domain is about to be deleted.

This simple description glosses over one important point. If the last capability for an object has been deleted, how can it possibly be called again? By definition, no program has the capability to do so. This problem is by no means insurmountable. Since the loss of access is detected at a low level by the protection system (before any information about the domain has been discarded) it should be easy to arrange that some suitable capability can be constructed and handed over to the part of the protection system responsible for dealing with inaccessible domains. Arrangements must be made to ensure that this capability can be deleted without provoking the recovery mechanism into action a second time. The means by which this is achieved will depend on the particular protection system concerned. It is sufficient to note here that some mechanism must be provided, and that there is no logical difficulty in doing so.

Care must also be taken to consider the possibility that the tidying up operations within the domain may themselves overwrite capabilities and can therefore cause more use counts to fall to zero. Whether recursive invocation of the mechanism can be tolerated will depend on the particular implementation, but at the very least the system must be prepared to handle a queue of objects awaiting deletion.

6.5.2 Limitations of use counts

The limitations of use count techniques when dealing with directed graph structures are well known: they do not detect inaccessible cyclic structures. If capabilities can be passed freely between domains as parameters, it is clear that it is possible to create cyclic structures of protection domains. All that is required is for two domains to be passed capabilities for each other.

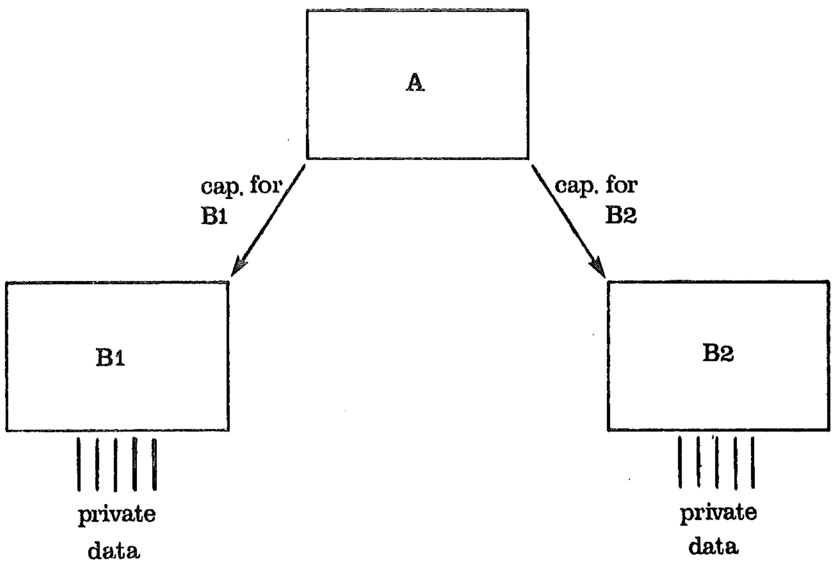

Figure 8: Creation of a detached cyclic structure (1)

Figure 8 denotes a domain **A** with capabilities for two other domains **B1** and B2. If A passes B1 its capability for B2, and B2 its capability for B1, then the structure shown in figure 9 is obtained. If **A** then deletes its capabilities for both domains, then the detached cyclic structure shown in figure 10 will result.

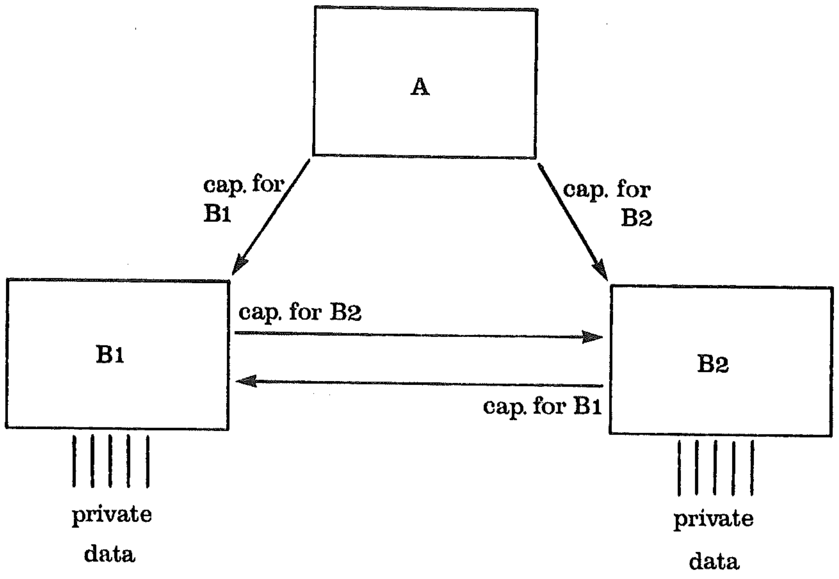

Figure 9: Creation of a detached cyclic structure (2)

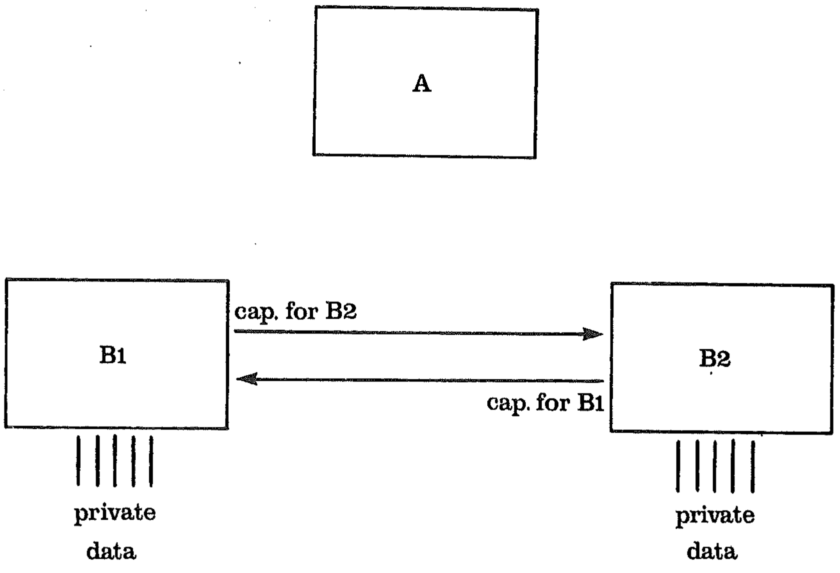

Figure 10: Creation of a detached cyclic structure (3)

Note that in the normal course of affairs B1 and B2 can never actually call each other, since there is no way that control can reach either of them. Either or both of them may, however, be expecting to be called again.

One way round this problem is to restrict the movement of capabilities for protection domains in such a way as to prevent cyclic structures from being created. For example, if capabilities can be moved but not copied, the graph of possible protection domain calls degenerates into a tree, and cyclic structures cannot occur [3]. In such a scheme, the use count of the capability for a protection domain could never rise above one. Such a restriction would, however, considerably reduce the power of the capability mechanism. It would, for example, disallow the following useful structure:

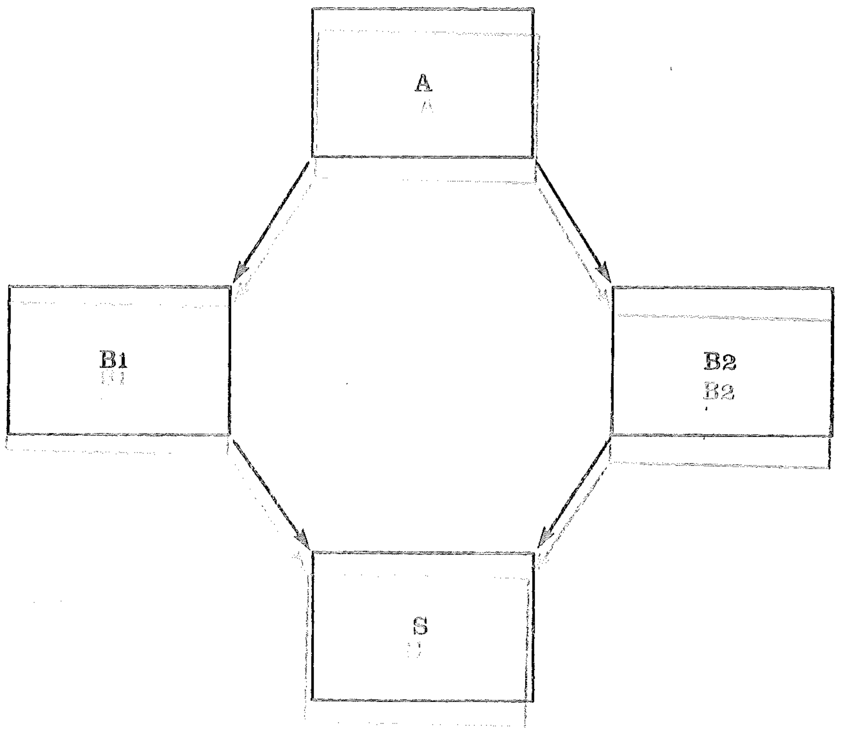

Figure 11: A non-hierarchic domain structure

---

[3] This assumes that the only way of passing domain capabilities between domains is by passing them as parameters of a domain call.

S is a domain which manages some abstraction. It is shared by domains B1 and B2, each of which can be called by A, which is assumed to be some controlling program. The ability to set up general non-hierarchical structures of the type shown in figure 11 is a valuable property of capability based protection mechanisms, and it would be unfortunate to lose it.

6.5.3 Garbage collection

In circumstances where use counts fail, garbage collection techniques can be applied. Provided that some root node for the graph of domain references can be identified, the graph of references can be scanned (presumably by a highly privileged program). The entries in the object map can thus be divided into those that can be accessed and those that can not. The objects that can never be referenced again can thus be made subject to a similar kind of cleanup operation to that suggested in the previous section.

6.5.3.1 The order of **finish** calls

There is though, an important difference between the detection of unreferenced domains by use counts and garbage collection. In the latter case, when the unreferenced domains have been isolated, there is no easy way for the protection system to determine the best order to report inaccessibility to the various domains. Going back to figure 6 on page 82, suppose that the domains A and I have both been isolated from the remainder of the computation. Which should be informed of the fact first?

When use counts were used, it was natural for A to be called first, and it was assumed that A made use of I in its tidying up operations (to release an interlock). If a garbage collector makes a random choice, and I happens to be called first, it may decide to release the interlock unilaterally, on the grounds that A has failed to do so. Having done this, I is in a consistent state, and is prepared to be deleted. All is not well, however, for the domain A. Its private data structures may no longer be consistent, since its interlock has been taken away from it but it has not been able to record the fact. Presumably it will discover that something is wrong when it is called to tidy up and attempts to release the interlock. By this time, however, damage may already have been done: the forcible release of the interlock may have allowed another instance of **A** in a different process to proceed. This second instance may then erroneously assume that **A** has been left in a consistent state.

6.5.3.2 An example

At this point, it is worth giving a concrete example of this problem. Suppose that the domain **A** in figure 6 (p. 82) is a protection domain which provides an interface to a visual display unit. One of its jobs is to present a 'virtual terminal' interface for full screen working; domain **C** could, for example, be a screen editor. In order to use the terminal for this activity, it has to be set up in a special way, and **A** takes the responsibility for doing this. To prevent more than one simultaneous use of the terminal in this manner, A makes use of interlocks, provided by **I**. **A** may need to perform various actions <u>before</u> releasing the interlock, such as setting the terminal back to a standard state with the cursor in a sensible place. Under normal circumstances, the editor, **C**, will make a final call to **A** just before it terminates. **A** will do its final work, release the interlock, and all is well. In a well designed system, the editor should be <u>able</u> to do this even if it runs out of allocated CPU time, or the user breaks out of it, or some other disastrous error occurs. But one cannot compel the author of **C** to handle such things properly, and unless one is prepared to make the (administrative) rule that only certified programs may be given a capability for A, then it is clear that **A** must look after itself.

The use count scheme solves this example quite well. As soon as the capability for **A** is overwritten, **A** is called to tidy up. It resets the terminal to the standard state, releases the interlock and exits. When the capability for I is overwritten, I is called, and finds nothing wrong - at least as far as A is concerned. On the other hand, using a garbage collector may lead to the interlock manager being invited to tidy up before the terminal handler. In this case the terminal interlock will be released before the terminal handler has had a chance to restore the standard state. In particular, some other program may start using the terminal before it has been properly reset, possibly causing incorrect output to be produced.

This example has deliberately been chosen as one in which the consequences of failure of the mechanism are not disastrous, but merely annoying. Techniques for maintaining the integrity of abstractions managed by programs are known, and are widely used in database systems. Such techniques can be justified when the consequences of failure would be serious, or if the abstraction is required to survive such things as hardware failures of the equipment on which the system runs. But for relatively simple jobs such as the above example, the expense of using database techniques can hardly be justified; the cost of such mechanisms is out of proportion to the effect achieved. What is required is a much cheaper way of solving a restricted class of problems.

6.5.3.3 When to garbage collect

There is a further problem with the garbage collection approach. This is the difficulty of knowing when to do it. It was stated earlier that there should be an attempt to detect the loss of access to a domain as early as possible. This implies that a garbage collection should be done whenever a capability is overwritten, so that the inaccessibility is detected as a side effect of its immediate cause. This would almost certainly be rejected on efficiency grounds; garbage collectors are usually expensive in terms of CPU time, and their cost depends more on the amount of useful material they scan than on the amount of garbage they succeed in collecting. If garbage collection were done as a side effect of every deletion, many of the collection attempts would be futile, and it is unlikely that such an overhead would be tolerated.

In systems which rely on garbage collection, collections are usually done only when necessary. In the model considered earlier, the object map would probably be garbage collected when it became full or nearly full. If garbage collection were relied on to detect unreferenced protection domains, there is a danger that the delay before the unreferenced domain is notified will be intolerably long. In particular, in the example of the terminal handler, the process could easily deadlock if it needs to claim the interlock but is unable to do so because both **A** and I are waiting to be informed of their inaccessibility before they can release it.

Other possibilities spring to mind. The garbage collection could be done on a regular basis regardless of logical necessity. A continuously running asynchronous garbage collector could be used. Both of these would have the effect that the inaccessible domains would be discovered eventually, and one could even put an upper bound on the time which could elapse before such discovery. Nevertheless, this solution remains somewhat unsatisfactory. The main objection is the scope for non-deterministic effects which depend on the exact timing of the garbage collection. The domains being disposed of may share data structures with domains which are still part of the main computation, and the side effects of the garbage collection may therefore be detectable. One would prefer that from the point of view of the main computation, the effects of the tidying up operations were deterministic.

6.5.4 Use counts and garbage collection

One possibility which is always open when choosing between use count techniques and garbage collection techniques is to have a combination of both. Use counts detect most of the common cases of inaccessibility, and garbage collection need only be used to detect the unreferenced cyclic structures.

This would give the right effect for the terminal handler example, and probably for most others, since cyclic structures are unlikely to be very common. Moreover, it is known that anything discovered by garbage collector is a detached cyclic structure, and the problem of knowing which domain to call first is less acute. Referring back to figure 9 on page 92, the relationship between B1 and B2 is symmetrical, and there is no way that a system which knows nothing of their internal workings can decide which should be called first. It is a reasonable specification in this case, to assert that B1 and B2 will be notified in an undefined order; the two domains are expected to be able to cope with being called in either order.

On the other hand, in the structure shown in figure 12 the three domains are not symmetrical; a human observer would probably assert that C should be notified last.

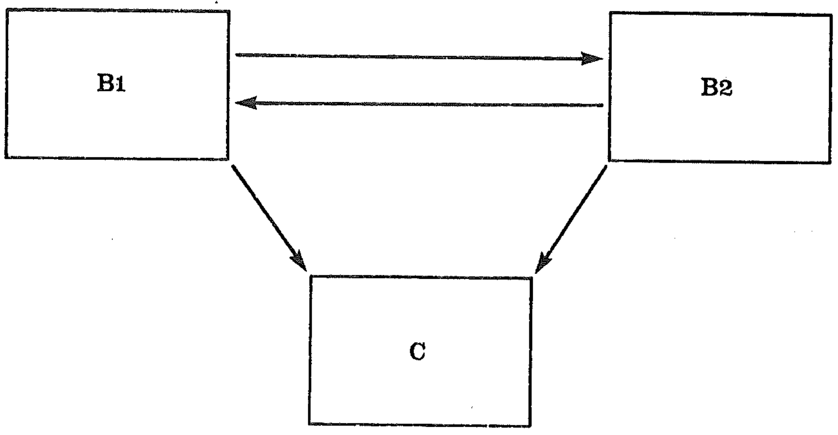

Figure 12: A more complex cyclic structure

A further disadvantage of combining use counts and garbage collection is that that it makes the problems associated with sole use of garbage collection worse. A garbage collector called when essential will be called less often (since much of the unreferenced material will be detected by the use count mechanism), and a garbage collector called at other times will be seen as more of an overhead (since it will only rarely do anything useful).

6.6 Summary of the basic solution

This chapter has presented what might be termed a 'basic' solution to the problems raised in chapter 3. The solution appears at first sight to be adequate, but difficult to implement in a sensible and efficient way. The next chapter considers the limitations of the basic solution, and goes on to suggest refinements.

7. Refinements of the basic solution

7.1 A different approach

Up to now, it has been tacitly assumed that the facilities for allowing domains to tidy up after they have been abandoned are to be added to an existing protection structure as an afterthought. It is for this reason that the gross condition of complete loss of access to a domain was stipulated as being the sole condition for informing the domain that it should tidy up. A protection system usually has no information about the future behaviour of a computation which would enable it to choose any better condition. The aim of this section is to explore a different approach, which necessarily requires a slightly different view of the way programs interact.

7.1.1 An example in which the idealized model fails

In order to show that implementing the strategy given in section 6.4 is not the whole story, an example will be given of a circumstance in which it has the wrong effect. An example was given in section 6.5.3.2 of an applications program, C, making use of the services of A, a terminal interface. This in turn made use of the services of I, an interlock manager (fig. 13). Both A and I need to maintain private data structures, and both therefore need to be informed of their impending deletion.

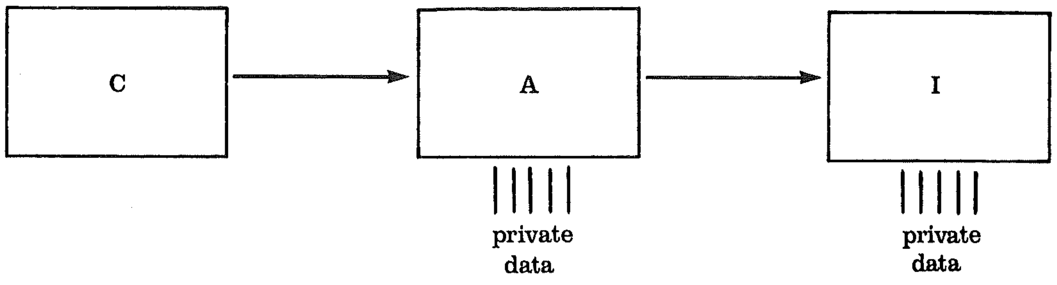

Figure 13: Example of interacting domains

Now assume that the domain A contains a bug: its termination routine does not call I to release the interlock, simply because the programmer has forgotten about it. Naturally one cannot hope that the interface provided by A will work properly under such circumstances, but the fact that I can get an opportunity to tidy up its own data structures (the interlocks) would suggest that it ought to be possible to avoid total deadlock.

When C has finished with the services of A, it makes its final **finish** call to A. A does whatever is necessary to the terminal, tidies its internal data structures, neglects to call I and returns to C. At this moment the state of the computation becomes erroneous. A believes its work to be complete, and has no intention of calling I again, but it still holds a capability (by assumption, the only capability) for I, whose work is not complete.

If C deletes its capability for A, the domain A will be notified that it is about to be deleted [1]. This call will probably have no effect; there is no reason to suppose that the 'tidy up' code in A will be any more correct than the main line code. Eventually the representation of A will be destroyed by the operating system, and I will become unreferenced. It in turn will be given an opportunity to tidy up, and will find work to do - i.e. release the interlock which was claimed by A. The erroneous state has thus been corrected.

There are two things wrong with this way of doing the tidying. First, although the correct thing was done, it was not done at the right time. The inconsistent state which came into existence when A returned to C could not be detected and corrected until (at the earliest) C had overwritten the capability for A. Experience has shown [Cook 78] that programmers are often lazy about overwriting capabilities they no longer need. It may even be the case that A is a serially reusable program, and C is keeping the capability in case it needs it again. In either case, the overwriting of the capability for A is likely to be considerably delayed.

Secondly, although the right thing is eventually done, there is no way that the fact that A has acted incorrectly can be reported to A itself, since its representation has already been destroyed at the time the error is discovered. It would be preferable if the action of A which created the erroneous state (i.e. its return to C) could be made to raise some exception within A as soon as it happens. This would make it far more likely that the programmer responsible for A would be made aware of the existence of the bug in his program.

----

[1] Optimizations to remove this apparently futile call will be considered later; they do not affect the logic of the argument.

7.1.2 Changes proposed

If this is to be done better, it seems clear that the protection system will need to have more information about the internal state of protection domains. The difficulty raised in the previous section came about because the protection system was unaware that:

1. Domain I's work was incomplete

and 2. Domain A (the only one capable of calling I) has already done its cleanup work (incorrectly).

It would appear from this that it would be useful if the protection system could be made aware of these facts. Unfortunately there is little a protection system can do to determine them from the behaviour of the programs: they are maintaining abstractions which the system knows nothing about. One must, therefore, be prepared to rely on information provided by domains about their own state. There is no way to ensure that such information is accurate; but it is sufficient to ensure that a domain which provides incorrect information about its state can harm only itself. If, in addition, the provision of this information enables the computation as a whole to proceed more smoothly, or if it enables programming errors to be detected earlier, then it is probably worth while.

The fundamental piece of information that the protection system requires from a domain is an assertion about whether the internal data structures of the domain are deemed to be in a consistent state. Asserting that they are consistent is equivalent to asserting that the domain need not be called again to tidy up (this does not, of course, imply that the domain must not be called again). The converse assertion is equivalent to saying that the domain must be called again before its representation is destroyed [2]. The obvious way that the domain can indicate its state is by passing some

---

[2] One can consider making more detailed assertions, such as how much CPU time is required for the cleanup operation. For the time being, only a simple binary value will be considered.

parameter to whatever operation it uses to relinquish control, though some protection systems may require a special system call to be made just before the return. The operation of noting the state can be made very simple, since the protection system does not need to do anything with the information except note it for future reference. A domain which relinquishes control with an indication that it needs to be entered again must provide some code to deal with a call made to enable it to tidy up; this may be formally regarded as setting up a handler for an exception which can be raised later by the protection system.

Note that whether a domain needs to be called again is a property of <u>each</u> return; it is not a fixed global property of the domain. One can imagine a program which sometimes returns with tidying up work to do and sometimes does not. The terminal handler described in section 6.5.3.2 is an example of such a program. It can be written as a serially reusable program; whether or not it must be called again before destruction depends on what it has done to the terminal and whether it holds an interlock.

In the paragraphs which follow, a domain which returns indicating that it does not need to be called again will be said to have returned **finished**, and a domain which does need to be called again will be said to have returned **unfinished**. A domain which has never been called is treated as having returned **finished**. A final call which is made to a domain to enable it to tidy up will be called the **finish** call.

The first advantage of recording information about whether a domain needs to be called again is a practical one. Many programs will have no requirement to be informed of their impending deletion, and for such programs it would be preferable to avoid the overhead of making an extra call before they are deleted. If the protection system records whether or not a domain needs to be called again, it becomes easy to implement this optimization.

The main reason for the change is that it makes it possible for the protection system to detect erroneous states earlier. Returning to the example, if I returns **unfinished** to A, it can be asserted that it would be inconsistent for A to return **finished** to C. It must either call I again to release the interlock (so that I returns **finished**), or it must return **unfinished** to C. If it does attempt to return **finished**, then the protection system will raise some exception within A. At first sight this would appear to deal with the example much better than the original proposal did. It does, though, pose a number of questions, which the following sections attempt to answer.

- What does a caller do if a domain always returns **unfinished** and refuses to do otherwise? (see section 7.1.3)
- What happens if a domain repeatedly attempts to return **finished** when it holds capabilities which have returned **unfinished**? (see section 7.1.4)
- How exactly does the protection system detect that an inconsistent state has arisen? The example given was a particularly easy case; what happens when there is more than one capability for a domain? (see section 7.1.5)
- How is overwriting of capabilities dealt with in this scheme? (see section 7.1.6)
- Can capabilities for domains which have returned **unfinished** be passed between domains as parameters? (see section 7.1.7)
- What limitations are imposed on what a program can do if called to tidy up? (see section 7.1.8)

7.1.3 Domains which refuse to return finished

If a domain refuses to return **finished**, then the caller can be inconvenienced, since it in turn will not be allowed to return **finished**. There are two ways of coping with this. The first is for the caller to overwrite the capability for the domain, and thus throw the burden of dealing with the problem onto the protection system. A second solution is for the protection system to provide a mechanism which allows any domain to call another domain for which it has a capability in the particular way which indicates a **finish** call. A domain would not be allowed to return **unfinished** from such a call; if it attempted to do so, the protection system could either raise an exception within the domain or override the attempt to return **unfinished** and return **finished** instead. In either case, the action is conceptually the same; the domain is given a final chance to tidy up.

7.1.4 Domains which invalidly return finished

If a domain attempts to return **finished** when it holds capabilities for domains which have returned **unfinished**, the protection system can raise an exception within the domain. The handler for that exception may be programmed in such a manner that it (erroneously) deems the exception to have been handled and passes control back to the domain's main program. This may then attempt to return **finished** again, thus leading to a loop. It ought not to be necessary to prevent this loop entirely, since it is no worse than an infinite loop in an ordinary program, which it must be possible to break by imposing CPU time limits or by using attention mechanisms. Provided, therefore, that the <u>default</u> exception handlers behave in a reasonable manner, this is unlikely to be a problem.

The default exception handler would typically report the error to the programmer in whatever manner is appropriate for the language system being used. It would then need to tidy up the domain's capabilities before returning **finished**.

In order to make it easier for language implementors to write this default handler, it would be useful if the protection system provided an extra facility which is not logically necessary. Its action would be to identify any capabilities which would prevent the current domain from returning **finished**, and make the **finish** call to them. After using this facility, it would be guaranteed that the current domain could validly return **finished.** A similar effect (but with rather more drastic side effects) could be achieved by providing a facility to overwrite all capabilities in the current domain for domains which have returned **unfinished**, or even all available domain capabilities. The treatment of overwriting of capabilities (to be discussed in section 7.1.6) will then deal with any **unfinished** domain which would prevent the current domain from returning **finished.**

It would be possible for the protection system to invoke action similar to that described in the previous paragraph whenever a domain erroneously attempted to return **finished**, rather than relying on the language system to provide a default action. It is <u>not</u> proposed that this should be done. An erroneous attempt to return **finished** would almost certainly be caused by a programmer failing to understand the specifications of the domains whose services he uses. Such errors should be reported rather than being covered up by the protection system.

7.1.5 Detecting inconsistent states

In principle, the protection system can tell when an erroneous state (in the sense described in section 7.1.2) is about to arise. The following definition can be given: it is invalid for a domain D to return **finished** if there exists a domain U such that:-

1. U has returned **unfinished**

and 2. D has a capability for U

and 3. there does not exist a domain E (different from D and U) such that:-

    a. E has returned **unfinished** or is on the current call stack

    and b. E has a capability for U.

Parts 1 and 2 of this definition describe the conditions expressed loosely in section 7.1.2, and are sufficient to deal with the example given.

The third condition is required to cope with more complicated cases. Consider again the non-hierarchical structure shown in figure 11 on page 93. It may be that the domain B1 is responsible for the initialization and termination of a sequence of actions using S, and B2 makes intermediate calls to simple functions provided by S. While S is in use it will return **unfinished**, and B1 will have returned **unfinished** to A (it is forced to do so by parts 1 and 2 of the above definition). Later, A calls B2, which uses S. Assuming that B2 retains no internal state, and does not need to be called again, should it be forced to return **unfinished** to A? So long as the path A to B1 to S exists, there is no good reason for forcing B2 to return **unfinished**, and part 3 of the definition expresses this.

Imposing the rules suggested makes it somewhat easier for the protection system to decide whether it should intervene to allow a domain to tidy up. When a domain returns **unfinished**, then provided that the last call was not a **finish** call, all that is necessary is to record the fact for future reference. When a domain attempts to return **finished**, the protection system is required to check that such a return is valid. The computation required to determine this fact is not trivial, but is fairly modest. The capabilities available in the domain must be examined [3]. Capabilities for domains which have returned **finished** may be ignored, since it is impossible, under the conditions given, for the only access route to a domain which has returned **unfinished** to be via such a capability. If one or more capabilities for domains which have returned **unfinished** are found, then parts 1 and 2 of the definition are satisfied and it is necessary to test part 3. Condition 3a restricts the number of domains which need to be examined, to see whether they satisfy condition 3b. Note that when examining a domain, the algorithm need only consider the capabilities directly available; the constraints imposed ensure that a capability for an **unfinished** domain can never be 'lost' inside a **finished** domain.

---

[3] This section describes the logic of the algorithm. Optimizations should be possible in a practical implementation.

7.1.6 Overwriting of capabilities

The need to handle capabilities which are overwritten remains. It is possible to deal with them using the same techniques described in sections 6.5.1, 6.5.3 and 6.5.4, the only difference being that unreferenced domains which have returned **finished** need not be called again.

The proposed constraints allow a different algorithm to be employed, which allows intervention at the moment of deletion of the reference to the object, albeit with some overhead. Whenever a capability for a protection domain, D, is deleted, the protection system can check whether it last returned **finished** or **unfinished.** If it returned **finished,** there is no need to consider it further; whether or not it is unreferenced is of no immediate concern. If, on the other hand, D returned **unfinished,** then it is a potential candidate for being called to tidy up. The conditions under which this ought to be done are essentially the same as part 3 of the definition in section 7.1.5. The **finish** call should not be made to D if a domain E (different from D) can be found such that:-

a. E has returned **unfinished** or is on the current call stack

and b. E has a capability for D,

The search must start from each of the domains on the current call stack, and follow all the access paths from them which lead to domains which have returned **unfinished.** If no such domain can be found, then the protection system should make the **finish** call to D. Note that in this case D may in fact still be referenced from a domain which has returned **finished;** the programmer must therefore take into account the fact that the domain <u>might</u> be called again even after the **finish** call.

7.1.7 Passing domain capabilities as parameters

Since protection domains are represented by capabilities, it is possible to pass them between domains as parameters and return them as results. Capabilities for domains which have returned **finished** are not treated specially in the proposed scheme, so there is no difficulty about these. The only case which might cause difficulty is that a domain may need to return **finished**, passing as result a capability for a domain which has returned **unfinished**.

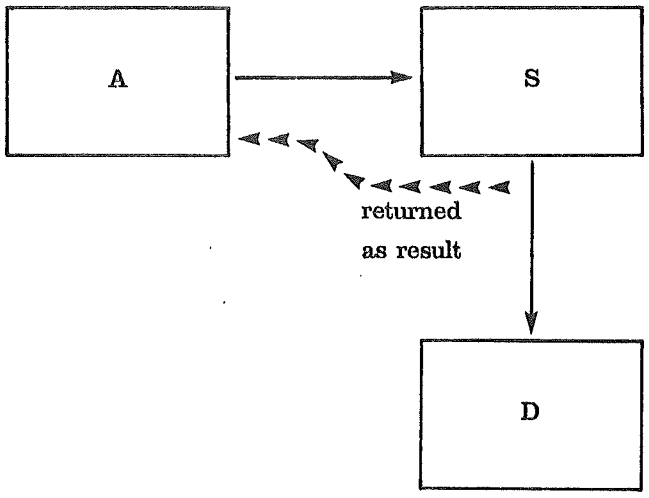

Figure 14: Returning a domain capability as a result

An example of this is shown in figure 14. Domain **A** requires a capability for domain **D**. It may be that it can only obtain this via some intermediate domain **S**, which obtains the required capability and calls D to perform some initialization. Assume that D returns **unfinished** from this initialization call. S then returns to A, passing over the capability for **D**. In order to indicate that A is being given full responsibility for **D**, S will return **finished**.

Care needs to be taken to ensure that this is possible. When considering whether the **finished** return is permissible, any capabilities being returned as results must be added to the set of capabilities accessible to A. If this is done, the return will be valid, because part 3 of the condition in section 7.1.5 will not be satisfied.

7.1.8 Limitations placed on the finish routine

The main constraint placed on the **finish** call is that it must return **finished.** If it attempts to do otherwise, the protection system may take drastic action. Whatever that action is, the net effect will be that the domain will be deemed to have returned **finished** regardless of what it actually did - the domain has had its opportunity to tidy up, and can thus be abandoned. Care must be taken, though, to take account of any capabilities it may have for other domains which have returned **unfinished.** If any domain capabilities are found which satisfy all three parts of the condition in section 7.1.5, then the protection system must make a **finish** call to those domains in turn.

It is also necessary to consider what resources are to be made available to complete the **finish** call. It must be possible to prevent a **finish** routine from taking an indefinite amount of time. One solution to this is to impose a time limit on <u>all</u> **finish** calls. This time limit would need to be documented so that the programmer can know how much work he can expect to be able to do when tidying up. Another possibility is not to impose limits on a **finish** call as such, but to rely on attention handling mechanisms. The programmer would normally have an indefinite amount of time to tidy up, unless the user decided to terminate the computation, in which case a time limit would be imposed.

Whichever method is chosen, each domain must be permitted a reasonable time in which to tidy up; no single domain should be able to take all of the available resources. The algorithms necessary to do this, whilst preserving the ability to stop a computation within a modest time interval, are discussed in section 5.3.5.

A further discussion may be found in [Taylor 78]. In Taylor's scheme (see section 4.3), time is regarded as a resource for which a domain may have a capability. It would probably be useful in this case for domains to make more detailed assertions about their state when they return. In particular, they could specify how much time they require to perform their tidy up operations.

7.1.9 Degenerate cases

In order to demonstrate the effects of discriminating between returning **finished** and **unfinished**, it is instructive to consider the degenerate cases of the algorithm proposed.

If domains invariably return **finished**, then the algorithm degenerates into the original behaviour of the system in which nothing at all was done to enable domains to tidy up. It can never be invalid for a domain to return **finished**, because it cannot possess a capability for an **unfinished** domain, and therefore part 1 of the condition for an invalid return can never be satisfied. Overwriting capabilities for domains which have returned **finished** need not be treated specially (see section 7.1.6).

The other degenerate case is one in which domains always return **unfinished** from calls other than **finish** calls. Returning **unfinished** under these circumstances is never, by definition, invalid. The condition for making the **finish** call when overwriting a capability degenerates into a test for the accessibility of the domain. This is essentially the model discussed in section 6.5. On efficiency grounds, this is clearly a case to avoid. In addition, as one would expect, it exhibits the disadvantages expressed in section 7.1.1.

7.1.10 Summary of this approach

The effect of the proposal has been to identify a sub-graph of the graph of all possible domain references. This sub-graph consists only of those arcs which point to domains which have returned **unfinished.** There are two motivations for introducing this concept. The first is that it enables certain cases of inconsistent behaviour to be discovered earlier than would otherwise be possible. The second is that the sub-graph is smaller than the entire graph of domain references, and it should therefore be quicker for the protection system to scan when it is necessary to do so.

The handling of the degenerate cases shows up another property of the approach: the mechanism imposes only a small overhead in simple cases, and is particularly cheap if it turns out not to be necessary to use it at all. The overhead increases as the mechanism is used more, but if the mechanism is invoked only in circumstances in which it is needed, the cost should be modest.

7.2 An alternative approach

It may be the case that even the limited amount of work required to keep track of capabilities for domains which have returned **unfinished** is excessive. The cost of the mechanism is a consequence of the idea that the responsibility of one protection domain for causing the proper termination of another is expressed implicitly by the possession of capabilities. This means that whenever the graph of domain references changes, work may need to be done to find out whether a domain has been abandoned, and if it has, to intervene. A further problem which might arise is that the programmer cannot discover the status of a particular capability - i.e. whether he is expected to make the **finish** call or not [4]. There is an implicit assumption

----

[4] It would in principle be possible to provide a system call to make this information available, however.

that whether a domain will return **finished** or **unfinished** in any particular case will be documented as part of its specification, and the mechanisms provided in the protection system to force the call of the **finish** routine will be invoked only if those specifications are disregarded in such a manner that an inconsistent state arises.

If this turns out to be inconvenient, an alternative approach can be considered. This is based on the notion that the responsibility of one domain for the proper termination of another domain should be made explicit rather than being implicit.

7.2.1 The basis of the mechanism

The original proposal discussed in section 6.5 treated all domain references as equivalent. The refinement described in section 7.1 identified some domain references (those referring to domains which have returned **unfinished**) as different. This third scheme goes one step further in the same direction by suggesting explicit marking of certain domain references.

Consider figure 11 on page 93. In section 7.1.5 it was suggested that this could represent a system in which one domain, **B1**, is responsible for initialization and termination of the use of S, whereas B2 simply makes use of functions provided by S.

Suppose that for every domain, it is defined that there is precisely one other domain which is deemed to be **responsible** for it. This fact would be recorded as some attribute of a particular capability. In the example, one would expect **B1** to be nominated as responsible for S, and **A** as responsible for B1. This is shown in figure 15 on the next page, the specially marked references being shown as bold arrows.

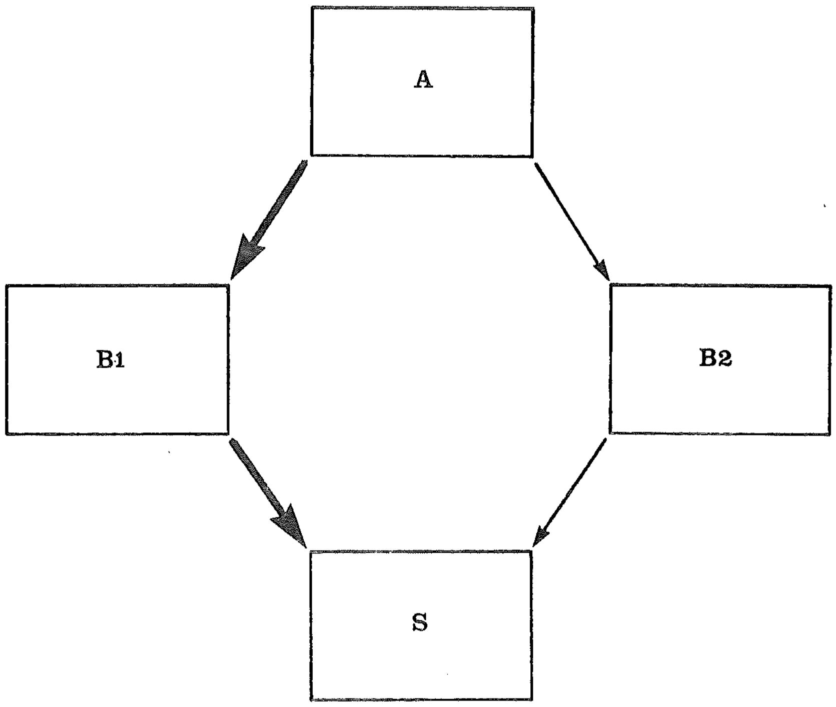

Figure 15: The responsibility tree and shared domains

The algorithm is based on the assumption that if a domain **D** is responsible for another domain **E**, and D destroys its capability for **E**, then the protection system will at that point intervene to allow **E** to tidy up. (At the same time, any domains for which E is in turn responsible will also be dealt with in the same way.) Since other references to **E** might still exist, it is important that it should not be called again in such a manner that it would return **unfinished.**

There are essentially two ways to achieve this effect. The first is for the protection system to invalidate all other references to the domain - i.e. to revoke access to it [5]. The other method is for the protection

---

[5] Whether this is possible will depend on the particular capability system. If there is a central object map, as described in section 6.5.1, revoking access to the object should be straightforward.

system to raise an exception within the domain to indicate to it that the specially marked capability has been destroyed. This must be done regardless of whether the domain last returned **unfinished** or **finished,** though in the latter case, reporting the exception may be delayed until the domain is next called. After this exception has been raised, it would be up to the author of the domain concerned to refrain from returning **unfinished** again [6].

Whichever method is chosen, the protection system must also check whether the domain whose marked capability was destroyed was in turn responsible for any other domains. If, in the example in figure 15, A overwrites its capability for **B1**, then it would be necessary to raise exceptions in both B1 and S.

In this scheme, the operation of moving capabilities becomes a little more complicated. The protection system must ensure that there is never more than one marked capability for a domain. In addition, it must be possible to transfer the marked capability from one domain to another; proper treatment of the example of returning a capability as result in section 7.1.7 (figure 14) requires this. This operation is separate from passing a capability for the <u>use</u> of a domain without transferring <u>responsibility</u> for it.

It must also be pointed out that the proposal loses much of its value if the sub-graph of marked references is allowed to loop. The mechanism would have little value if two domains could be marked as being responsible for each other. It is therefore required that the sub-graph of marked entries is a tree [7], called the **responsibility tree.** A straightforward way to enforce this is to make the rule that responsibility for a domain may only

----

[6] In particular, it may be part of the domain's specification that it will refuse to do any further work for its callers if this happens.

[7] To be strictly accurate, it can be a forest of trees, which each root being a protection domain on the current call stack.

be passed from domain A to domain B as a parameter of a domain call, if A is responsible for B; for a domain return, there are no restrictions.

7.2.2 Detecting inconsistent states

The rule for detecting inconsistent states can now be rephrased. It is invalid for a domain D to return **finished** if there exists a domain U such that:-

1. U has returned **unfinished**

and 2. D is responsible for U

The need for the third condition, which required a potentially large search, has gone away, because any references to U which are not part of the responsibility tree do not count.

7.2.3 Overwriting of capabilities

The action to be taken when a capability is overwritten depends on whether the capability is a specially marked one or not. If it is not, then nothing special need be done. If it is, then it indicates that a domain is overwriting a capability for another domain for which it is responsible. This means that one or more domains are being removed from the responsibility tree.

The protection system must make the **finish** call to any of these domains which last returned **unfinished**. In addition, precautions must be taken to ensure that none of them will return **unfinished** again, as discussed in section 7.2.1.

7.2.4 Summary of this approach

By identifying an even smaller sub-graph of the graph of domain references, the responsibility tree, this approach reduces even further the amount of work which needs to be done to detect inconsistent states. It does so at the cost of restricting the movement of domain capabilities to enforce a hierarchy of responsibility of one domain for another (though the freedom of ordinary calls between domains remains unrestricted).

A further feature of the system is that it requires more help from the programmers, who are forced to specify the responsibility relationship between the protection domains. There are arguments both for and against the desirability of this. It requires the programmer to do a little more work, but by requiring the programmer to supply more information about his intentions, it increases the likelihood of an error being detected by the protection system.

7.3 A comparison of the proposed solutions

The revised proposal of section 7.2 was presented as a more efficient refinement of the proposal of section 7.1. Ignoring efficiency considerations, which of the two techniques would be preferable?

This is a rather difficult question to answer without the experience of using implementations of both. If the mechanism were to be added to an existing protection system, then the original proposal would probably be the easier of the two to use. Only one new user level facility would need to be added to the high level language systems [8] - the ability to specify whether a return is to be **finished** or **unfinished** [9]. The revised proposal requires more to be done: facilities must be provided to enable the responsibility tree to be manipulated. Also, programs which do not need to use the mechanisms provided may be affected by them, in so far as they need to take account of the responsibility relationship between domains.

----

[8] This assumes that ordinary exception handling facilities are already present. The new exceptions raised by the **finish** call and the erroneous attempt to return **finished** should map cleanly onto language level exceptions which are raised during the operation which returns to a calling domain.

[9] The default return would presumably be **finished**.

In the first proposal, 'responsibility' for a domain is implicit. One can envisage circumstances in which a domain suddenly, through no action of its own, becomes responsible for the termination of a domain of which it has hitherto been simply one undistinguished user. The worst consequence this can provoke is that an exception will be raised if the domain attempts to return **finished**; unfortunately this is exactly what the program which has no special requirements of its own will do. Whether this would really be a problem is not known, but as mentioned earlier, suitable default exception handlers provided in language systems should enable the simplicity to be regained. In the revised proposal, this problem cannot occur at all, since all transfers of responsibility are explicit.

7.4 Implementation techniques

The detailed implementation of the proposals will not be considered here, since it will depend so much on the basic protection system. A number of general points are nevertheless worth making.

The computations necessary to implement the mechanism are by no means trivial. Moreover, some of the actions to be taken are associated with frequent events such as protection domain changes. It is important that the ordinary throughput of the system is not seriously impeded by the added mechanism. This implies that much of the mechanism should be implemented at the same level as the basic protection system; if domain changes are performed in microcode, then support for the proposed mechanisms should be put into the microcode too.

Implementing the entire mechanism in microcode is likely to be impractical. What is really required is a means of ensuring that the operations which do not require any special action to be taken are fast. A greater overhead on more unusual events can be tolerated. Consider, for example, the implementation of the definition of invalid **finished** returns in section 7.1.5 (p. 107). Parts 1 and 2 of the definition are good candidates for implementation in microcode. Recording the fact that a domain returned **unfinished** should be straightforward, and only a limited search is necessary to find out whether a domain possesses a capability for such a domain. Part 3 of the definition need be done only if at least one such capability is found, and could be implemented, if necessary, at a higher level.

Even if it is not feasible to do domain searching in microcode, it may nevertheless be possible to slave the results of a search done by higher level software. Provided that care is taken to invalidate the slaved information at the appropriate time, much repetitive searching may be avoided.

7.5 A note concerning protected objects

This chapter has been mainly concerned with the handling of complete protection domains, which were defined to consist of a set of capabilities for code, workspace and other resources. Some capability systems [10] allow the creation of protected objects, and allow capabilities for them to be manipulated. A capability for a protected object does not give direct access to the representation of the object; it must instead be passed to a type manager which has a capability which allows it to make use of the representation.

Many of the problems addressed in this chapter apply also to protected objects. A type manager may need to relinquish control while the representation of an object is in a non-standard state, so that it must be called again later. Note that there are now two cases to consider: either the capability for the object may become disused, or the capability for the type manager may be lost. In the former case the object should be passed to the type manager before it is finally destroyed; in the latter case the

----

[10] An example is the Hydra system [Cohen 75].

type manager would need to be given an opportunity to deal with <u>all</u> the objects it managed. Analogous techniques to those proposed for protection domains should enable this to be done.

8. Conclusions

This dissertation has explored a number of problems faced by the designers of protected operating systems. The emphasis has been on providing mechanisms which will enable the competent programmer to handle exceptional conditions in such a way that he can maintain the integrity of his protection domains regardless of the behaviour of other domains with which he may be cooperating.

In particular, a new class of exception has been recognized, which may be loosely expressed as the failure of one domain to call another when it should. The very nature of this exception, consisting as it does of a non-event, presents particular difficulties when attempting to provide a sensible algorithm to deal with it. Chapters 6 and 7 presented a solution to some of these difficulties and demonstrated that far more could be done about them in operating systems than typically is.

8.1 Evaluation

The problem was recognized largely as a result of observing some unsatisfactory properties of the operating system being built for the CAP computer. The provision of the mechanisms described in chapters 6 and 7 would improve matters considerably.

In particular, those mechanisms would enable the following improvements to be made to the CAP system:

- Input and output streams could be closed down properly, and device interlocks released, even if the user failed to close them explicitly.

- Resources associated with connections over a local communications network could be released at the proper time.

- It would be possible to provide user semaphores and other interlocks more easily.

- Ad hoc mechanisms would not be needed in the inter-process communication system (see section 5.2.3).

This raises the question of whether the mechanisms are of general applicability, or whether they are simply necessary to correct mistakes which were made in the design of one particular system. The author believes that the mechanisms are of more general value, whilst acknowledging that they have some limitations.

The ability of protection domains to retain internal state is not restricted to the CAP system. On the contrary, the use of CAP protected procedure calls in a coroutine like fashion is a little contrived, since it is a mechanism provided by language libraries rather than an inherent feature of the CAP architecture. Systems which implement protection domains as processes which communicate by message passing support the concept more naturally.

The main limitation of the proposed mechanism comes from the assumption that no attempt is to be made to handle failure of the mechanism itself [1]. There are applications, particularly those involving databases, for which this assumption is unreasonable; such applications maintain the integrity of

----

[1] This includes, for example, hardware failure of the computer on which the system is implemented.

something which has a longer lifetime than that of a single run of the operating system [2]. For many applications, however, such as those involving management of operating system resources, this assumption is reasonable; if the operating system crashes, some resources cease to exist and no longer need to be managed. For these applications, the provision of the proposed mechanisms should greatly simplify the job of enabling both the operating system and user subsystems to maintain their integrity.

8.2 Suggestions for further work

The main test for a mechanism such as the one proposed is whether it works in practice. An implementation would be valuable; deficiencies in the mechanism would come to light and possible improvements might become apparent. In a number of respects, the existing CAP operating system is ill-suited to this work; other systems may provide a better environment for experimentation. In particular, a new operating system for the CAP machine is under development [Herbert 78, Herbert 79]. The architecture of this system is rather different from that of the original system; it has a global object map and microprogram maintenance of use counts. It is also an example of a system based on message passing as the means of communicating between domains. It may prove to be possible to incorporate at least some of the ideas of chapters 6 and 7 into this system.

A further area in which more work may be valuable is in the treatment of system failure. When an operating system is restarted after a crash, it frequently needs to perform such activities as checking out its filing system and correcting inconsistencies. A non-monolithic operating system

----

[2] One frequently finds that database systems are taken as examples of the use of exception handling techniques to maintain permanent data structures in a consistent state. The possibility of total system failure is often neglected in such examples.

perhaps ought to enable user supported subsystems to do analogous things. Providing such facilities would remove the main limitation of the mechanism mentioned in section 8.1.

Finally, it would seem natural to extend the work to deal with distributed systems. A distributed system has much in common with a domain based protection system for a single machine: the components are protected from each other and the interfaces between them can be narrow and well defined. Distributed systems have the property that it is relatively common for one component of the system to suffer catastrophic failure whilst the remainder continues to function. The problems created by this possibility tend to be dealt with in a rather ad hoc fashion, relying almost totally on self-defence by the individual components. It would be useful to have a more unified approach to the problem.

References

Birrell 76	Birrell, A. D., **Attentions**, CAP Project Note 25th June 1976, (University of Cambridge Computer Laboratory internal document).
Birrell 77	Birrell, A. D., **System programming in a high level language**, Ph.D. Thesis, University of Cambridge, 1977.
Bourne 75	Bourne, S. R., Birrell, A. D. & Walker, I., **ALGOL68C Reference Manual**, University of Cambridge Computer Laboratory, 1975.
CAP 1	**CAP Hardware Manual**, (editor A. J. Herbert), University of Cambridge Computer Laboratory, 1978.
CAP 2	**CAP System Programmers' Manual**, (editor A. J. Herbert), University of Cambridge Computer Laboratory, 1978.
CAP 3	**CAP Operating System Manual**, (editor A. J. Herbert), University of Cambridge Computer Laboratory, 1978.
Cohen 75	Cohen, E. & Jefferson, D., **Protection in the Hydra Operating System**, Proceedings of the 5th Symposium on Operating System Principles, Austin, Texas, November 1975, pp. 141-160.
Cook 78	Cook, D. J., **The evaluation of a protection system**, Ph.D. Thesis, University of Cambridge, 1978, section 6.1.
Geschke 77	Geschke, C. M., Morris, J. H. & Satterthwaite, E. H., **Early experience with MESA**, CACM vol. 20, 8, August 1977, pp. 540-553.
Goodenough 75	Goodenough, J. B., **Exception handling: issues and a proposed notation**, CACM vol. 18, 12, December 1975, pp. 683-696.
Graham 72	Graham, G. S., & Denning, P. J., **Protection - Principles and Practice**, AFIPS Conference Proceedings 40, 1972 SJCC, pp. 417-429.

Harrison 79 — Harrison, D. J. & Thompson, C. E., **Cambridge Aids to Assembler Programmers**, University of Cambridge Computing Service, 3rd edition, May 1979, section 10.4.

Herbert 78 — Herbert, A. J., **A microprogrammed operating system kernel**, Ph.D. Thesis, University of Cambridge, 1978.

Herbert 79 — Herbert, A. J., **A hardware supported protection architecture**, Proceedings of the 2nd International Symposium on Operating Systems Theory and Practice, Rocquencourt, France, October 1978, ed. D. Lanciaux, North-Holland Publishing Co. Ltd., 1979, pp. 293-306. (Also reprinted, with minor revisions, as Appendix 1 of [Wilkes 79].)

Honeywell 72 — **MULTICS PL/I Language Manual**, Cambridge Information Systems Laboratory, Honeywell Information Systems Inc., 1972.

IBM 1 — **IBM System/360 Principles of Operation**, IBM System Reference Library, GA22-6821.

IBM 2 — **IBM System/360 Operating System: Supervisor Services and Macro Instructions**, IBM System Reference Library, GC28-6646.

IBM 3 — **IBM System/360 Operating System: MVT Guide**, IBM System Reference Library, GC28-6720.

IBM 4 — **IBM System/360 Operating System: Time Sharing Option Guide**, IBM System Reference Library, GC28-6698.

IBM 5 — **OS PL/I Optimizing and Checkout Compilers: Language Reference Manual**, IBM System Reference Library, SC33-0009.

Lampson 69 — Lampson, B. W., **Dynamic protection Structures**, AFIPS Conference Proceedings 35, 1969 FJCC, pp. 27-38.

Larmouth 76 — Larmouth, J., **Taming OS/360**, BCS Symposium on Software Engineering, The Queen's University of Belfast, April 1976.

Levin 77 — Levin, R., **Program Structures for Exceptional Condition Handling**, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1977.

Lindsay 77	Lindsay, B. G., **Exception Processing in Computer Systems,** Ph.D. Thesis, University of California, Berkeley, November 1977.
MacLaren 77	MacLaren, M. D., **Exception handling in PL/I,** Proceedings of ACM conference on language design for reliable software, ACM SIGPLAN notices, vol. 12, 3, March 1977, pp. 101-104.
Melliar-Smith 77	Melliar-Smith, P. M. & Randell, B., **Software reliability: the role of programmed exception handling,** Proceedings of ACM conference on language design for reliable software, ACM SIGPLAN notices, vol. 12, 3, March 1977, pp. 95-100.
Mitchell 79	Mitchell, J. G., Maybury, W. & Sweet, R., **MESA Language Manual,** CSL-79-3, Xerox Palo Alto Research Center, Systems Development Dept., Palo Alto, California, April 1979, chapters 8 and 10.
Morris 78	Morris, J., Private communication to J. J. Horning, reported in **Programming Languages,** lecture notes of an advanced course on computing systems reliability, University of Newcastle upon Tyne, 31st July - 11th August 1978, p. 20.
MPM 73	**The Multiplexed Information and Computing Service: Programmers' Manual,** Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., 1973.
Needham 71	Needham, R. M., **Handling difficult faults in operating systems,** Proceedings of the 3rd Symposium on Operating System Principles, Stanford, October 1971, pp. 55-57.
Organick 72	Organick, E. L., **The Multics System: an examination of its structure,** MIT Press, Cambridge, Mass., 1972, pp. 187-216.
Parnas 76	Parnas, D. L. & Würges, H., **Response to undesired events in software systems,** Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, October 1976, pp. 437-446.
Richards 79	Richards, M. & Whitby-Strevens, C., **BCPL - The Language and its Compiler,** Cambridge University Press, 1979.
Ritchie 74	Ritchie, D. M. & Thompson, K., **The UNIX Time Sharing System,** CACM vol. 17, 7, July 1974, pp. 365-375.

Schroeder 72a	Schroeder, M. D., **Cooperation of mutually suspicious subsystems in a computer utility**, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Mass., 1972 (MAC TR-104).
Schroeder 72b	Schroeder, M. D. & Saltzer, J. H., **A Hardware Architecture for Implementing Protection Rings**, CACM vol. 15, 3, March 1972, pp. 157-170.
Singer 80	Singer, D. W., **BCPL exception handling system**, Rainbow Group Note, 30th September 1980, (University of Cambridge Computer Laboratory internal document).
Slinn 77	Slinn, C. J., **Aspects of a capability based operating system**, Ph.D. Thesis, University of Cambridge, 1977.
Taylor 78	Taylor, R. J. B., **Process coordination and resource management**, Ph.D. Thesis, University of Cambridge, 1978.
UML 69	**Titan Machine Code Programming Manual**, University Mathematical Laboratory, Cambridge, 3rd edition, November 1969, Chapter 9 and Appendix D.
UNIX 79	**UNIX Programmer's Manual**, Bell Telephone Laboratories Inc., Murray Hill, New Jersey, 7th edition, 1979.
Walker 74	Walker, R. D. H., **The structure of a well protected computer**, Ph.D. Thesis, University of Cambridge, 1974.
Walker 81	Walker, R. D. H., **Notes on the E4 executive**, Private communication.
Wilson 71	Wilson, R., **The Titan Supervisor**, Dept. of Computer Science, The Queen's University of Belfast, 2nd edition.
Wilkes 79	Wilkes, M. V. & Needham, R. M., **The Cambridge CAP computer and its operating system**, The Computer Science Library, Operating and Programming Systems Series, Elsevier North Holland Inc., 1979.
Wilkes 80	Wilkes, A. J., Gibbons, J. J. & Singer, D. W., **Exception handling in BCPL**, Rainbow Group Note, 8th July 1980, (University of Cambridge Computer Laboratory internal document).