A RECURSIVE VIRTUAL MACHINE ARCHITECTURE -
IMPLEMENTATION AND ASSOCIATED PROTECTION PROBLEMS

BEST COPY

AVAILABLE

Variable print quality

TEXT BOUND
INTO
THE SPINE

ACKNOWLEDGEMENTS

The author would like to acknowledge the help given by the staff of the Newcastle University Computing Laboratory, most notably the supervision received from Dr H C Lauer, Mr P M Melliar-Smith, Professor B Randell and Dr C R Snow. Also he would like to thank his wife Pat without whose encouragement this thesis would never have been completed.

ABSTRACT

Investigations into the security of computer systems are becoming more important. With more people daily coming into contact with these machines, and personal data stored within the systems accessible in many cases to any knowledgeable user, the question of privacy of information held in these computer systems becomes paramount. A different approach, from that currently used, is advocated in this thesis. The implementation of a recursive virtual machine system is described, which it is proposed, will permit a highly secure approach to providing an inter-process protection structure in a computing environment.

On attempting to extend this system from permitting purely synchronous processing to allowing some form of asynchronous processing, it was discovered that this could not be achieved without a radical alteration in the initial design proposals. This thesis describes the synchronous machine implemented and provides a discussion of the problem of providing a generalized asynchronous processing mechanism together with a proposed solution to this problem.

CONTENTS

Chapter 1 :- Introduction and Overview of Thesis

1.1 Historical Background.

1.2 Hierarchical Computer Systems.

1.3 Extendible Computer Systems.

1.4 Structured Programming Techniques.

1.5 Terminology.

1.6 Access and Protection Mechanisms.

1.7 Virtual Machine Systems.

1.8 Virtual Machine Systems and Asynchronous Processing.

1.9 Summary of the Thesis

Chapter 2 :- Associated Computer Systems

2.1 Introduction.

2.2 Non-Hierarchical Computer Systems.

2.2.1 The GEC 4000.

2.2.2 HYDRA.

2.2.3 CAL-TSS.

2.3 Hierarchical Computer Systems.

2.3.1 VM 370.

2.3.2 The 'Hardware Virtualizer'.

2.3.3 The 'Virtual Machine Monitor'.

2.3.4 CAP.

2.4 Protection and Addressing Systems.

2.5 Conclusions.

Chapter 3 :- Design of the RVM

3.1 Introduction.

3.2 The Virtual Memory Addressing Mechanism.

3.2.1 The RVM Virtual Memory.

3.2.2 RVM Segment Tables.

3.2.3 The RVM Hierarchy of Virtual Machines.

3.2.4 RVM Hardware Addressing Mechanism.

3.2.5 The Address Translation Algorithm.

3.3 Environment Crossing.

3.3.1 Creating New Virtual Machines.

3.3.2 Returning To A Parent Virtual Machine.

3.3.3 Transfer Back To A More Abstract Virtual Machine.

3.4 Protection In The RVM.

3.5 Service Calls Across Levels.

3.6 Co-Operating Processes.

3.7 Applications of the RVM.

Chapter 4 :- Implementation of a RVM

4.1 Introduction.

4.2 The Burroughs' Hardware.

4.2.1 General Purpose Registers.

4.2.2 The Next Instruction Register.

4.2.3 The Scratchpad Registers.

4.2.4 Other Features.

4.3 Interfacing With The Burroughs' Supervisor.

4.4 Design of the Recursive Virtual Machine.

4.4.1 The RVM Addressing Mechanism.

4.4.2 The RVM Instruction Set.

4.4.3 RVM Program Control.

4.4.4 The RVM Data Stack.

4.4.5 The Environment Save Area.

4.4.6 Programming the RVM Interpreter.

4.4.7 Hardware Features of the RVM.

4.4.8 An Associative Store for the RVM.

4.4.9 Conclusions to RVM Design.

4.5 Performance of the Recursive Virtual Machine.

4.5.1 Performance of RVM's Address Translation Algorithm.

4.5.2 Experiments with the Environment Crossing Property of the RVM.

4.6 Conclusions.

Chapter 5 :- The Asynchronous Processing Problem

5.1 Introduction.

5.2 Virtual Machine Systems.

5.2.1 Asynchronous Processing in the RVM.

5.3 Capability Systems.

5.3.1 The GEC 4000 System.

5.3.2 The CAL-TSS System.

5.3.3 The HYDRA System.

5.4 Revocable Capability Mechanisms.

5.5 Conclusions.

Chapter 6 :- A Mechanism to Permit Asynchrony In The RVM

6.1 Introduction.

6.2 Requirements For A Solution.

6.3 Considerations of Efficiency.

6.4 The Proposed Mechanism.

6.4.1 The Environment Protection Semaphore.

6.4.2 The 'Called Route' Segment.

6.4.3 Associative Memory Considerations.

6.4.4 Table of Protected Environments.

6.4.5 Appraisal of Mechanisms.

6.5 Refinements To The Proposed Scheme.

Chapter 7 :- Conclusions

7.1 Introduction

7.2 The RVM's Relationship With Other Structured Computer Systems.

7.2.1 Revocable Capabilities and Asynchronous Processing.

7.3 Topics For Further Study.

7.3.1 Implementation of an Asynchronous RVM.

7.3.2 Optimum Associative Memory Size.

7.3.3 Alternative Environment Protection Algorithms.

7.3.4 The Problem of Only Protecting Shared Objects.

7.4 Conclusions.

CHAPTER 1 - INTRODUCTION AND OVERVIEW OF THESIS

1.1 Historical Background

Since their initial conception, computer systems have evolved rapidly from very large, fast calculating machines into tools enabling people in all walks of life to collect and examine data of various kinds. However, despite increasing sophistication there has been little change in the overall structure of the computer system as seen by the user, and it is to this topic that this thesis relates.

Initially computers were used by a single person who, by manipulating the switches and keys of the machine and writing programs in machine code, was able to use the computational power available for the solution of problems previously beyond his scope. As more people began to realize the possible uses of computers, schemes were devised in order that their usage could be made more efficient. This led to the production of separate processors for providing input and output facilities, and later programs were written to schedule the various computer facilities between several users. These programs formed the basis of current operating systems and were designed to permit a growing population of potential users to access computers while providing a solution to the problem of resource contention. As more users came to share the facilities of a particular computer system so the need to protect the operating system from user program malfunctions, and also the users from each other, became increasingly more important.

Historically, this requirement for a protection scheme to be provided in a computer system led to the development of the 'two-state' machine. In such a system a program may execute in one of either the 'system' or 'user' states, the operating system being the only suite of programs permitted to execute in the privileged system state. All other programs could only execute in the user state. Any program executing in the system state had complete access to all the resources of the configuration together with the ability to perform certain privileged operations. A user program, however, had the resources available to it limited by the operating system, and was unable to perform any of the privileged operations. Such a scheme can be used by the operating system to provide an excellent protection mechanism between the users of the computer system, assuming that the operating system is functioning correctly. Unfortunately as computer systems have increased in size so have the operating systems associated with them; as a result most of these systems do not provide the clean protection mechanism they strive for. Consequently a malicious user often can gain access to parts of the system for which permission would not normally be granted. The result of such an act may cause the loss, or change, of certain items of private information. This information could be owned by another user of the system or the system itself, and the result of such an action could possibly cause the system to malfunction and then to 'crash'.

Alternatives to a monolithic computer system architecture such as this have been devised and implemented. Another alternative is proposed in this thesis and it is hypothesised that should such an approach be adopted, then any necessary operating system could be more easily written and tested, and a clearer approach to providing the necessary protection between processes, be they initiated by users or the operating system, would be possible.

1.2 Hierarchical Computer Systems

In an attempt to model complex computer systems, Zurcher and Randell adopted a multi-level modelling technique based upon the concept of describing the system at several 'levels of abstraction' [ZR 68, Ra 69]. It was claimed that several representations of a system can, and can usefully, coexist. These representations are at different levels of abstraction.

Example: An input spooling process views its output as files which are stored somewhere on a disk. The disk file handler which provides the files, is concerned with physical disk drives, segments and tracks etc. The representation of a file is said to be at a higher level of abstraction than that of a segment or track.

A similar approach to computer system design was adopted by Dijkstra in his design of the 'THE' Multiprogramming System [Di 68b]. Here a strictly hierarchical structure was adopted, all representation of objects at one level of abstraction being composed of representations of objects at lower levels of abstraction. A primary design aim of this system was a high degree of reliability which could readily be demonstrated. In fact, Dijkstra was able to check each state of the system at each level in the hierarchy, thus more easily illustrating the overall reliability of the system.

A review of computer systems which have adopted a multi-level approach to operating systems design clearly indicates the benefits to be gained. Perhaps the most widely publicised of such systems is 'MULTICS' [CV 65, Or 75, SS 72]. This system was designed with the four criteria of functional capability, economy, simplicity and programming generality uppermost while providing the most useful set of access control mechanisms in a computer utility. The designers considered that if these mechanisms could be easily understood then this was the best way to achieve confidence in the system. The hierarchical approach to computer system design was implemented in MULTICS as a series of concentric rings of protection [SS 72]. Each ring represented an environment of resources at increasingly higher levels of abstraction. The initial intention was to support a multi-ring supervisor system, as in the THE system. The main difference being that the MULTICS structure was to have run time significance whereas the THE structure was purely a design and implementation tool. However hardware constraints have meant that the MULTICS supervisor essentially resides in one ring, at level zero. It is important to note that only level zero processes are allowed to perform asynchronous Input and Output operations, and although modifications to the hardware have permitted more supervisor functions to be provided at more abstract levels, the rules concerning asynchronous I/O still apply. This is a problem which is central to this thesis, and it is discussed further in Chapter five.

1.3 Extendible Computer Systems

Schemes such as those just mentioned have demonstrated the advantages of structuring an operating system in a hierarchical fashion. Present computing trends are towards users interacting with complex packages which do not form part of the operating system but execute as user tasks. In many cases the designers of such packages wish to provide an abstract machine with which the user can interface. There is a requirement to provide a user's program with some objects, which may be of an abstract nature, and to prevent this user program from accessing these objects in any manner other than that prescribed.

Examples:

i) A data base system which provides new file handling facilities but wishes to prevent users of this system from accessing their files via the standard operating system file operations.

ii) An APL system which provides a new level of more complex arithmetic operations than those of the machine hardware.

The various ways in which the extra facilities currently are provided in these two examples illustrates the sharp contrast between the current methods used. A popular way of implementing a Data Base Management scheme is to provide a collection of subroutines for the use of the interacting program. This approach permits the user a great deal of flexibility when designing his program. The program can be written in any programming language and will be as efficient as the programmer makes it. The only constraint on the programmer is that all interactions with the data base be made via the supplied subroutines. However since the program appears

to the operating system like all other user programs there is no run-time constraint to prevent direct interaction with the data base via the standard operating system filing routines.

On the other hand, many APL systems are completely interpretive. They provide, as an abstraction of the actual machine, a new set of instructions which form the APL instruction set. Such a scheme certainly prevents users from directly interacting with the operating system, however it does not permit a user to access certain primitive functions of the machine, eg. 'ADD', 'SUBTRACT', etc., which, for efficiency considerations, may be desirable. There is in this case some loss of efficiency due to fully interpreting each abstract instruction.

Clearly, the desirability of providing an abstract machine on which a programming system can run is closely linked with the level of abstraction methodology. The major requirement is the provision of mechanisms to enable such systems to be extended beyond the boundaries of the basic operating system. Such systems have been designed, and of note are the Project-Sue system [At 72a, At 72b] and the RC400 system designed by Brinch Hansen [Br 70]. Both these systems are more ambitious than the 'THE' system, and each provides a system 'kernel' ('nucleus' on the RC400) which handles machine interrupts and provides some synchronisation primitives. The remainder of the operating system functions in the same way as a user program, new levels of the system providing a more abstract machine environment in which processes may run. Also, since both user programs and operating system programs can have the same facilities, it is easy to see how such a scheme can be extended beyond the operating system into the provision of user programming packages.

1.4 Structured Programming Techniques

In parallel with these developments which provided a hierarchical and extendible computer system, proposals were being made with regard to the design and construction of programs. A more 'structured' approach to program development has been advocated [DHH 72] and this, associated with the concept of user defined 'types' [Br 73], has enabled programs to be built in a similar manner to that discussed when decomposing operating systems using the level of abstraction concept. The use of a structured programming methodology has been shown to increase efficiency in programming effort and to lead to fewer errors being introduced into these programs.

It is to be expected therefore, that by combining structured programming techniques with the level of abstraction methodology, operating systems can be built which are well protected, error free and thus reliable.

1.5 Terminology

Before any further discussion is attempted there must be some definition of the terminology which is being used.

The definition of a process which is used here is that given by Horning and Randell [HR 73], in which a process is defined as a triple (S,f,s) where S is a state space, f is an action function in that space, and s is the subset of S which defines the initial states of the process. A computation is defined as a sequence of states from the state space S, obtained by applying the action function f, first to an initial state and then to each succeeding state. A process thus generates all the computations generated by its action function from its initial states. This definition is, by its nature, very low level. However by using the techniques of combination and, abstraction and refinement discussed by Horning and Randell, it is possible to deduce new 'higher' level processes which may be more appropriate to the requirements of a particular application.

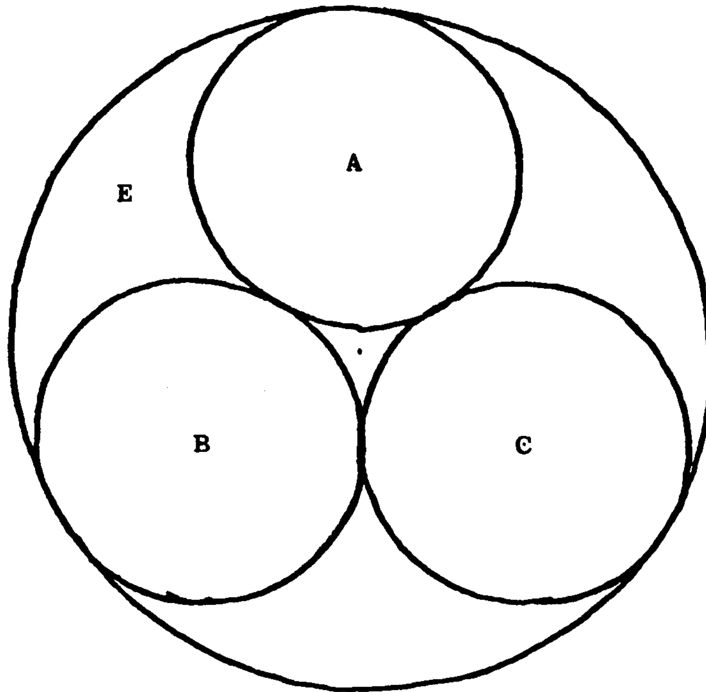
Access to an object is defined by an algorithm within the system. It may be implemented in hardware, micro-code or a piece of program and there may be a number of ways in which a particular object can be accessed. For example, an object which is a segment of memory could have its possible access rights defined as 'read' or 'write' a word, or 'execute' an instruction; an object which is a procedure may have 'execute' or 'read' access rights; an object which is a physical device, eg. a line printer may have 'access' or 'no access' access rights.

Within a system the objects of which it is comprised can be divided into disjoint sets by type, all objects of a particular type having the same set of defined access rights. Thus for any object which is a segment of memory the access rights 'read' and 'write' a word, and 'execute' an instruction would be defined. Of course, for a particular process, the protection system may limit the access of some object to a subset of those defined for that type. For example, the only permissable access of a given segment of memory may be 'execute' an instruction. The dynamic creation of new types and the deletion of old types throughout the lifetime of a system considerably helps in producing a general, extensible system.

The notion of an environment or a 'domain' [Lam 69a] is fundamental to protection as it forms the means of structuring a system for protection purposes. It allows processes to reference a group of objects, and associates the permitted access rights with these processes for the objects. An environment can be defined as that entity which specifies, at each instant in time, the objects available to any process within that environment. Furthermore, the environment also specifies the manner in which any of its processes may access the objects currently available. Usually the permitted accesses to an object will be a subset of those defined for the type of object. A process may execute within a number of environments during its lifetime, though at a particular instant, it will be associated with precisely one environment.

In order to permit processes to fully exploit such an access control system, it is desirable both to allow the set of access rights constituting an environment to change and to allow a process to switch from one

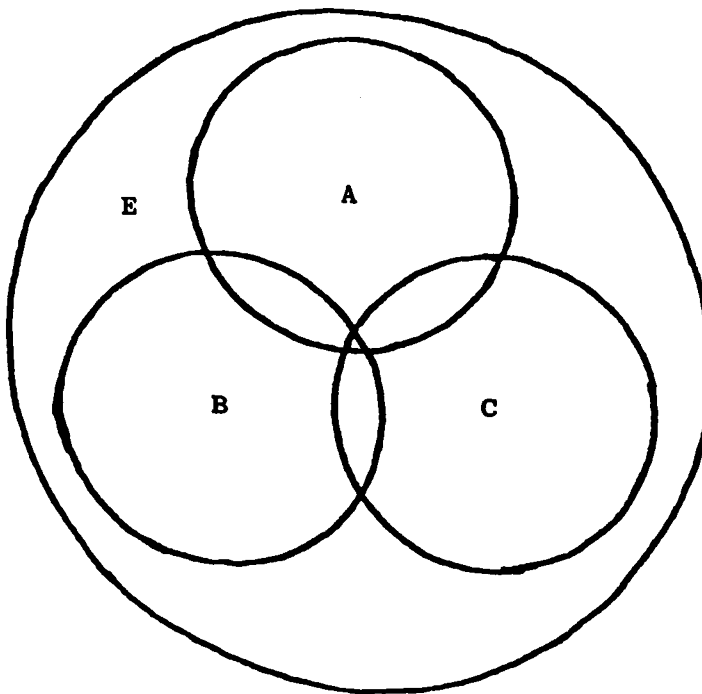
Environments Within A Computer System



The large circle, E, represents the set of objects constituting the real computer system.

The smaller circles A, B and C represent that subset of objects from E constituting the environments A, B and C respectively.

a) Set of environments with no shared objects.



b) Set of environments with shared objects.

Figure 1.1

environment to another. The switching of environments may be a less efficient operation than that of making a small change to the contents of an environment, either by adding an object to, deleting an object from, or changing the access rights to an object in the environment. By a series of such changes it may be possible to avoid environment switching altogether; however environment switching is not be be prohibited since it may provide, in some cases, a more efficient and often a conceptually cleaner solution to a particular protection problem.

With the basic constituents of a computer system defined, a computer system can be defined as well-protected if and only if processes can at all times only access those objects currently specified by their environment.

For example, referring to figure 1.1, for the system to be well protected a process within environment A must never be able to access objects within environments B or C. This does not preclude processes in different environments from sharing access to objects, as illustrated in figure 1.1 (b).

In order to provide an environment a function, f, is provided which maps the abstract objects of this environment into real objects of the underlying computer system. The environment E_A is defined by the mapping function f_A of the environment constituting the whole computer system, E.

ie. $E_A = f_A(E)$.

If the environment E_A is defined in terms of some intermediate abstract machine which in turn is mapped into the real computer system then the mapping function f_A may be some function of the mapping function f_B.

ie. $E_A = f_A (E) = f'_A (E_B) = f'_A \left(f_B (E)\right)$

The number of intermediate mapping functions used to establish a particular environment defines the level of abstraction, i, of the environment.

The notion of a process migrating from one environment to another has already been mentioned briefly. One example of this activity can be considered as occuring each time a basic machine instruction is executed in an abstract machine environment. In order to perform the machine operation the process involved is transferred to the least abstract environment of the computer system and the required objects mapped into real machine objects, words of core memory for example. In this case the machine hardware uses the mapping function supplied in order to transform the abstract objects of the calling environment into real machine objects; and the only transformation performed is that for those objects actually required for the particular machine operation involved. The alternative approach, of redefining the calling environment so that basic machine operations can be performed directly, is considerably more cumbersome if carried out for each abstract machine operation as it may be impossible to decide in advance which objects need to be renamed.

In order for a process to move from one environment to another, any objects which are to be shared between the two environments must be renamed in the new environment's terms. This renaming will involve a number of mapping transformations dependant upon the level of abstraction of each environment.

The number of transformations performed when a process migrates from one environment to another is defined as the distance, d, between the environments.

A Set of Structured Computer System Environments

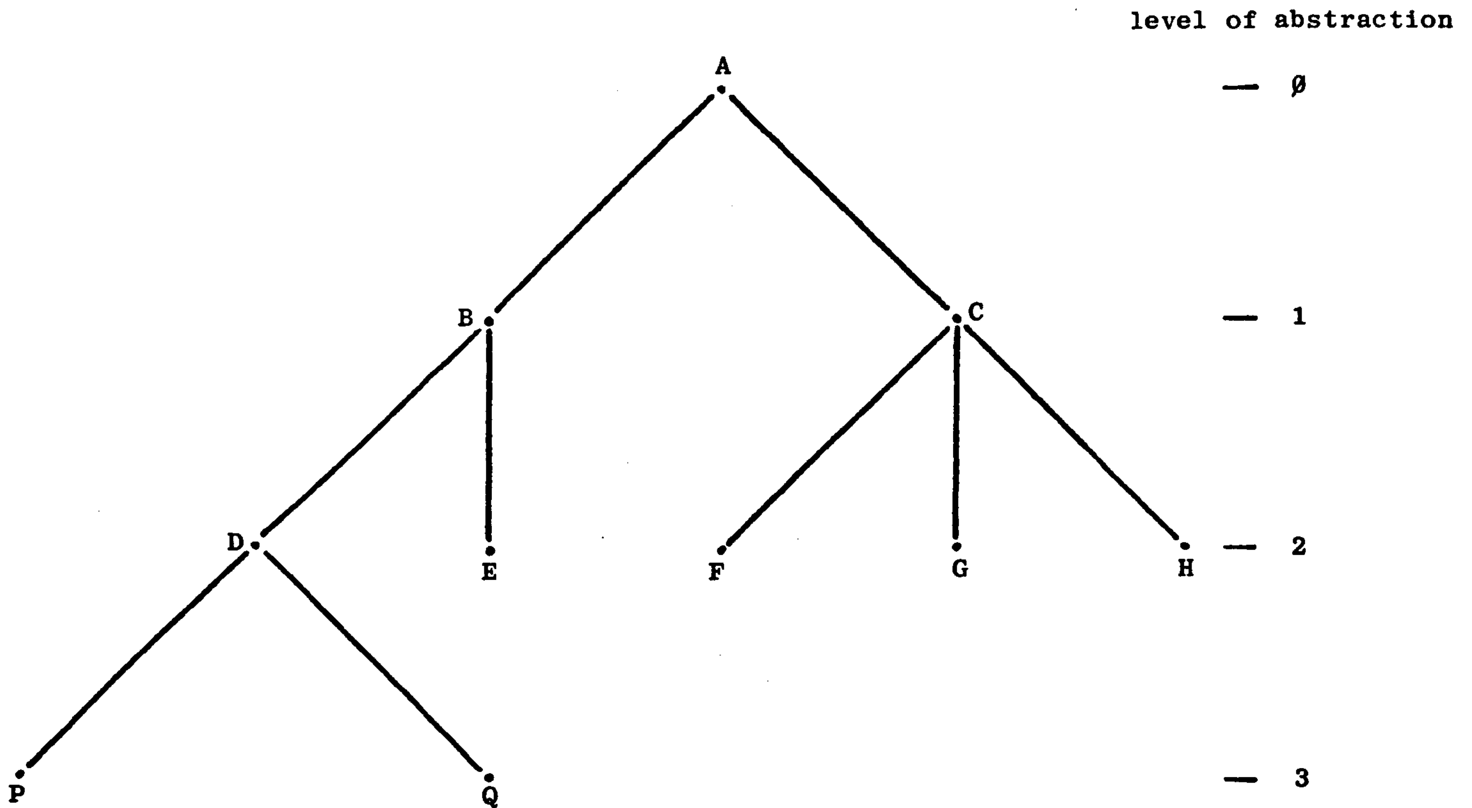


Figure 1.2

Examples:

Referring to figure 1.2

Environments P and D are separated by a distance 1

Environments P and Q are separated by a distance 2

Environments Q and E are separated by a distance 3

Environments D and G are separated by a distance 4

One critical factor when discussing structured computer systems is their overall efficiency, especially when considering the performance of the machine at more abstract levels than that of the basic hardware. It is therefore essential to provide a criterion for judging whether or not such a system is efficient.

A structured computer system will be deemed efficient if the cost, C, of a process being transferred between two environments is at worst directly proportional to the distance between the two environments.
ie. $C_{A.B} < K \times (d_{A.B})$

In practice K should be minimized, and it will be shown that, in the case of the 'recursive virtual machine' described in this thesis, techniques exist which reduce K to acceptable proportions, less than 1 in many cases. However if the cost function is not linear, but is squared or exponential for example, it is considered that whatever optimizing techniques are employed to reduce K's value, it will never prove feasible to construct systems which require an indefinite number of levels to be crossed. Within a particular system there may be several cost functions, K, each dependant upon a particular type of inter environment transfer. This is perfectly acceptable provided that each cost function is at worst linear with respect to the number of levels crossed.

With these definitions in mind the following points should be noted with regard to a well-protected computer system.

i) An environment, which exists at any level in the hierarchy, must be unaware of the environments at less abstract levels which have mapped the basic machine objects into a set of virtual machine objects.

ii) A process existing within an environment should be able to regard the environment provided to it as a machine in which it executes. The operations of this machine should appear reliable, atomic and deterministic; though the program may of course interact in a non-deterministic manner, through the machine, with some other process known to it.

iii) If a process creates subsidiary environments, then erroneous operation of the process can damage these subsidiary environments.

iv) If a process within an environment collaborates with another parallel process, possibly through the use of shared data, then erroneous operation of the process may present invalid or meaningless data to the parallel process.

v) Erroneous operation of a process must never be able to damage those processes which provide it with its environment; furthermore, any program which creates a subsidiary environment must protect itself against any program which is executing within that environment. For example, if a program gives a process in a subsidiary environment access to the data which defines that environment, then the supervisory program must be considered responsible if this data is altered.

vi) It must be ensured that it is impossible for a process to construct new 'privileged' environments maliciously, which, because of some knowledge of their immediate supervisor, can damage other, less abstract environments.

vii) It must be impossible to usurp the total machine's resources by any malicious process.

1.6 Access and Protection Mechanisms

Having discussed some of the reasons for providing a well-protected and extensible computer system, it is perhaps pertinent to mention the mechanisms which have been used to provide such systems. The 'access matrix' scheme discussed by Lampson [Lam 71] and used in Project Sue [At 72b] associates a set of access attributes for each object in the system with each environment that exists in the system. This is an extremely general approach and it enables the system to determine at any instant whether or not a particular process has access to each object. A further scheme is the 'lock and key' approach discussed by Needham [Nee 72], whereby access to an object is granted only if the environment containing the process which is requesting the access has the 'key' for this object. Such a scheme is clearly more efficient on storage space than the access matrix, but it may be slower in searching for the keys to a particular object. Both of these schemes are, in effect, different implementations of the 'capability list' mechanism first discussed by Dennis and Van Horn [DV 66] and later further described by Lampson [Lam 69a].

Initially, implementations of capability machines did not permit the redefinition of user defined objects. MULTICS [Or 72], for example, employed a capability mechanism for its access control and addressing schemes, but was unable to permit the renaming of objects. These implementations held a master table of all possible objects in the system and then an executing process could only gain access to a particular object if it currently owned a capability for this object. Thus each environment,

in the system is defined by the capability list mapping real objects into abstract objects of that environment.

Later schemes were devised to permit the renaming of objects in a system, however these tended not to adopt the strict hierarchical approach taken by the MULTICS designers. The Plessey System 250 [En 74] is one such realization of a computer system which has based its addressing structure and protection policy on the implementation of a capability list mechanism. The flexibility of such a mechanism has, in this case, resulted in a system which is an ever changing list structure that can be arbitrarily complex in its interconnections.

The work on the HYDRA system [Wu 74, WLP 75] has shown that a number of protection problems can be solved using the capability list approach. The arguments in favour of this approach show that most protection problems can be sensibly solved using this technique. A serious criticism of HYDRA, and other capability systems, has been the inability to 'revoke' capabilities passed on to more abstract environments. That is to say that once an environment has been passed the capability to access a particular object it cannot be forced to return this capability to the less abstract environment. In this way it is possible for processes to misuse the system by accumulating more than their fair share of system resources. It is the solution of this problem, although it appears in a different guise, that is fundamental to this thesis.

At the Stanford Research Institute, it is proposed to use the capability list approach in order to build a provably secure operating system [Neu 74]. The intention at SRI is to structure the system hierarchically, so that using the level of abstraction methodology the proof of security

may be more easily obtained. In this scheme it is intended to use a 'revocable capability' mechanism similar to that described by Redell and Fabry [RF 74] in order to over come the serious problem discovered in the HYDRA system. However, two questions remain to be answered in this system, the first concerns the overall efficiency of the system and the second concerns a problem of revocable capabilities in systems permitting asynchronous processes.

With regard to the overall efficiency of the system it is interesting to note that other hierarchically designed capability systems, most notably MULTICS, have reverted to two-level systems due to the inability to permit efficient execution of processes at more abstract levels. The effect of this approach is to have a maximum distance between environments of 2, thus lessening the cost of processes being transferred between environments. This cost of transferring between environments was a major factor in the design of both the HYDRA and Plessey 250 systems. As a result a non-hierarchical structure of environments is provided, with the distance between environments being 1 in the majority of cases where processes need to migrate between environments. A hierarchical structure of environments is permitted, but it is accepted that there will be a loss of efficiency in their approach in this case.

The flexibility of the capability mechanism permits a structured computer system to be built in a non-hierarchical manner, and also enables many protection problems to be solved. The introduction of the 'revocable capability' mechanism further provides a solution to the particular problem of subordinate environments not returning capabilities for objects when they are no longer required. However it is conjectured that

a further problem, concerning the interaction of asynchronous processes, is introduced if revocable capabilities are used, and this problem will be discussed in depth later in this thesis.

1.7 Virtual Machine Systems

The 'virtual machine' approach to computer system design provides another approach to providing a well-protected and extensible computer system. This approach has naturally followed from the use of 'virtual memory' in large multi-user systems [De 70, Pa 72]. The virtual memory concept provides a protection and allocation mechanism for controlling the memory resources of a computer system. Users are separated from each other and the operating system by each being allowed access to a particular abstract machine with its 'virtual memory' defined by the operating system.

Virtual Machine mechanisms, such as CP 67[MS 70, Pa 72], present to both users and designers a set of computer systems which are protected from each other. The virtual machines resemble the existing hardware of the real machine, and in some cases it is possible to re-define recursively another virtual machine from a user job already running in the system. The main feature of virtual machine systems is not however this recursion, but the ability to support a variety of different operating systems for the hardware of the machine.

The ease with which a virtual machine system can be re-configured to support several operating environments has led to the proposal of a virtual machine system in a naval tactical environment [PH 76]. In this proposal a degree of redundancy is permitted when supporting a computer configuration on board a naval vessel. In a normal situation this redundancy is provided by a substantial overhead of expensive hardware and software, and it is hoped that by employing virtual machine techniques

this can be considerably reduced. The arguments proposed to support such a system cite the flexibility, security and performance of a virtual machine system which has been properly designed in the areas of hardware and software.

An example of an existing, and widely used, virtual machine system is CP-67. This system was designed to run on an IBM 360/67 configuration and originally provided users with a virtual IBM 360/65 on which their programs could be run. The basic difference between the two machines is a hardware paging box which forms part of the IBM 360/67. This paging box is used to map automatically all the virtual memory addresses used on the machine, and provides the virtual memory requirements of CP 67. An ordinary user would interface to the machine via the Cambridge Monitor System, a simple single user operating system. He then was given the impression of having, totally at his disposal, an IBM 360/65. Other users could however interface to the system via the standard OS operating system or even directly program the 'bare' machine. Unfortunately this arrangement did not permit portions of the CP 67 operating system to be tested as a user job and later versions were produced which did this by providing users with a virtual IBM 360/67 on which programs could be run. This development led to the VM370 system, now widely used on IBM computers.

The ability to provide recursively one abstraction of a machine architecture on top of another clearly allows the basic machine architecture to be extended. Any virtual machine which has the capability to recursively create a further abstract machine could provide an abstract 'APL' or 'Data Base System' machine. This illustrates the flexibility of this approach when

structuring an operating system. Starting from the basic types of objects of this computer hardware, more abstract virtual machines can be built which provide to their users higher level types of objects and these objects can then in turn be used to produce even more abstract objects.

Example:

In a Data Base system the following abtractions apply:-

A Data base is a set of files,

A file is a collection of sectors on a disc.

The usefulness of an implementation of such a scheme on conventional 'two-state' computer architecture is questionable however, since most processor exceptions will force control to an environment which is permitted full supervisor-state privileges. This might not be the desired sequence of events, for the facility to recover from this exception may be provided by a process in an intervening environment. In terms of our definition of the efficiency of the system, the distance between two environments is increased by twice the level of abstraction of the environment actually providing the required service.

Example of CP67 Operation

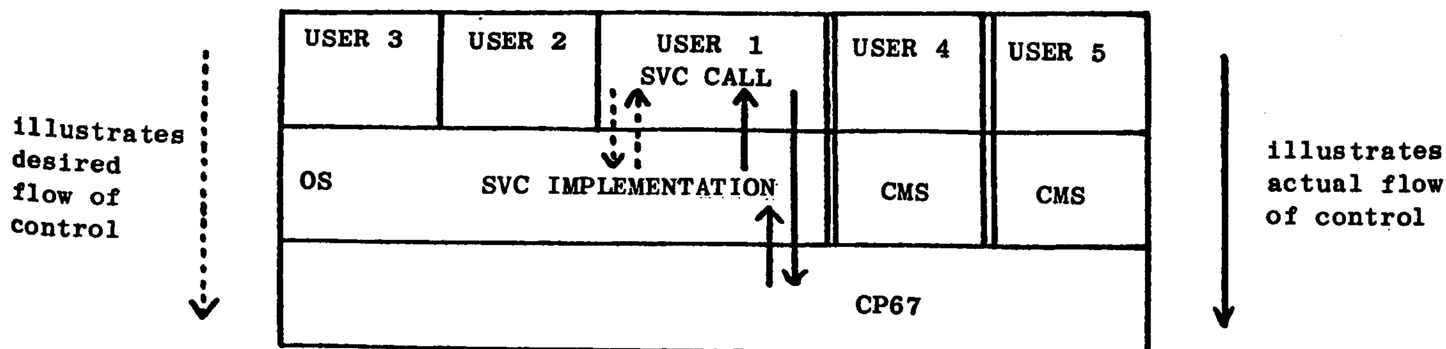


Figure 1.3

Example:

In the case of CP 67 any attempt by a process in some environment to issue a 'supervisor call' operation will result in control being automatically passed back to CP 67 itself. This may not be the desired action. For example if the system is currently in the state described in figure 1.3 and a process user 1, issues an OS supervisor call operation, then the sequence of control is the following. The user 1 process is stopped and control passed to CP 67. The supervisor call process within CP 67 then determines that this is an operation provided by the OS environment it has set up and passes control to this environment. The required operation can then be provided and user 1 restarted. Clearly however the desired flow of control is merely from user 1 to OS and back again directly. The distance between the two adjacent environments when user 1 calls OS is, in this case, 3 instead of the desired 1.

It is the presence of a supervisor state which causes the problem just described, and it should be noted that the effect of this problem is seriously amplified if virtual machines are permitted to redefine other virtual machines recursively. In fact the amplification effect is so bad that although a recursive version of CP 67 exists, now known as VM 370, only a single level of recursion is possible due to efficiency considerations.

The necessity of the supervisor state has in fact been seriously questioned by Lauer and Snow [LS 72] and further discussed by Buzen and Gagliardi [BG 73]. As a result of these discussions, together with the considerations of performance necessary for providing a useful system, several new systems designs have evolved. These systems simplify,

generalize and make recursive the architecture of hardware capable of supporting virtual machines [LW 73, Go 73, BH 75]. The adoption of a computer architecture design to support any number of virtual machines recursively, appears to provide a mechanism for hierarchically structuring a computer system and also provides a means of extending the system to any required level. A 'recursive virtual machine' architecture is therefore proposed herein, whose design follows that described by Lauer and Wyeth [LW 73] and Lauer [Lau 74]. Attempts to implement this design in micro-code on a Burroughs B1700 [Bu 72] have shown the feasibility of such a solution in terms of efficiency and performance. The details of this performance evaluation are given in section 4.5 of chapter four.

A

B

C segments

on secondary

segments

storage

in

X
card
input
buffer

core

segments

on secondary

storage

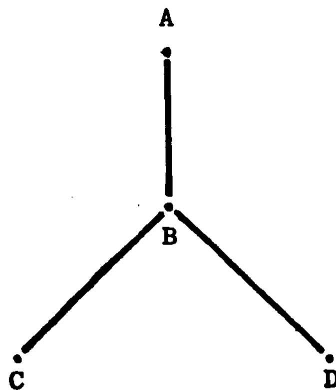
segments

in

D

core

Allocation of objects to environments



Hierarchical structure of environments

Figure 1.4

1.8 Virtual Machine Systems and Asynchronous Processing

During the implementation of the recursive virtual machine, described in chapter 4, an interesting problem arose concerning the implementation of asynchronous Input and Output facilities. The problem arises that the protection of the system can easily be broken while any process is having an I/O operation performed in parallel with any other normal computation in the system.

This problem can be discussed simply, with relation to figure 1.4. If a process within C wishes to read a card from the card reader into a section of its virtual memory, named X, then this I/O processing may be taking place while another process within environment B can execute. If this process is permitted to execute it may decide to schedule a process within environment D. Being unaware of C's asynchronous activity, and finding a need to provide D with more in core segments, it may move C's core segments onto secondary storage passing the freed core memory to D. All this would be acceptable except that C now has access to objects outside its environment, the input from the card reader overwriting a segment belonging to D.

This is a simple example of the problem of providing asynchronous processing in the recursive virtual machine. Further investigation of the problem shows that it is not merely one of providing asynchronous I/O operations, but concerns generalized asychronous processing, as encountered in a complex multi-programming computer system. The cause of the problem lies in the fact that while a process in an abstract environment, C, is having an operation performed in a less abstract environment, A, any

process in an intermediate environment, B, can redefine any more abstract environment under its control

It is postulated that this problem is the dual of the revocable capability problem in capability based systems. For the renaming of objects in recursive virtual machine systems can be likened to the revocation of capabilities in capability schemes. Whereas in many capability schemes revocation of capabilities was not originally permitted, in recursive virtual machine systems a mechanism is needed to prevent the renaming of objects during certain critical times.

1.9 Summary of the Thesis

The aim of this thesis is to illustrate the feasibility of a virtual machine approach to providing a clean protection system. It is advocated that a good computer system should exhibit the following properties:-

i) It should be well protected, and thus reliable.

ii) It should permit the renaming of objects between environments of the system, and thus extensible.

iii) It should permit a hierarchical structuring of environments, thus simplifying the system and thus making it more readily demonstrated to be correct.

iv) It should permit any number of processes within the system to co-operate asynchronously.

v) It should be efficient overall, thus enabling users of the system to be given useful proportions of time from the various processors which constitute the system.

The implementation described in chapter four provides all of these features except asynchronous processing, and a mechanism which will allow this feature to be integrated into the system is proposed in chapter six of this thesis.

Chapter two discusses other protection systems in more detail, namely VM 370 [Pa 72] and other virtual machine systems, the GEC 4000 system [GEC 74], HYDRA [Wu 74, WLP 75], CAL-TSS [Lam 68, Lam 69b, LS 76] and CAP [Wa 73, NW 74]. It is argued that all suffer problems which though apparently different are in fact all manifestations of the difficulty of providing a computer system which satisfies the five criteria already

mentioned. Furthermore it is proposed that both the capability based systems and the virtual machine systems are essentially the same.

In chapter three, the design of a Recursive Virtual Machine architecture, the RVM, is discussed. The 'special' mechanisms required to permit processes to execute in environments at any level of abstraction are considered together with the constraint that each RVM operation should be accessible to all processes at any level, providing that each environment at all lower levels has permitted its use. Questions of efficiency are raised, especially with regard to the extra processing required to access an object within an environment at a more abstract level than the bare machine. Finally mechanisms are proposed which permit the automatic by-passing of environments for the type of operations commonly considered as 'supervisor call' operations. In this way, it is proposed that the RVM eliminates the problems of implementing supervisor calls which were discovered in VM 370.

Having discussed an RVM design, chapter four describes the implementation of a purely synchronous RVM. This is realized by writing an interpreter in the micro-code of a Burroughs B1726. The instruction set chosen, in order to provide all the facilities required by the RVM, is discussed and tests are described which illustrate the feasibility of executing programs in environments at more abstract levels than that of the bare machine. In order to increase the efficiency of the RVM an associative store is introduced and the tests are repeated in order to assess the usefulness of this mechanism. Figures of the performance of the RVM while executing these test programs at different levels of abstraction are discussed and it is claimed that this implementation fulfils all of the

requirements of the RVM architecture described in chapter three, except that no asynchronous processing is provided.

In chapter five the problems of providing asynchronous processing in virtual machine systems and the associated problems in other protection systems are examined. The dual problem to asynchrony in virtual machine systems is seen to be the 'Revocable Capability' problem in capability based systems. It is postulated that the reason why asynchronous processing causes a problem in virtual machine systems is that access to any object can be revoked at any time by a process in an environment which has created the object. On the other hand in a capability based system asynchronous processing is possible because the revocation of access rights is not permitted. The Revocable Capability [Re 74, RF 74] is seen as a solution to a serious failing in both the HYDRA and CAL-TSS systems, however this solution is seen as an extension of the virtual machine mappings employed in the RVM. Thus it is postulated that the use of revocable capabilities, within a system which permits asynchronous processing, will exhibit the same properties as those encountered during this research.

In chapter six a mechanism is described which is designed to permit asynchronous processing within the RVM. Criteria for permitting two or more processes to execute asynchronously are first discussed, together with the constraints which must be imposed if the protection system defined by the RVM is not to be broken. From these discussions a solution to the problem is evolved which, it is claimed, will permit a sensible efficient use of asynchronous processing within the RVM. It is suggested that this mechanism will be sufficiently efficient to permit its use for each basic RVM operation, although in practice there will be many basic operations where this is not a requirement.

Finally, in chapter seven, the work undertaken in this thesis is related to other work being undertaken in the area of reliable computer systems. The relevance of the mechanism to permit asynchronous processing in the RVM is discussed in relation to the problems envisaged in the use of revocable capabilities. It is suggested that there will be a requirement for a similar mechanism when revocable capabilities are involved and that the mechanism could be extended to permit its use in this case.
A summary of the various areas left for further study, as a result of this research, is then given. Each area is briefly discussed and an indication given of the sort of results which might be expected in each case.

CHAPTER 2 - ASSOCIATED COMPUTER SYSTEMS

2.1 Introduction

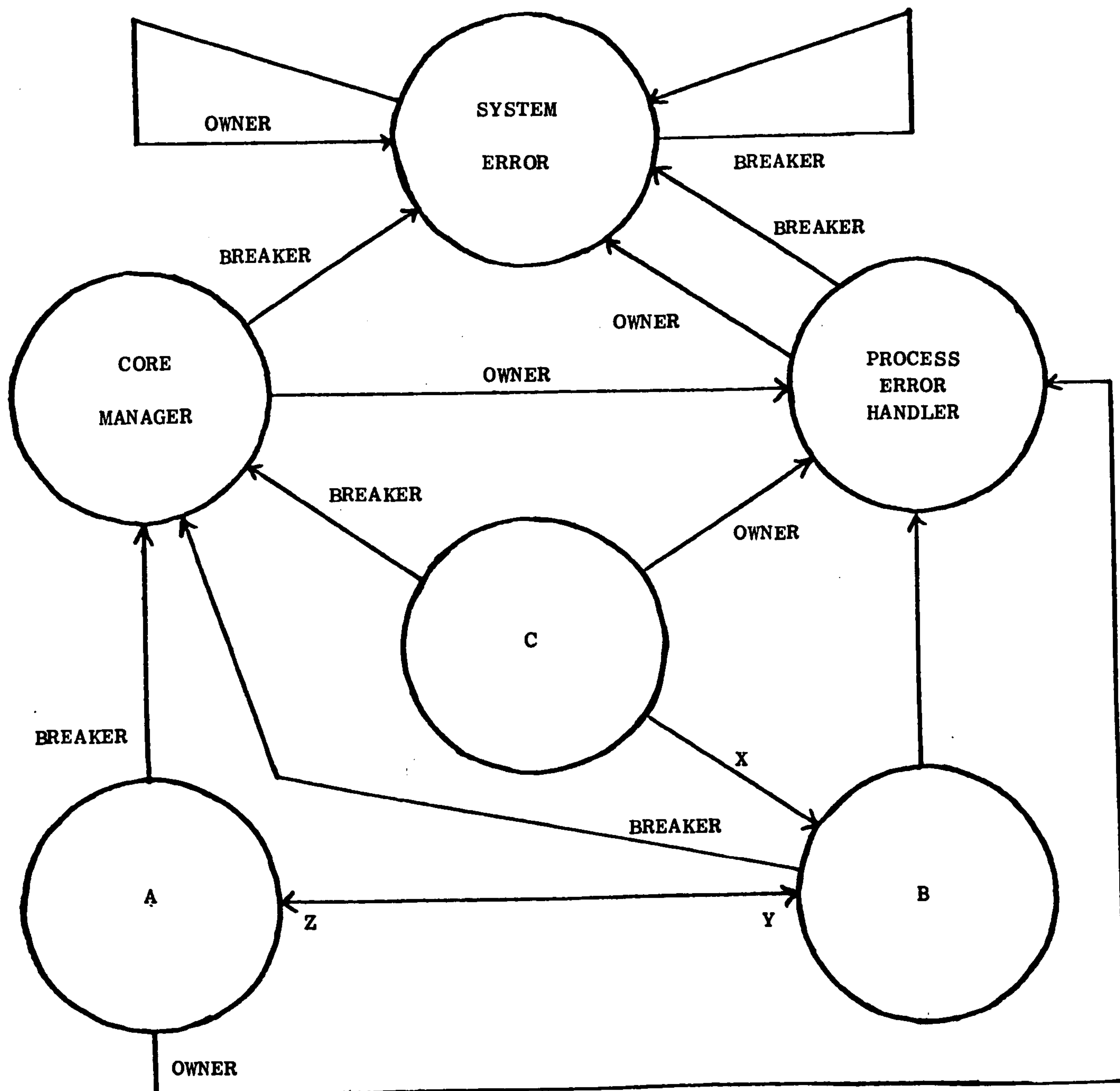
Many different computer systems have been proposed with the aims of security and reliability uppermost. These systems fall into two broad categories, those which utilize a hierarchical structure of protection and privilege and those which do not. However it is widely accepted that the best way to ensure a well protected computer system is to provide an environment structure in which processes can be contained.

In this chapter several of these computer systems are described and a close similarity between each approach is shown to exist. As a result it is proposed that any problem area in one approach may have a dual in the other, or alternatively that it may be possible to achieve a solution to the problem by closer examination of the alternative approach.

2.2 Non-Hierarchical Computer Systems

Most non-hierarchically structured systems which have been built, HYDRA [Wu 74, WLP 75], CAL-TSS [Lam 68, Lam 69b] and the GEC 4000 [GEC 74] for example, are designed around a capability mechanism which provides the basic protection mechanism for each environment in the system. The following sub-sections examine these systems in more detail.

GEC 4000 Structure



KEY: ⟶ indicates a route from one environment to another.

Segment Breaks force a message to be transmitted along an environment's 'BREAKER' route.

Other Errors are trapped by a message being transmitted along environment's 'OWNER' route.

Figure 2.1

2.2.1 The GEC 4000

An advantage of adopting the non-hierarchical approach to computer system design can be illustrated by examining the efficiency of the GEC 4000. Environments within the system ('Processes' in GEC terminology) are linked together by 'routes', see figure 2.1. In order to stimulate a process in another environment a 'message' is sent along the appropriate route. The presence of a route between two environments indicates that the capability to transfer messages between the two environments exists. For example, if a process in environment A attempts to access a segment that is currently on secondary storage then a message will be sent, on the BREAKER route, to the CORE MANAGER indicating that this segment is to be fetched into main store. As can be seen the distance between any two communicating environments is always 1 and thus the cost of switching between any two environments is always minimised.

A further point of efficiency, with regard to the GEC 4000, concerns the object naming mechanism used. Each environment in the system has, associated with it, a list of all the segments currently accessible to it, together with the type of access each is permitted. This list is termed a PAST. The PAST is maintained as a global table containing the real hardware address of each segment. Segments can be passed on to processes in other environments with messages, and it is always the global name that is stored in each environment's PAST. Because a global name space is adopted there is only ever a single mapping function, f, which needs to be applied in order to translate a name within a particular environment into a real object. The effect of this approach is that all environments exist at a single level of abstraction from the hardware of the machine.

The global name space approach, also adopted in MULTICS, has the disadvantage that new types of objects cannot be created in order to set up totally different abstract machines. Furthermore it is impossible to rename, or change the access to, a segment previously passed to a process in another environment.

2.2.2 HYDRA

The HYDRA system developed at Carnegie-Mellon University was designed to handle a multi-processor configuration of mini-computers. Although the HYDRA system itself is not structured hierarchically it will permit processes to be created hierarchically if so desired. The overall strategy is to permit processes to execute within a given protection environment, the privileges available to this environment being dependent upon the operations requested by any process within this environment. Thus for a process to perform a privileged operation it is not necessary for there to be a change in environment.

A 'kernel' mechanism is used to manage the physical resources of the hardware and provides, to programs executing on top of this, a wealth of abstract types of objects which can then be manipulated. Since a primary aim of HYDRA was to provide a secure multi-processor system, the kernel must possess a protection mechanism which prohibits its corruption by user programs.

The HYDRA system is fullyextendible, processes being permitted to define new environments, or capabilities in existing environments. The designers of the system felt it was important to keep separate the areas of policy and mechanism, and that kernel mechanisms should allow a program to implement any policy it may find appropriate to its needs [Le 75] . This separation of policy and mechanism particularly applies to the protection policy. The mechanism provided by the kernel in order to implement protection policies is the capability mechanism. Obviously

there is a consistent protection policy throughout the operating system, but it is possible for a user sub-system to be built alongside this with a totally different protection policy, errors in the sub-system policy being trapped before propogating to the operating system.

The protection mechanism as implemented in HYDRA was designed to have five properties. These are defined by Jones and Wulf [JW 74] as follows:-

i) The set of mechanisms should be sufficiently small that it is feasible to prove the properties which characterize the mechanism.

ii) The mechanism should be efficient, especially if dynamically invoked.

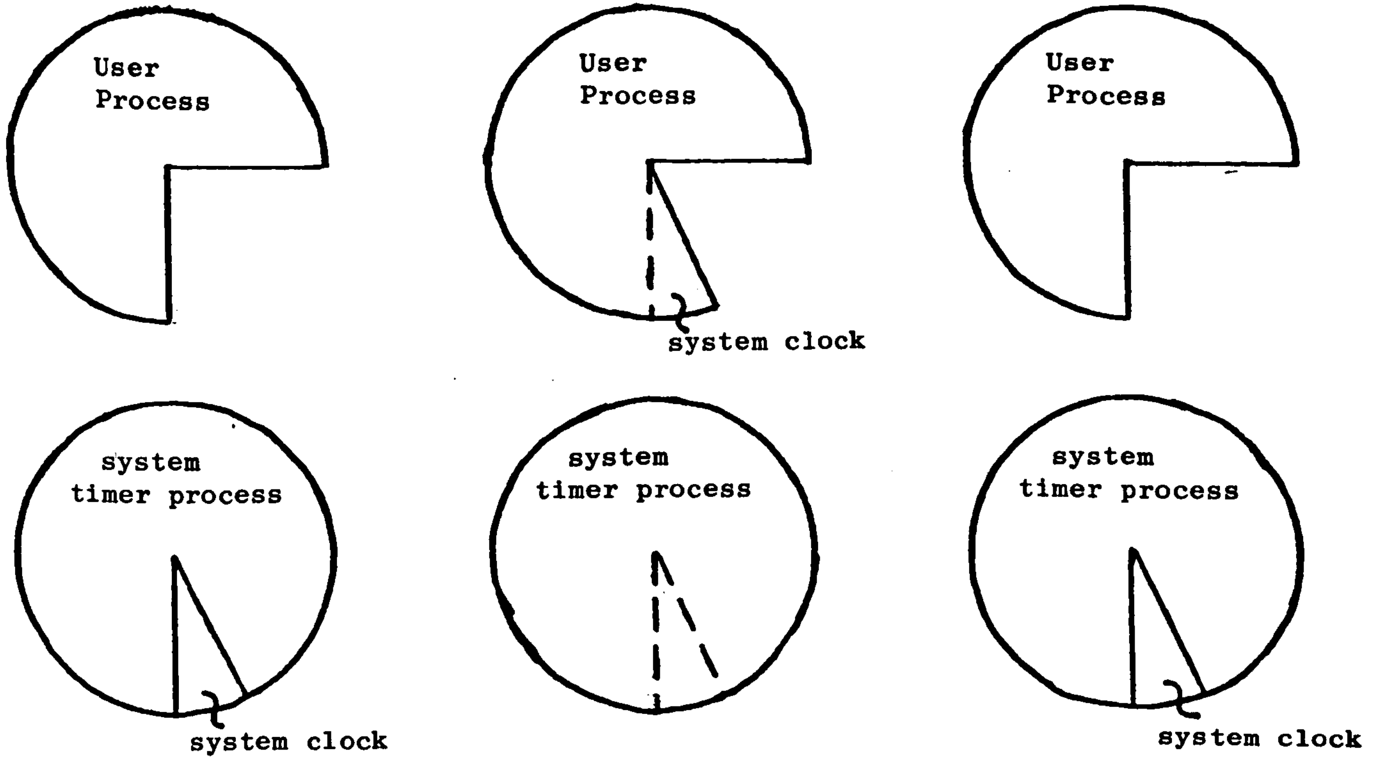
iii) The cost of its use should be linear in respect of the number of mappings performed from abstract object to real object.

iv) The mechanism should permit its users to express a policy in natural terms.

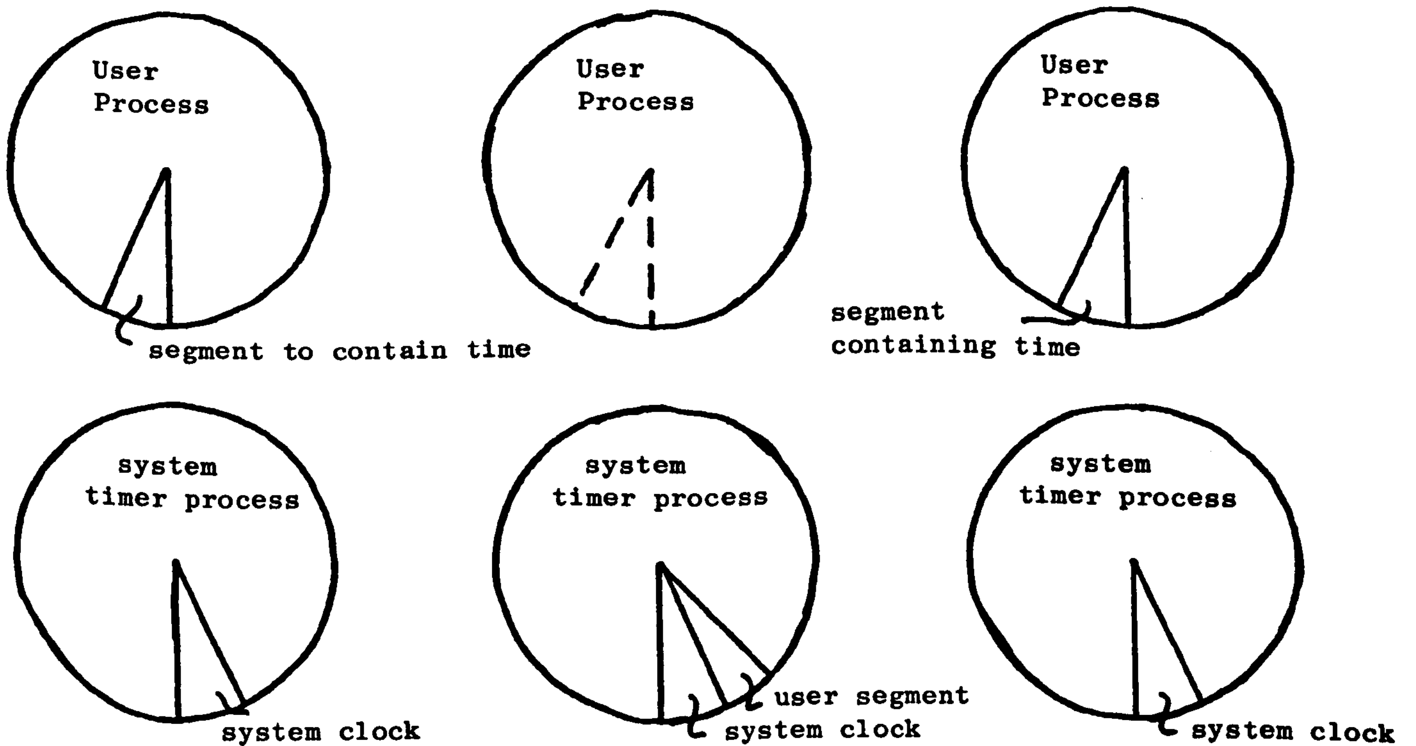
v) The mechanism should allow extension so that a user can create a new object type and use the mechanism for his own security policy enforcement as readily as the system policies use it for system objects.

In order to permit the fifth property the global name space approach to capability definitions, as discussed in 2.2.1, had to be modified to permit the creation and destruction of objects. Unfortunately the cost of the mechanism's use is not linear with respect to the distance between called and calling environments. In particular there is a major problem of efficiency when 'calling' a kernel operation, especially

Example of HYDRA 'Rights Amplification'



a) HYDRA Scheme



b) Environment transfer scheme

Figure 2.2

from environments at a level of abstraction greater than one. This lack of efficiency, coupled with the ease with which a capability mechanism can provide a security policy for certain classes of protection problem, led to a set of policies being devised which avoid the use of a strictly hierarchical environment structure.

The HYDRA designers believe that, ideally, an execution environment should contain the minimum set of rights needed to perform the task, or tasks, executed within that environment since each unnecessary right provides an additional opportunity for errors to have a 'worse' consequence. Thus a process executing within a non-privileged environment has to request 'rights amplification' for some of its objects when it is necessary to call a procedure which requires these extra privileges[Jo 73]. A result of this is that on entering a particular execution environment, a privileged process is permitted to amplify the rights for the objects to which it requires access.

Example:

Referring to figure 2.2; a system clock may be maintained by a system process in a fixed segment of the system. A user process which wishes to read the time of day must call the privileged 'time' process to request 'rights amplification' to the clock segment while the time is noted. A more usual approach would be to transfer control to the system process, determine the time, and return to the user process.

By implementing improvements and extensions to the original capability mechanisms many interesting protection problems have been solved in HYDRA. The 'rights amplification' problem is one example of a difficulty encountered by not adopting a hierarchical environment structure. Other examples of problem areas for which solutions have been found in HYDRA but would cause little difficulty in a hierarchically structured machine are 'monitoring' and 'confinement'.

The problem of monitoring requires a parent process always to be able to monitor its descendant processes, and in a hierarchical structure this should always be the case since the hierarchy will be one of protection and privilege.

The problem of confinement is concerned that the contents of an object should not leak out of a particular environment. This is a serious matter in HYDRA since a process can always pass on one of its capabilities to a process in another environment, However in a hierarchically structured system, the right to access an object can always only be passed to descendant environments and it is then impossible to pass this object from that environment without notifying the parent process.

2.2.3 CAL-TSS

CAL-TSS in an operating system designed for a CDC 6400 at the University of California at Berkeley. The major aims of the system were to provide a general purpose operating system, capable of supporting both batch and time-sharing operations which would be competitive in performance with the manufacturer's operating system. Further goals for the properties of the system, as seen by the sophisticated programmer, were to provide a protection system based on capabilities and to construct the system as a sequence of layers, each protected from those at a more abstract level. Users were to be permitted to add layers in the same way, intercepting and handling exceptions without incurring any overheads during 'normal' sequential processing. An implication of this was that users had to be able to create new types of objects.

The generalized structure of a capability system and the ease with which capabilities may be passed between environments has permitted a non-hierarchical system to be built. As in the HYDRA system a major reason for this approach is the inefficiency involved in calling 'kernel' operations. Lampson and Sturgis [LS 76], in their discussion of the CAL-TSS implementation, explain that this inefficiency is primarily caused by the fact that the kernel mechanisms are all supplied by software.

One major failing of CAL-TSS, the GEC 4000, and early HYDRA systems was the inability of a parent process to retrieve capabilities passed to processes in more abstract environments until such descendants finally return them. In this way it is possible for malicious processes to

Mapping of capabilities within CAL-TSS

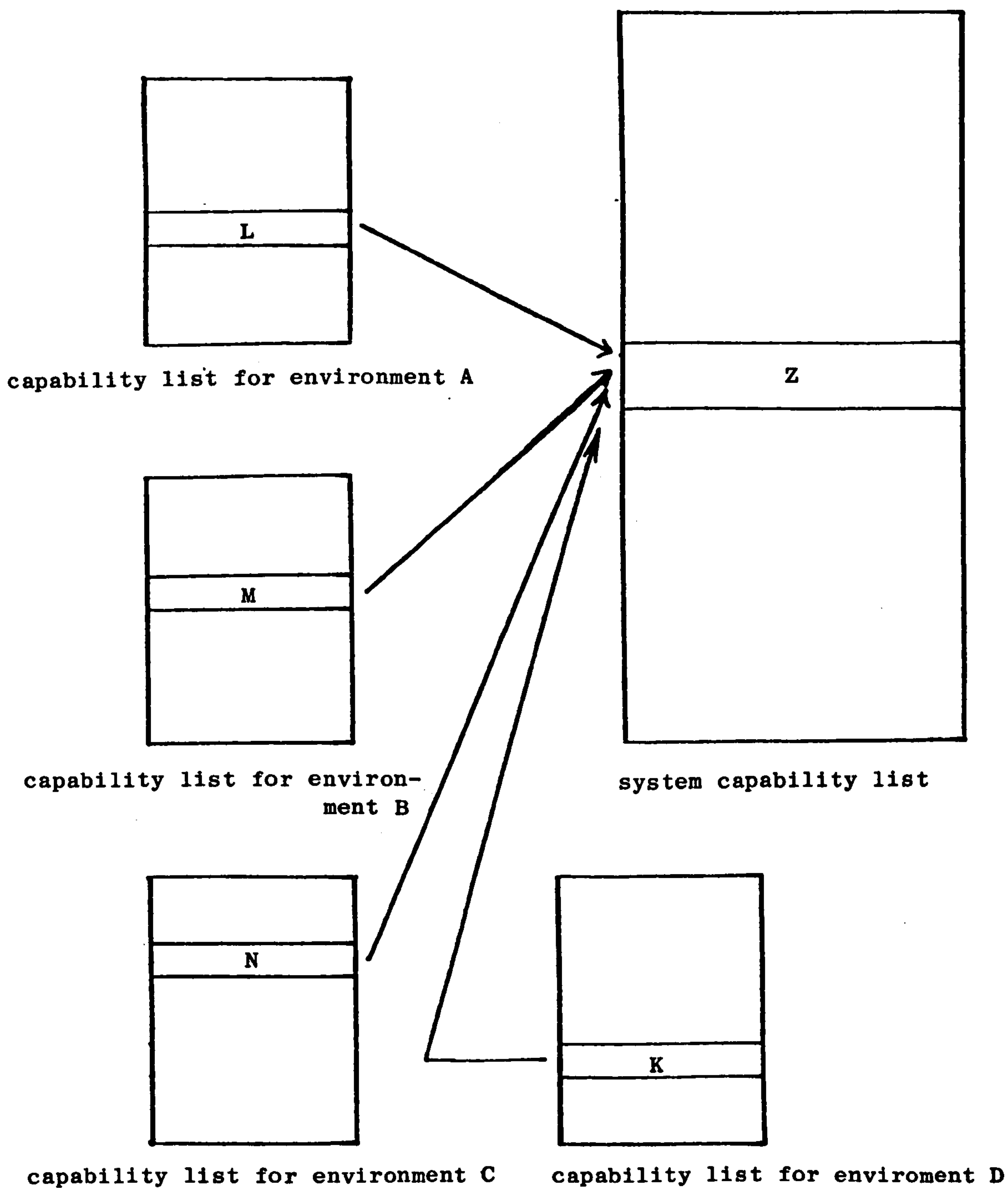


Figure 2.3

request capabilities for certain critical objects and once having received them to refuse to return them to their owner. In CAL-TSS the possibility of retrieving the capability, 'revocation' [RF 74], is dismissed since it is quite feasible that it has been passed to another environment of which the owner of the capability is unaware.

Example:

Figure 2.3 describes the structure of capabilities within CAL-TSS. A process within environment A has access to an object L within its terms, this is in fact the system object Z. A process within environment D also has access to the same system object Z, called K in its terms. The process in A may pass the capability on to a process in B which may in turn pass the capability on to a process in C. The process within A cannot now revoke access to Z for B since this implies revocation of Z in C, and A has no knowledge that C has access to Z. Furthermore the system capability list cannot be manipulated to delete Z as D still requires access to it.

This problem is partly caused because a global table is used to hold the names of all objects currently in the system. The reason for this approach is the fact that all abstract objects exist at the first level of abstraction of the system and thus efficiency is improved. In both the CAL-TSS and HYDRA system it is understood that if the cost of providing a multi-level mapping function could be made linear with respect to the level of abstraction then a hierarchical structuring of environments would be both feasible and desirable.

2.3 Hierarchical Computer Systems

As a natural progression from the level of abstraction methodology, discussed in the previous chapter, associated with the techniques of structured programming, developments have taken place which attempt to provide strictly hierarchical computer systems. Most of these developments follow the Virtual Machine approach, VM 370 [MS 70, Pa 72], the 'Hardware Virtualizer' [Go 73], and the 'Recursive Virtual Machine', RVM [LW 73, Lau 74]. An exception to this is the CAP computer [Wa 73, NW 74] which bases its protection scheme on a capability mechanism.

2.3.1 VM 370

Virtual machine systems have, in the majority of cases, only reached the design phase. In fact, as far as can be determined, the only implementation of a virtual machine system which permits repeated levels of virtualisation is the IBM CP/67, or VM 370, system.

CP/67, or VM 370 as it is now known, is the realization of a virtual machine system on the IBM 360/67 and IBM 370 series computers. As mentioned in chapter one these computers are conventional two-state machines and any use of a privileged instruction causes an automatic trap to the lowest software level. Such a scheme causes gross inefficiences when attempting to execute programs in environments at a distance greater than one from the bare machine.

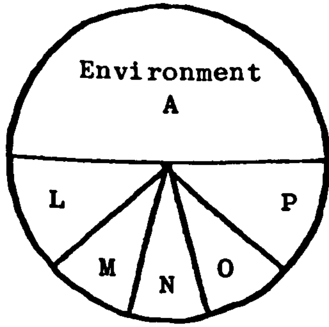
The security of the VM 370 control program has been rigorously checked by Belady and Weissman [BW 74], who undertook a series of experiments which were designed to break the protection mechanisms of this system. The results of these experiments showed that nearly all the design errors in this control program were in the area of software design, and that most of these errors could only be exploited by asynchronously using the data channel programs which were requested for programs executing in a virtual machine environment. Moreover, it was not possible for a user to break out of its current virtual machine environment without exploiting some of these asynchronous processing facilities.

The failure in the security of VM 370, when performing asynchronous I/O operations, primarily occurs because the underlying machine architecture is unsuitable for the software structure being imposed upon it. The

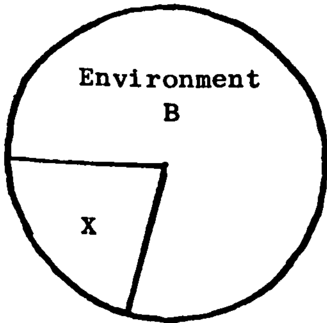
security breach in VM 370 occurs because a process requiring input or output to be performed must pass control to the control process at the lowest level of the software hierarchy, possibly by passing some intermediate environment. The control program is then able to schedule some other process while the IO is being performed, and this may be an ancestor of the original IO requesting process. This rescheduled process may thus have access to the data channel program being utilized by the control program and it may overwrite it either maliciously or accidentally, as a result this may cause the virtual memory containing the control program to be passed to another process.

The main point, illustrated by VM 370, is the unsuitability of a conventional architecture for providing a hierarchically structured multi-level computer system. However, of perhaps more importance to this thesis is the fact that by building such a system an extremely high degree of security has been obtained. The only real breach occuring in the area of asynchrony.

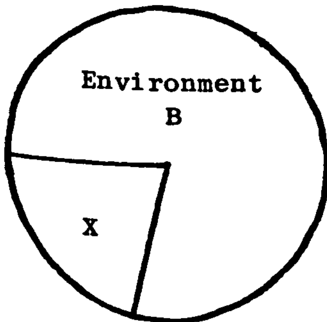
It is thus postulated that if a mechanism can be devised which will permit asynchronous processing, without causing any loss of efficiency or overall security, and this is implemented in a multi-level virtual machine system, then the resulting system will be viable in terms of efficiency, computing power and the protection mechanisms allowed.



Environment A contains objects L,M,N,O,P creates object A from L and M, passes X to environment B



Environment B contains object X, passes X to environment C



Environment C contains object X

Figure 2.4

2.3.2 The 'Hardware Virtualizer'

One proposal for a hierarchical multi-level computer system is the 'Hardware Virtualizer' described by Goldberg [Go 73]. This is a formal discussion of a virtual machine system. The use of an f-map, which provides a mapping of names of objects in the system, and a ϕ-map, which performs the mapping of these various objects, is described. These functions are both provided by a segment table mechanism in the RVM, individual entries performing the ϕ-map, and the position of each entry within a segment table providing the f-map.

Within the Hardware Virtualizer it is proposed that exceptions occurring in one virtual machine should be trapped by the virtual machine which has created the object requested.

Example:
Referring to fig 2.4. A process within environment A creates an object X from its available resources and passes it to the environment B. A process within environment B passes X to the environment C. If a process in B or C attempts to access X then an exception will occur and the process will be trapped in environment A.

Goldberg's study of the Hardware Virtualizer has led to the implementation of a Virtual Machine Monitor on a Honeywell series 60 level 68 computer [GS 76]. In order to effect this implementation minor architectural changes were made to the processor and other system components. The designers of this system comment that, in order to produce efficient virtual machines, it is necessary to avoid a significant amount of virtual

machine monitor software interference. The results of Goldberg's work have led to a system being developed which supports both Honeywell 6000 or Honeywell level 68 virtual machines. It permits the running of multiple distinct copies of the MULTICS and GCOS operating systems on a single hardware system.

This implementation demonstrates, to great effect, the feasability of virtual machine systems. The performance on a virtual machine approaches that of the real machine, throughput is high and the overheads are low. In contrast to the VM 370 system each user need not have his own virtual machine, interactive facilities can be provided by either GCOS-TSS or MULTICS running on a single virtual machine, and from this virtual machine multi-access facilities can be provided for a number of users. Unfortunately discussion of the possibilities of recursive use of this virtual machine monitor system is non-existant, and it is presumed that the approach adopted is non-hierarchical, all virtual machines existing at a single level of abstraction from the hardware.

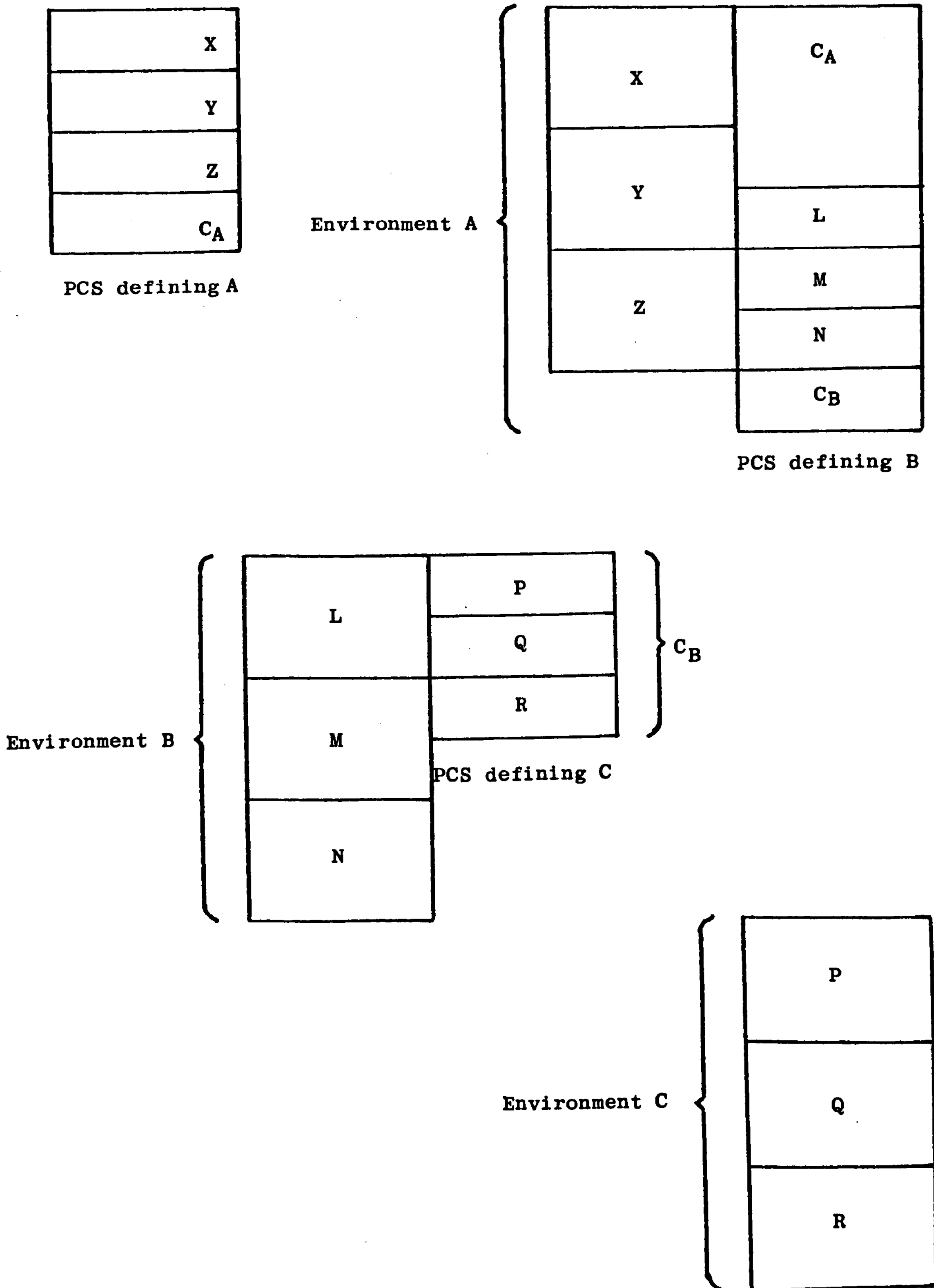
2.3.3 The 'Virtual Machine Monitor'

Another proposal for a virtual machine system is presented by Belpaire and Hsu [BH 75]. This paper proposes a design based upon a 'Virtual Machine Monitor' which is permitted access to a stack of resource mapping resisters. The purpose of these registers is to permit the hardware to access directly the memory locations where the mapping of virtual objects into real objects is actually contained. The RVM also provides these facilities through its 'display' and 'segment table' mechanism, as does the Hardware Virtualizer's ϕ-map and f-map mechanism.

Belpaire and Hsu claim that their approach is more general than either of these two other schemes, due to the resource mapping registers being part of the hardware thus leaving the resource allocation policies to the operating system designers.

The RVM, Hardware Virtualizer and Virtual Machine Monitor all base their resource allocation and protection mechanisms on a hierarchically structured resource mapping strategy. Although the implementation of each scheme may be realized differently all three are based on essentially the same fundamental concept, that of mapping objects in one environment into other objects in the next more abstract environment. Also in all cases ancestor environments have complete power to revoke objects from any descendant environments. This essential similarity between these three systems leads to the conjecture that any fundamental problem area in any one system will have its dual in each of the other systems.

The CAP System Structure



Note that all PCSs are in special capability segments

Figure 2.5

2.3.4 CAP

The CAP computer, built at Cambridge University, has been designed as a hierarchical structure of protection environments, and a capability based mechanism is used to provide the overall security. Two separate protection schemes have been implemented within the overall structure of the CAP computer, one is designed to protect processes executing in different environments and the other is designed to protect processes executing within the same environment.

The inter-environment protection scheme is strictly hierarchical in nature and primarily concerned with storage protection. In this respect CAP closely resembles a virtual machine system, and indeed the capabilities it uses are very different from those used conventionally. The CAP system, see figure 2.5, defines an environment by a set of 'indirectories' which are mapped through a 'process capability segment' (PCS). The PCS defines a list of capabilities for all objects that a process within this environment has permission to access. It is only by specifying the capability for an object, defined in the PCS, that a process may access this object.

A closer examination of the CAP capability structure reveals an extremely close resemblance to virtual machine systems. A capability is specified as part of an address and this is held as an index to an indirectory for this PCS. Furthermore the protection mechanism groups objects into segments, each capability specifying the permitted access to a set of contiguous memory locations consisting of the named object.

Capability lists are grouped to form capability segments, such segments being a different type from ordinary memory segments, and objects of the capability segment type are provided with special hardware features to prevent them from being overwritten in error.

Walker, in his thesis [Wa 73], mentions the reason for the CAP decision to separate capability segments from memory segments. This was made, not from any theoretical reason, but primarily to ease the overall implementation of the system and to prevent descendant environments from corrupting their own capability lists. Thus there appear close parallels between this system and other virtual machine systems.

The reasons why a dual protection scheme has been implemented in CAP, are particularly pertinent to this thesis. Within the CAP system a process is not permitted to migrate into another environment if it is at a greater distance than 1. This introduces high overheads when it is actually required to migrate to an environment at a greater distance since the intermediate environments at the intervening levels of abstraction must be entered and left before the reaching of the required goal. These high overheads, some indeterminate function of the distance between the two environments, led to the introduction of a within environment protection scheme. This within environment protection scheme is designed to permit privileged processes to execute within a less privileged protection environment, without a less privileged process within that same environment being able to break the overall security of the system.

In practice the CAP system only utilizes the two lowest levels. The high cost of maintaining environments at higher levels of abstraction proved prohibitive, thus re-inforcing the designers beliefs that both protection mechanisms were needed. It is still possible to set up a hierarchy of environments as originally intended but the resulting system makes little use of this feature.

2.4 Protection and Addressing Systems

The previous sections have illustrated that there are many similarities between protection systems, eg. CAP, HYDRA, CAL-TSS, and addressing systems, eg. RVM, VM 370, Hardware Virtualizer. Wyeth [Wy 76] in his thesis, discusses several protection systems and also illustrates this duality of the two approaches to providing a secure computer system.

Both protection and addressing systems aim to protect processes from either malicious or inadvertent damage by other processes. An addressing scheme achieves its aim by the mapping of core addresses into segments or pages and then permits a particular process to access only those parts of the total machine resources which exist in its own address space. Processes can, of course, communicate by being given access to a common resource, and this resource may be addressed differently by each process. In such cases the responsibility for the security of the interacting processes lies with the processes themselves, and they must ensure that sensible use is made of the common resource.

Protection systems, of which the capability systems are the most elegant, define a protection environment in which a process may execute. Each attempt to access an object is made by profferring this object's name, the protection system may then permit or reject the access.

Protection and Addressing systems both permit the dynamic reallocation of objects between processes, even mapping the same objects into different real resources of the machine hardware.
Of course, as the inter-environment protection scheme of CAP illustrates, a protection system can be implemented by the use of an addressing mechanism.

2.5 Conclusions

It is interesting to note that all of the real systems which have been discussed, with the exception of VM 370, do not fully utilize the hierarchical environment structure which all, except for the GEC 4000, provide. In all cases the argument against using a hierarchical structure appears to be one of efficiency, and even users of VM 370 will admit that it is impractical to provide virtual machines at high levels of abstraction.

In the case of CAL-TSS and VM 370, the reason for this inefficiency is the unsuitability of the underlying computer hardware. In CAP an impractical means of transferring control from an abstract environment to an ancestor has been implemented, and in HYDRA the hierarchical structure suffered because of the extra emphasis placed on the non-hierarchical structuring of environments.

Since the essential similarity between the various approaches to providing a secure and reliable computer system is so apparent, protection systems must be re-examined to investigate whether the problem of providing asynchrony in the RVM can be solved. Alternatively, it is postulated, some weakness may be exposed in the design of these different systems such that further mechanisms will have to be employed in order that their overall security is maintained.

Chapter 3 - DESIGN OF THE RVM

3.1 Introduction

The motivation of adopting the virtual machine approach to providing a well protected computer system has been discussed in chapter one of this thesis. Once such a proposal has been adopted it remains to define the features that are considered desirable in such a system and these are summarized as follows.

i) There should be no supervisor state, instead a hierarchical structuring of protection environments is to be implemented, with each environment the responsibility of its immediate ancestor.

ii) The system should be reliable. In particular a fault occuring in any environment should not cause the propagation of any fault beyond the ancestor environment which has permitted this fault to occur. In this way sections of a system could be tested while the production system is executing.

iii) The system should provide a protection mechanism which can easily be utilized by any process executing within the system. It is this mechanism which, it is to be hoped, will permit protection policies to be developed which will increase the overall reliability of the system.

iv) Objects within the system should be renameable, thus permitting the system to be extended.

v) The system should be 'efficient', with at worst a linear cost function when executing at different levels of abstraction.

As explained in chapter two, although several proposals for a similar system have been made, the only hierarchical virtual machine system known to exist is VM370. This system is a realization of a virtual machine system on conventional two-state architecture and consequently exhibits gross inefficiencies when processes execute at high levels of abstraction. By implementing the RVM it is hoped to prove that an efficient virtual machine system can be built such that processes aa high level of abstraction can gain a significant amount of processor utilization.

In this chapter the design of the recursive virtual machine architecture chosen will be discussed. The above features of the system are examined and an explanation given as to how they are to be provided.

Throughout this chapter it will be assumed that only one processor is present in the system, and that no 'time-sharing' facilities are available. The introduction of asynchrony causes the, non-trivial, problems hinted at in the previous chapters and will be discussed at greater length in subsequent chapters.

3.2 The Virtual Memory Addressing Mechanism

Fundamental to the RVM design is the virtual memory mechanism which has been adopted. This forms the heart of the protection mechanism, each environment having access to only those objects within its virtual memory.

3.2.1 The RVM Virtual Memory

A virtual memory is defined as the set of objects which constitute all the resources of its virtual machine. It should be noted that this usage of virtual memory is an extension of that conventionally taken. In the case of the RVM 'memory' is that resource conventionally termed 'virtual memory', whereas 'virtual memory' in the RVM also includes any other resources available to a particular virtual machine. So at the lowest, 'hardware', level the RVM's virtual memory equals the total memory available to the hardware, no matter on what media it is stored, plus all the peripheral resources attached to the hardware configuration. A 'virtual memory' at some more abstract level will then be constructed from a subset of the base machine's resources plus some 'virtual' resources provided by any of its ancestor virtual machines.

The representation of a virtual memory within the RVM is as a collection of segments, each of which is conceptually a sequence of bits; although these bits need not necessarily map into the core store of the hardware, for a peripheral device for example. An attribute of each virtual memory is the set of possible names of its segments, and these are represented as non-negative inters up to some reasonable maximum. Associated with each virtual memory is a mapping function which specifies, among other things, which segments are identified with which segment names.

3.2.2 RVM Segment Tables

The mechanism which provides the mapping function between virtual memories is called a segment table and defines a particular virtual memory under discussion. Any particular segment may be identified with one or more names, but each may be identified with at most one segment. If two virtual memories have segments in common, it is immaterial whether those segments have either the same or different names in each virtual memory. In this way objects in the system can be renamed.

A virtual memory is, in fact, either equal to the hardware memory, which includes all the objects available to the hardware, total memory and peripherals etc., or it is a subset of another virtual memory. Thus the notion of virtual memory is made recursive. In the first case the segment table of a virtual memory is implicit in the hardware memory accessing circuitry, and in the second case, its segment table specifies the name and part of the containing segment in the containing virtual memory. Figure 3.1 illustrates the required format of a segment table entry. It specifies the size of the segment, the access permitted to the segment, the type of the segment and the base address of the segment; this last item is expressed as an offset within a segment of the environment which is setting up the new subordinate environment.

Format of a Segment Table Entry

ACCESS	TYPE	SIZE	BASE ADDRESS	
			CON-TAINING SEGMENT	OFFSET

Figure 3.1

A Hierarchy of Virtual Machines

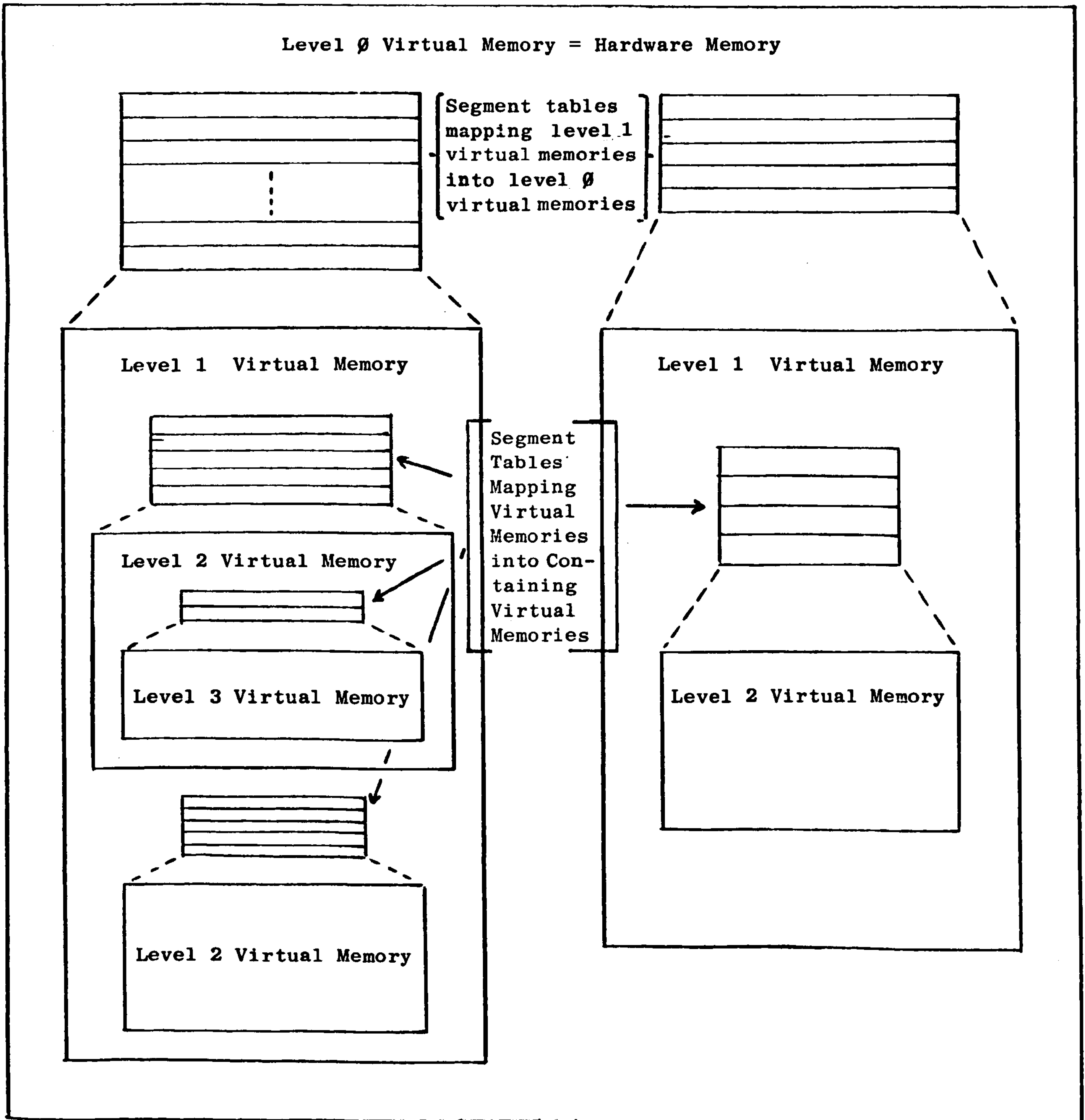


Figure 3.2

3.2.3 The RVM Hierarchy of Virtual Machines

These segment tables, which define a particular virtual memory, are located within the containing virtual memory, and the set of virtual memories in a system at any instant forms a hierarchical structure based upon this containment relationship. Since each segment table contains only names and quantities known within its own environment, processes within that environment can freely operate upon it, in this way managing the resources of their subordinate environments. This hierarchy of environments is best illustrated by an example and this is pictured in figure 3.2.

All objects within a particular virtual memory are identified by their virtual memory address, there being no specific reference to any actual addresses defined by the hardware. Each object is named by a pair, consisting of segment name and offset within this segment, and this is then mapped by the hardware into a physical address in the real machine.

The Address Mapping Mechanism

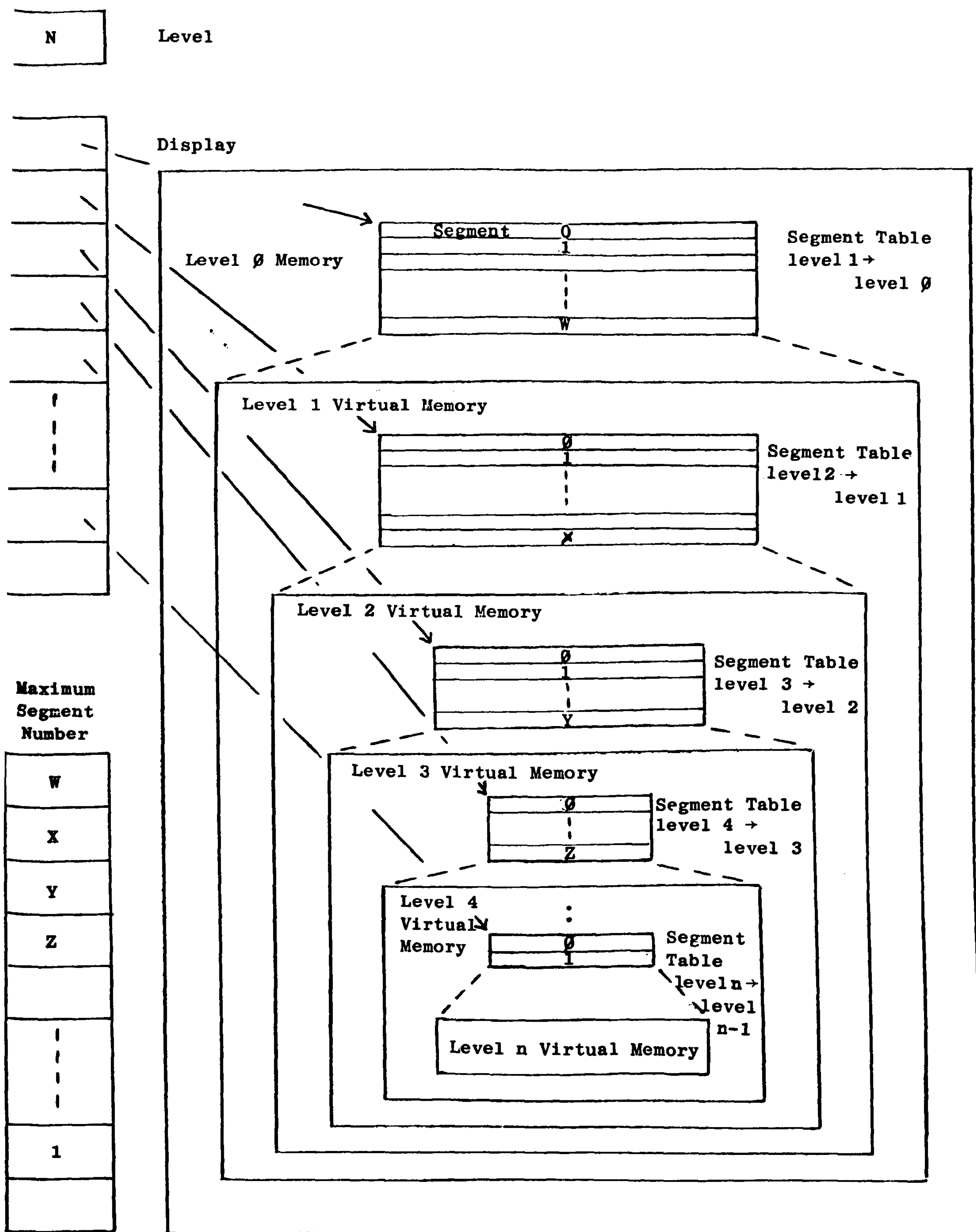


Figure 3.3

3.2.4 RVM Hardware Addressing Mechanisms

In order to accomplish this address mapping efficiently, the following mechanisms are used. These are illustrated in figure 3.3.

Within the processor is a small array of registers called the 'display', and a single register named 'level'. These registers are an integral part of the hardware and are not accessible to any program within the system. They are automatically maintained by the hardware as side effects of performing certain machine operations, such as those to transfer control to a new protection environment. The register 'level' contains, at all times, the degree of containment of the currently executing process, the hardware being regarded as level zero.

The elements of display contain the address of the segment tables defining the current hierarchy of execution environments. These are numbered from 1 to n, the segment table defining the level zero environment not existing, as this environment is defined implicitly by the hardware configuration. Such a mechanism clearly provides the hardware with the ability to translate addresses in one environment into the absolute addresses of the real machine.

In accordance with our definition of efficiency it is essential to ensure that the number of memory accesses for each call of the address translation algorithm is minimised. To perform the address mapping at least one segment table entry must be accessed for each currently active level in the system. Since the number of levels in this system is linear it is postulated that the address mapping mechanism is efficient. (Later, techniques for improving upon this will be discussed.) It should be noted that if the address held in the display are stored as the relative addresses of the segment tables within the current environment then the number of segment

table accesses for each call of the address translation algorithm will be exponential. As a result, it is proposed to store in the display the actual hardware addresses of each segment table in the currently active system.

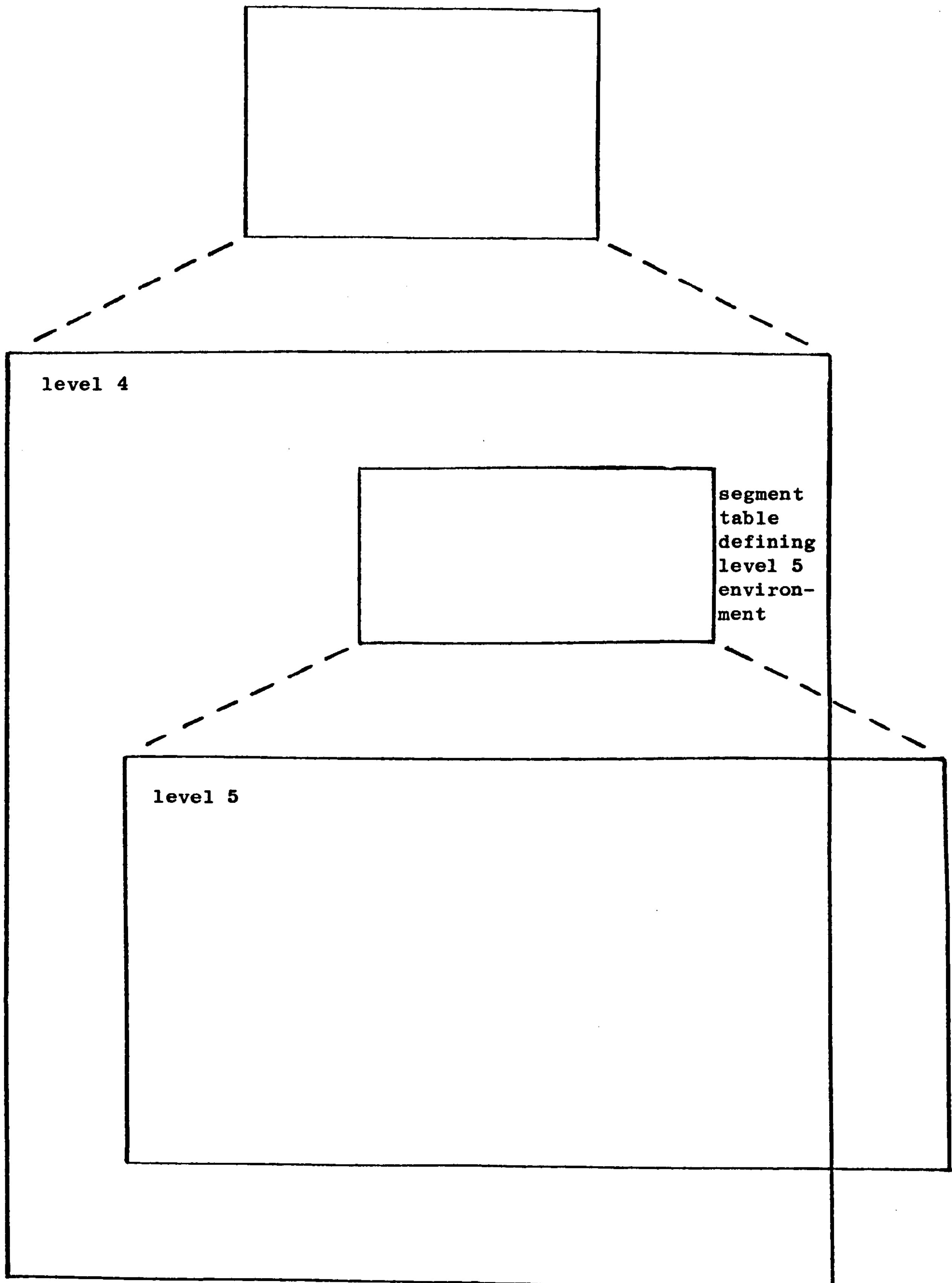


Figure 3.4

3.2.5 The Address Translation Algorithm

The address translation algorithm is now relatively straightforward. By addressing the appropriate entry in each segment table of the hierarchy, it is possible to impose checks for access, type and bounds. Any violation in one of these checks will result in a fault being signalled at the level of the environment which owns the segment table entry currently being examined. This ties in with the philosophy of preventing errors from propagating into less abstract environment from which they originated.

Example:

Referring to figure 3.4, a process executing within an environment at level 5 may attempt to access a location defined by its segment table in the corresponding level 4 environment. However, this object might not exist in the segment table defining this level 4 environment and thus a violation will occur when examing the segment table entry at level 3. The level 3 environment is therefore the most abstract environment at which this fault can be detected and this is where control is passed.

One further check when performing the address translation, is the validity of the segment name proffered at each level of the address mapping. This is performed by means of a second array of registers, the same size as the display, each item containing the value of the highest numbered segment at each level, names of segments being defined as non-negative integers. The three sets of registers, the 'display', 'level' and 'maximum segment number' constitute the total requirements for performing the address translation through the segment tables. The algorithm can be summarized by the following 'pseudo-algol' procedure.

```
procedure hardware address (segment, position)
        returns  (physical segment, physical position);
begin
        integer i;
        for i := level, i-1 while i > Ø do
        begin
              if segment ≥ maximum segment number (i) or
                 position > size (display (i)+ segment) or
                 violation of type (display (i) + segment) or
                 violation of access (display (i) + segment) then
                 signal fault at level (i, fault code);
              position := position + offset (display (i) + segment);
              segment := containing segment (display (i) + segment)
        end;
        return (segment, position)
end hardware address;
```

A careful study of this algorithm shows that it should be possible to perform address translations linearly with respect to the number of levels of abstraction, and this algorithm fulfils the efficiency requirement. Moreover, by the use of a proven mechanism such as a small fast look aside memory which contains the physical address of the most recently used items, it is anticipated that a processor adopting this mechanism would operate at nearly the same speed at all levels in a virtual memory hierarchy. Such a mechanism has been implemented on the IBM 370/168 in order to improve the performance of VM370.

3.3 Environment Crossing

Having considered how addresses of objects in one environment can be mapped into objects of the real machine, the question of a process, executing at some arbitrary level of abstraction, setting up an environment at the next, more abstract, level must be considered. For in this way virtual machines are permitted to be recursively defined on top of each other.

3.3.1 Creating New Virtual Machines

In order to create a new virtual machine, the parent process must construct a segment table which specifies the virtual memory of the environment it wishes to create. This virtual memory will define all the objects available to the virtual machine being created. The parent process is then able to initialize this virtual memory as required since all the objects are a subset of the parent process's virtual memory. No special privilege or knowledge of the physical resources of the hardware is required for this purpose since the only names in the segment table are the names by which its own objects are known. Once the next level environment has been set up it requires the processor to provide a (non-privileged) operation which will update level, display and maximum segment number correctly and then perform the transfer.

Such an operation is described by the following procedure, which accepts two parameters, the address of the segment table in the terms of the current environment and the number of segments in that table. It is assumed that the hardware of the RVM, at the lowest level in the hierarchy will need to access a number of registers, the current next instruction pointer for example, for any process executing in a given environment. For this reason procedures are referred to which save and reload these registers.

```
procedure transfer to son (segment table address, segment table length);
begin
      store hardware registers in current virtual memory;
      display (level + 1) := hardware address (segment table address);
      maximum segment number (level + 1) := segment table length;
      level := level +1;
      load hardware registers from new virtual memory
end transfer to son;
```

3.3.2 Returning to a Parent Virtual Machine

It has already been noted that should a protection violation of any kind occur while performing an address mapping, then control is passed directly to the environment which is capable of providing the object to which access has just been denied. An operation to return control to a parent process can thus be provided by performing a controlled 'signal fault at level' operation, and this can be defined by the following two procedures. (Note that an attempt to perform such an operation from a process within an environment at the least abstract level will cause the hardware machine to halt).

```
procedure return to father;
begin
      signal fault at level (level -1, return code)
end return to father;

procedure signal fault at level (i, code);
begin
      store hardware registers in current virtual memory;
      if i < Ø then halt
      else
      begin
            level := i;
            load hardware registers from new virtual memory;
            store fault code in current virtual memory (code)
      end
end signal fault at level;
```

These operations will maintain consistently the values of level, display and maximum segment number which reflect the level of nesting of virtual memories in the current system.

3.3.3 Transfer Back to a More Abstract Virtual Machine

In a hierarchical structuring of virtual machines, processes within one execution environment act as a supervisor for the next more abstract one, and it would be possible to build an operating system using the architecture so far described. However, in order to achieve an efficient system, it is a requirement that processes pass control directly back to calling processes in environments at more abstract levels. That is to say that an inverse operation to 'signal fault at level' is required. Clearly such a feature is essential from an efficiency point of view, since the most likely reason for returning to a supervisory process is to request some object currently unavailable in the calling process's environment.

The automatic transfer between environments in the system is an important feature of a hierarchical system such as this. For, as mentioned in the previous chapter, it is in the area of context switching that other similar systems have fallen down, especially with regard to the overall efficiency, Therefore, it is proposed that an automatic mechanism must be provided which permits the transfer of control back to a calling environment as well as to a process within a supervisory environment.

Lauer and Wyeth, in their definitive paper on the RVM architecture [LW 73], describe a mechanism which permits control to be passed automatically to a descendant environment. Their scheme relies on a set of pointers being maintained within each virtual machine which provides information to the hardware of the next, more abstract environment, see figure 3.5. This approach will fail if, as a result of being called by a process in a more abstract environment, the supervisory process decides to set up a new descendant environment to perform the requested operation, thus overwriting the pointers defining the called process's environment.

Original Automatic Return Mechanism

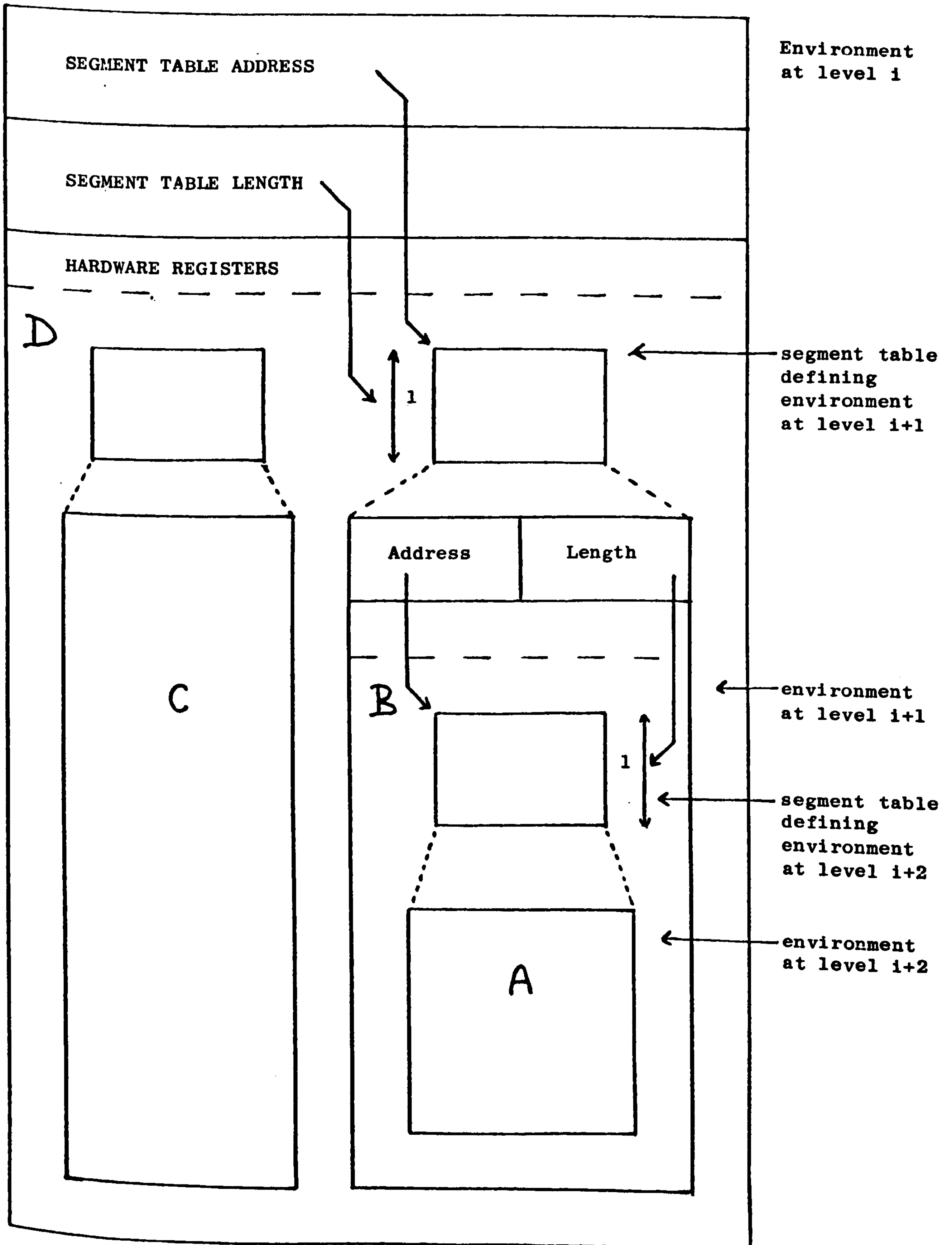


Figure 3.5

Automatic Return Mechanism

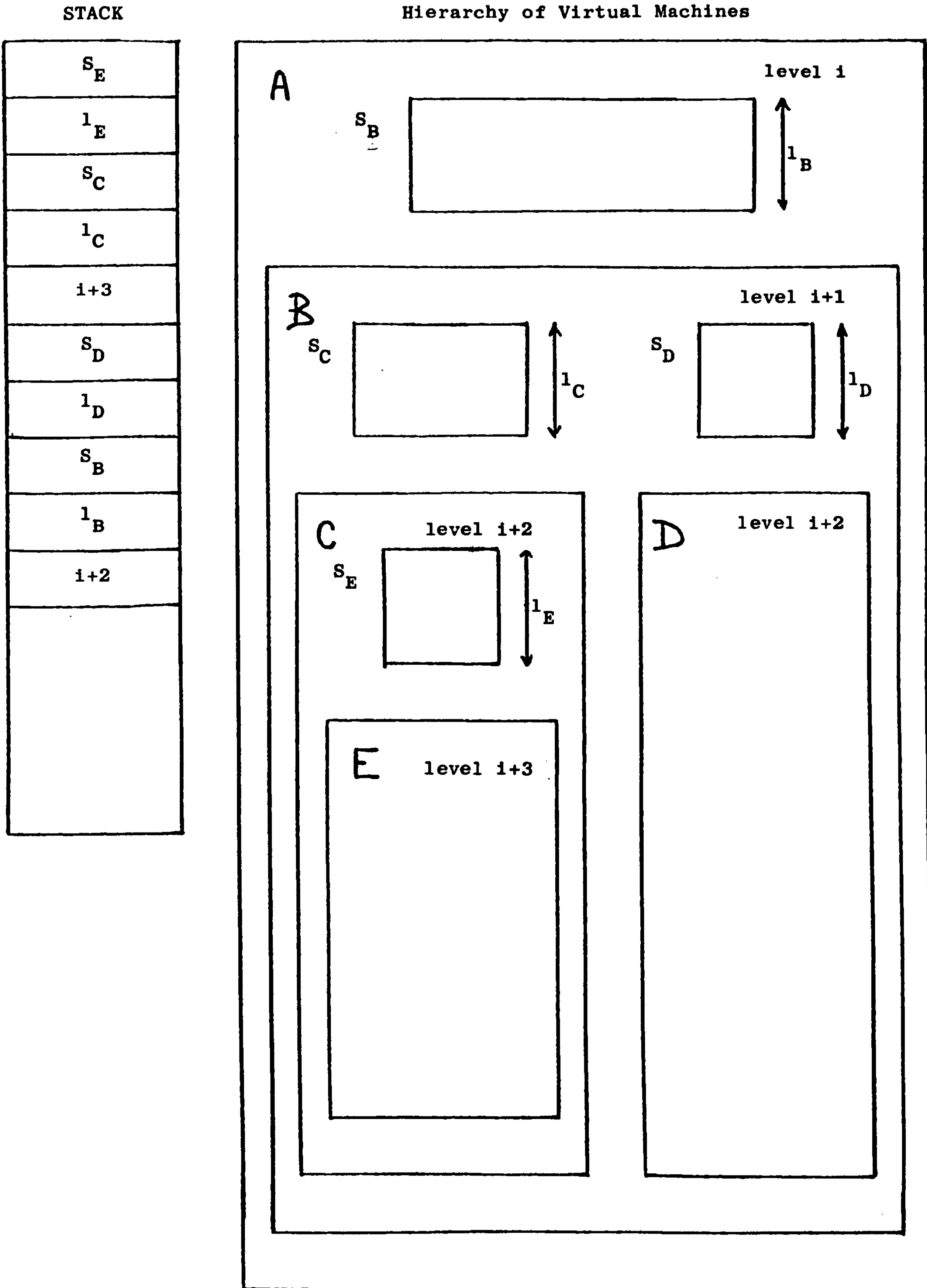


Figure 3.6

Example:

Referring to figure 3.5, a process within A requests some object provided at D. If this is supplied immediately and control returned directly then the 'segment table address' and 'segment table length' fields of environments D and B will ensure that the calling process is restarted. However if the called process within A creates the environment C in order to provide the object then this will overwrite the 'segment table address' and 'segment table length' fields within D. It will not then be possible to return control to the calling process without first passing control to a process within B which is capable of restarting the calling process in A.

The RVM implementation, described in the following chapter employs a hardware stack on which are placed the items from the arrays 'maximum segment number' and 'display' for each environment in the hierarchy which has been by-passed. Finally the old value of 'level' is placed on the stack in order that the hardware can return control correctly.

Example:

Referring to figure 3.6, a process within E requests an object provided at B. The display and maximum segment number values for levels i+3 and i+2 are placed on the stack as is current value of level. If, as a result of this, a process within D is created and this requests some object provided at A then the display and maximum segment number values for levels i+2 and i+3 are placed on the stack together with the current value of level. On returning to the process in D these latter items are unstacked, and when the object in B has been provided control can be returned to the calling process in E and the stack emptied.

A slight modification to the signal fault at level routine is required in order to provide this mechanism and it is now described as follows:-

```
procedure signal fault at level (i, code);
begin
    store hardware registers in current virtual memory;
    if i < Ø then halt
    else
    begin
        integer old level;
        old level := level; level := i;
        load hardware registers from current virtual memory;
        if code ≠ return code then
        begin
            integer j;
            for j := old level step -1 until level +1 do
            begin
                stack (display (j));
                stack (maximum segment number (j))
            end;
            stack (old level)
        end;
        store in current virtual memory (code)
    end
end signal fault at level;
```

The requirement for a mechanism which will permit a program to return control to a calling process within an unknown, more abstract, environment has already been discussed. There is always the possibility of incorporating this into the 'transfer to son' operation, however in the current implementation the slightly more straightforward approach of introducing a further, non-privileged, operation has been adopted; and this can be described in the following manner, thus completing a discussion of the special architectural requirements of the RVM.

```
procedure return control;
begin
      integer new level;
      store hardware registers in current virtual memory;
      unstack (new level);
      for level := level +1 while level ≤ new level do
      begin
            unstack (maximum segment number(level));
            unstack (display (level))
      end;
      load hardware registers from current virtual memory
end return control;
```

3.4 Protection in the RVM

The machine architecture which has just been described provides an extremely secure protection mechanism. A process executing within any environment of the system is unable to break out of this environment without accessing the segment table which defines the environment. Should a supervisory process pass this segment table to one of its descendants it is leaving itself wide open to the corruption of its own virtual memory and can be considered in error. The propagation of this protection violation can, however, progress no further down the hierarchy of protection environments, since only the environment of the process in error can be corrupted.

Also an attempt to gain access to an object which is not owned by the requesting environment will cause a fault to be signalled in the most abstract supervisory environment which has denied access to this object. Thus even if a process sets up an environment with a virtual memory containing objects to which it does not have access, a process executing within this new environment will still be unable to access these objects. In fact any attempt to do so will cause a fault to be signalled in the environment which has prohibited the use of such objects. From this it should be apparent that the recursive virtual machine addressing mechanism makes it possible to build a system based on a hierarchy of privilege and protection. There is, in fact, a natural ordering of types of access permitted. For example if memory objects are considered then a process may be given read, write and execute access to a particular segment and then pass this segment to more abstract environments. Alternatively the process may decide that the segment is only to be used for semaphores, the segment will then be passed as a new type on which only 'P' and 'V' operations are permitted. In particular it is asserted that:

The set of objects which a process can access is a subset of the objects which any of its immediately containing processes can access, and the 'privileges' (ie kinds of access) which it enjoys with respect to an object is a subset of those which any containing processes enjoy.

3.5 Service Calls Across Levels

As well as providing this hierarchy of protection and privilege, the system readily permits the creation of new types of objects, together with their associated operations. Once again it can be seen that if a process creates a new object and passes it to a descendant, then attempts to access the object will cause control to be passed to the object's creator, which is then able to perform the access in the required manner. Since in the original discussion of 'objects', in chapter one, it was stated that objects would be regarded as both physical and logical resource, eg. segments of code or abstract data types, and since an example of an abstract data type is an 'operation' upon another abstract data type, it is a logical step to envisage a mechanism which might be used to implement a 'supervisor call' or 'across level procedure call'. Some purists may regard such a mechanism for communicating between a virtual machine and its immediately containing environment as contrary to the spirit of the virtual machine concept. However, in any practical system, the need for such communication among levels is apparent (and sometimes awkwardly implemented).

A request for supervisory service is inevitably implemented by some operation which forces a change in context from the virtual memory environment of the caller to the virtual memory environment of the supervisor. In a conventional two-level system, the effect of permitting such an operation is that the supervisor, situated at the less abstract level, provides to its subordinate processes some new objects, eg. file transfer operations, pages, etc., and refuses access to some of its own objects, eg. disk file segments, absolute core addresses etc. Such a mapping is performed at every level of the RVM by the segment tables which define the environments in which the processes may execute, thus this mechanism provides a logical means of providing these 'supervisor call' facilities. A more powerful system can now

be contructed, for whereas in a two-level system there can only be one supervisor, in a system contructed using the RVM, each environment in the hierarchy acts in a supervisory capacity to those descendant environments at more abstract levels of the system.

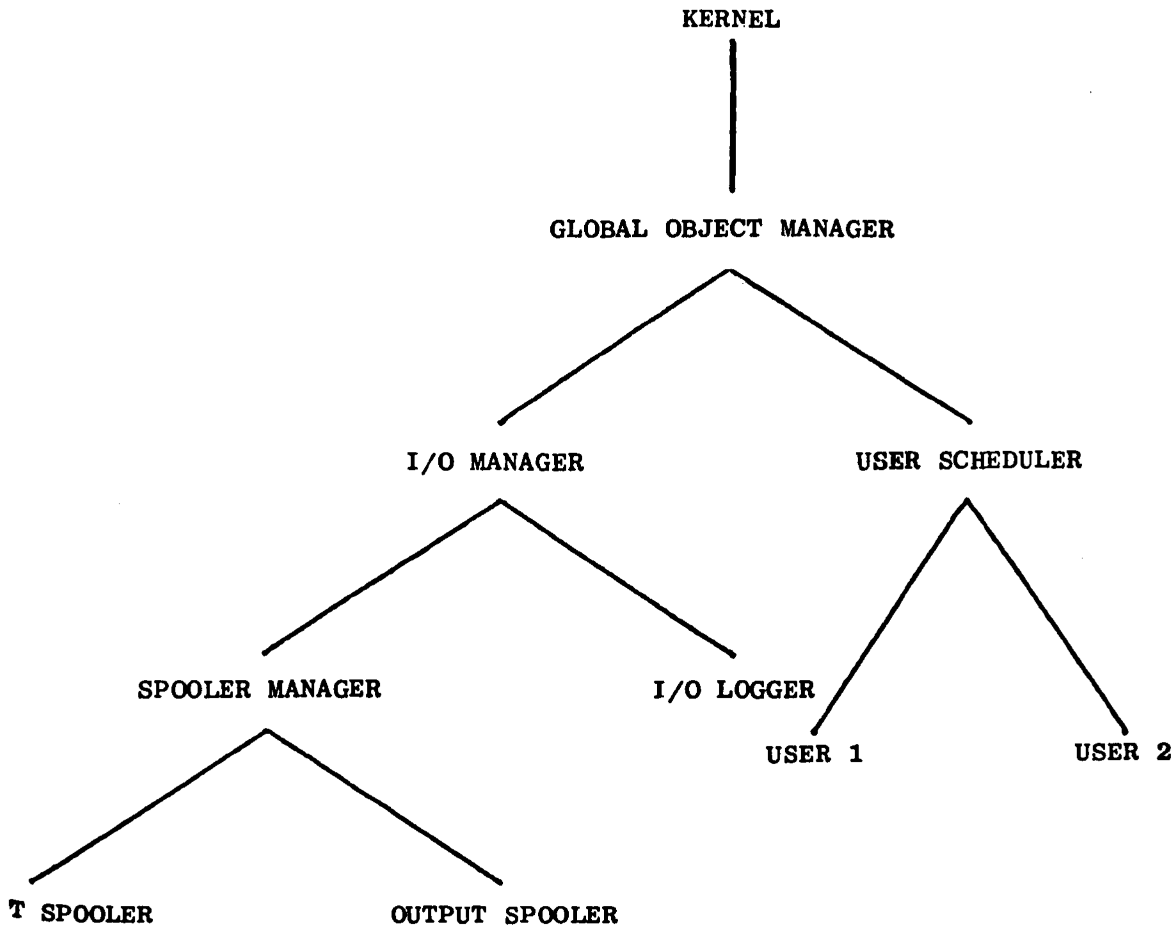


Figure 3.7

Thus, if an operating system were constructed as a tree structure of supervisory environments, an attempt, by any process, to print a line of output on the line printer might cause control to be passed to the Global Object Manager. Here a decision could be taken as to whether this process was to be given control of the line printer or perhaps the output was to go to a spooling file via the I O Manager. In the latter case a new supervisory 'spooler' environment would then be initiated, if necessary, which might then

pass control to a further environment which contains a process to output the required information onto magnetic tape. On completion of the task, control would then be returned to the I/O manager which would then return control to the Global Object Manager and thence to the calling process. This is the sort of sequence in which processes would be called in a conventional operating system, except in this case there is no protection between the various co-operating processes. This protection may indeed be desirable, especially if a new version of one of these processes is to be tested, and it can be achieved easily with the RVM. The tree structure of figure 3.7 illustrates such an implementation, each node of the tree representing a protection environment, and since each environment can be completely protected against the malfunction of any environment at a more abstract level, or an any other branch, then clearly the introduction of a new 'spooler manager' can only corrupt the 'input spooler' and 'output spooler' environments, thus ensuring the integrity of the remainder of the system.

3.6 Co-operating Processes

Of course, although it has been stated in this thesis that it may be desirable to completely protect each environment from more abstract environments and also environments in other branches of the tree structure, it is also possible for any two or more environments to share access to a particular object, providing this object has been placed in each environment's current virtual memory. Such co-operation between processes, especially within an operating system, is essential if 'real' systems are to be built. Study of the RVM reveals a straight forward solution to the problem of permitting two or more co-operating environments to gain access, via common entries in their segment tables, to a shareable object. There is, of course, the further possibility that this object may have a different name in each of the co-operating environments.

In such a situation, it is the responsibility of the supervisory environment for the environments which are sharing objects to ensure that any co-operating processes do not corrupt one antoher's virtual memories. This might be achieved by creating a new type of object, eg. a 'semaphore', which can only be accessed by the newly defined operations of 'P' and 'V'. Of course such objects may have been provided at a less abstract level, in which case they can simply be passed on to the co-operating environments. Of course the provision of 'P' and 'V' operations does not ensure their correct use by the co-operating processes. The supervisory process may prefer to provide 'monitors' which will ensure correct access to shared objects.

By restricting the objects which are to be shared, and by constructing them of types which can only be manipulated in an orderly fashion, it will

be possible to permit processes to co-operate in a safe and reliable manner. However, if processes are permitted to co-operate in a disorganized fashion, and as a result, corruption of one or all of the processes occurs, then the fault must be regarded as originating in the environment which set up the co-operating processes. Furthermore it should be noted that the protection mechanism will not permit any fault to propagate to a less abstract level than this supervisory one.

3.7 Applications of the RVM

In the Recursive Virtual Machine system, a protection mechanism has been developed which is based on a recursive, context - dependant addressing scheme. Processes operate within a virtual memory, which is defined by its immediate supervisor; also any process can create a descendant environment within itself. The mechanism which has been defined ensures that no process in the system need be aware of the fine structure of its descendants, nor whether or not they have descendants. This is because every request made to a process is framed in terms of its own virtual address space, no matter where in the hierarchy it originated. A process need only be concerned with allocating its resources to its immediate descendants, for they can allocate them further as appropriate.

Given this structure it has been illustrated how a copy of an operating system might run in a virtual machine under itself. Also it has been shown that the RVM fülfils the first four of the requirements for a machine structure as proposed at the beginning of this chapter.

Namely:- i) There is no supervisor state.

ii) The system is reliable.

iii) A sound protection mechanism is provided.

iv) Objects are renameable.

The fifth requirement, that of efficiency, has not been discussed at great length in this chapter. It has been shown how processes may automatically call other processes in environments at less abstract levels and how these processes might automatically return control to the calling process. It has been proposed that the mechanisms discussed in this chapter provide

theoretically efficient solutions to the various problems of providing a hierarchically structured computer system. This discussion will be extended further in the following chapter, where the implementation of a RVM is described and some experimental results obtained by executing programs at several levels of abstraction are discussed.

In conclusion, a set of mechanisms have been described which will enable a recursive virtual machine system to be constructed. Also the way in which such a system satisfies the criteria for providing a reliable, extensible computer system as proposed in this thesis has been discussed. By programming an interpreter, which provides these mechanisms, it has been illustrated that such an approach is feasible and this implementation is described in the following chapter.

CHAPTER 4 - IMPLEMENTATION OF A RVM

4.1 Introduction

Having discussed the mechanisms necessary to permit the construction of the RVM in the previous chapter, it remains to describe the effect of implementing such an architecture. In this chapter the implementation of a purely synchronous RVM is described.

This implementation was achieved by writing an interpreter in the micro code of the Computing Laboratory's Burroughs B1726 computer. This chapter documents this implementation. The various design decisions are explained, together with their resulting consequences, and some performance figures, obtained by executing some test programs at different levels of abstraction, are discussed.

Since the original intention was to permit processes to execute asynchronously no specific design decisions were taken which assumed a purely synchronous machine. Consequently it should be noted that although the current implementation allows only synchronous operations, any implementation of a mechanism to permit asynchronous processing should prove relatively straight-forward.

4.2 The Burroughs' Hardware

The machine configuration at Newcastle is a Burroughs B1726 with a main memory of 48k bytes and a control memory size of 4k bytes. Associated with this core storage is a range of associated peripheral devices which are not relevant to the discussion in this thesis. The design of the B1700 is such that when a program requests the use of a particular interpreter as much as possible of this interpreter is placed in the fast control memory. A copy of the whole interpreter is also stored in the main memory and an algorithm in the hardware decides from which memory the next micro-instruction should be fetched. The design of the B1700 thus encourages the use of different interpreters, each specifically tailored to the requirements of a particular task. In this way it is possible to provide several different virtual machine environments, each completely protected from each other and also capable of providing the type of resources required for their particular users. Several of these virtual machine systems are supplied by Burroughs themselves, Basic and Cobol systems for example; it has been claimed by Wilner [Wil 72a] that this attempt to vanquish the rigid structures of a conventional machine makes the B1700 nearly as efficient as a conventional machine at its best, and far more efficient in the majority of cases where the task to be performed is not easily mapped into a conventional architecture.

The flexibility of the B1700 in order to provide different emulated machines in which programs could execute proved the true value of the B1700. Indeed in terms of understanding the underlying hardware, the programming of the interpreter for the RVM proved to be a relatively straightforward task.
The B1700 has a bit addressability feature, again discussed by Wilner [Wil 72b], which it is claimed further increases the B1700's efficiency over conventional

machines. Indeed Wilner claims that by the suitable use of "Huffman-type" coding techniques, and variable length operations etc, the bit addressability can also dramatically improve main memory utilization. Bit addressability is, in fact, an extremely useful facility, even if it is not fully exploited to conserve main memory in the RVM implementation, for it enables an interpreter to extract sub-fields readily from items in store, eg operation codes from an instruction word.

A complete description of the B1700 hardware is given in the B1700 Reference Manual [Bu 72]. However, it is relevant to describe the main features of the machine as these influenced the final architecture chosen to represent the RVM.

4.2.1 General Purpose Registers

Within the hardware of the B1700 there are four general purpose registers X, Y, L and T. These registers are each twenty four bits long and can all be used to read or write items from or to main memory. Arithmetic and logical operations are performed by placing the required operands in the X and Y registers and the result is automatically provided in the appropriate function register, XPLUSY, XMINUSY, XANDY, etc. These function registers are also twenty four bits long, thus users of the B1700 are encouraged to operate on operands of less than twenty five bits. It is possible to use operands of less than twenty four bits by setting a register which determines the length of operands submitted to the arithmetic and logical 'function box'.

Shift operations are performed on the X, Y and T registers, the T register also being used to 'extract' a number of bits from within a twenty four bit field.

The L and T registers may be divided into six four bit sub registers, and these form the basis of the complete set of four bit registers. These registers can be tested against each other or a mask or have arithmetic performed upon them with the option of 'branching on overflow'. Using this facility it is possible to perform simple looping operations, however an upper bound of fifteen is placed on such calculations.

4.2.2 The Next Instruction Register

The next micro instruction to be executed is always held in the sixteen bit M register. It is permitted to load this register with any desired value, this however does not overwrite the original value, instead an 'inclusive or' of the original value and the new value is performed and the resultant value is used as the next micro instruction. Thus by the use of this technique it is a straightforward task to modify instructions as required and in particular avoids the use of tables stored in main memory when attempting to perform the equivalent of a 'goto switch' statement.

4.2.3 The Scratchpad Registers

There are thirty two scratchpad registers each containing twenty four bits of information. These may be accessed singly or in pairs as forty eight bit entities. The usefulness of these registers stems from the need to constantly change the contents of the general purpose registers. It is therefore possible to store a general purpose register temporarily in a scratchpad rather than writing it away to main memory every time it is to be overwritten. Also it is possible to store certain frequently used items in the scratchpads (the absolute main memory address of the current RVM

instruction being interpreted, for example) and in this way the efficiency of the interpreter can be increased.

4.2.4 Other Features

The previous sections have given a summary of the B1700 hardware features which have been utilized in the implementation of the RVM. Other important facilities, such as the 'soft' interrupts provided, have been used minimally. This has stemmed from the philosophy that the RVM is a system which has been designed to provide all its own protection mechanisms. Furthermore, the mechanisms provided by the B1700 prove incompatible with the requirements of the RVM

In fact, the original design aims were to build a system which would run completely independently of any Burroughs' software, except for any necessary 'boot strap' mechanism, and this would indeed be possible. However, since only a synchronous machine has been implemented, the final version still uses a considerable amount of Burroughs' software in order to perform the necessary I/O for testing purposes.

B1700 Memory Layout with RVM loaded

MAIN MEMORY

MCP BASE REGISTER

MCP CONTROL INFORMATION

INTERPRETER FOR RVM

RVM BASE REGISTER

RVM CONTROL INFORMATION

EMULATED MACHINE

16k bytes

RVM LIMIT REGISTER

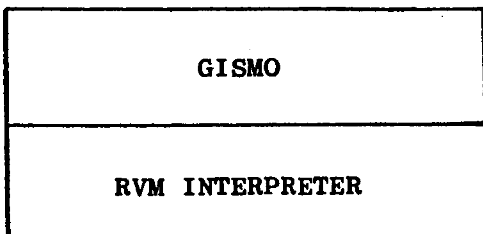
RVM RSN

GISMO

MCP LIMIT REGISTER

MCP RSN plus I/O INFORMATION

CONTROL MEMORY



4.3 Interfacing With the Burroughs' Supervisor

The fact that Burroughs' system software is still required to execute the RVM has had a dramatic effect on the overall memory utilization of the B1700. Both the Burroughs' supervisory system, MCP (Master Control Program), and the micro-instruction program which provides a standard I/O interface for the MCP, GISMO (Generalized Interface and Supervision for Multi-process Operations) are used extensively. As a result it is important that the RVM interpreter uses only those areas of memory permitted by MCP.

The decision to use MCP and GISMO has drastically reduced the amount of main memory which is available for use by the RVM, since system tables and other control information commandeer a large proportion of the main memory available. The current size of the interpreter for the RVM is 5k bytes. This includes all the micro-code necessary to interpret each RVM instruction, the RVM's internal tables plus a quantity of debugging software. As a result it was hoped to be able to make 32 k bytes of main memory available for the RVM to use as its personal core storage. Unfortunately the overheads of MCP's memory management system are such that it is only possible to load into memory an emulated machine with a store size of 16k bytes. (Details of MCP's memory management system are provided in the B1700 MCP Reference Manual [Bu 75]).

The manner in which the storage for the emulated machine has been mapped, although conforming to the rules laid down by MCP, does not in fact follow the pattern of other Burroughs' interpreters. Figure 4.1 describes the layout of the B1700 main memory while the RVM is executing, and the following paragraphs explain why this particular approach was adopted.

The Burroughs' approach, when providing interpreters for various 'virtual machines', is to separate the code which is being interpreted from the data which is being manipulated by the interpreter. In order to distinguish between the interpreter's code, micro-instructions , and the code which the interpreter is executing these are respectively referred to as M-instructions and S-instructions. The data is placed in an area defined by the program's Base and Limit registers, and any attempt to access an item in memory which is outside of this area will cause a bit to be set in one of the other hardware registers.

The interpreter is therefore able to detect, by examining this interrupt register, any attempt to read or write outside of a program's own data area. In this way an interpreter can process the S-instructions of a program which has been written in a re-entrant manner, and may have several data areas concurrently in use. However an interpreter is always able to override these checks and may then read or write data into any location within the main memory of the machine. In fact this is essential in order to set up data to communicate with the MCP.

An interpreter interfaces with MCP via an area of shared memory, known as the Run Structure Nucleus (RSN). Each process within the system, including MCP, owns such an area which is defined as a fixed number of bytes immediately following the process' data area, ie. Limit Register = location RSN[0]. The Run Structure Nucleus is used for storing both messages to be passed between two communicating processes and data which is required to be preserved for the duration of the inter process call, the scratchpad registers for example.

MCP makes extensive use of this area for inter process communication, in

particular standard items of Burroughs' interpreters are stored here. The items are typically the calling process' base and limit registers and the next instruction pointer for the S-instruction which is next to be interpreted. In the RVM implementation only minimal use has been made of the RSN. Only those locations have been used which are essential to maintaining a compatible interaction with MCP when requesting I/O operations etc. This is because of the underlying assumption that eventually the RVM will be able to operate independantly of MCP and GISMO.

Since the protection mechanism provided by the RVM is much more sophisticated than that of MCP, and since more main memory becomes available to the emulated machine if there is no physical separation of code and data in MCP's terms, the decision was taken to make the whole of the emulated machine's memory, data as far as the MCP is concerned. As a result, on loading the RVM, a minimal S-code segment is requested from MCP, this is then ignored by the interpreter. In fact the code is initalized in the data space of the basic recursive virtual machine. Currently this is performed by the interpreter as it initializes its workspace, it could however be performed by loading the required code from disc.

4.4 Design of the Recessive Virtual Machine

The main aims when choosing an architecture for the RVM were as follows:-

i) Simplicity - The instruction set and storage accessing should be simple in order to separate the problems of implementing a sophisticated interpreter from those of implementing an interpreter which would purely provide the RVM facilities.

ii) Suitability - The instruction set chosen must be suitable for the needs of the RVM concepts. In particular it was essential that the setting up of new environments should be straightforward.

iii) Generality - The architecture should be able to provide all the facilities of a conventional machine, as well as the special facilities required by the RVM.

4.4.1 The RVM Addressing Mechanism

At the centre of the RVM design is an addressing and protection mechanism based upon indirection through segment tables. An object named in one environment has its name mapped into the name of an object in the next, less abstract, environment. This mapping is performed by the use of a segment table, contained in the next less abstract environment, which defines the more abstract environment. Progress is made successively through these environments until the name of the object is obtained in the absolute terms of the emulated machine.

A word-based addressing scheme was chosen as this permitted a larger addressing range than a byte based scheme, and it was intended to permit the whole machine to be addressed from a single segment. Initial considerations, as to the amount of main memory available on the B1700 and the probable proportion of this available to the RVM, led to a machine size of 16K words to be chosen. It was felt that this would permit a sensible appraisal of the RVM's performance in a 'virtual memory' situation in the future, as well as permitting a reasonable amount of work to be done without using 'virtual memory'.

The decision was taken to make use of the four-bit arithmetic functions where possible since this would considerably simplify the looping functions. As a result an upper limit of fifteen was placed on the maximum segment number available in any environment at any level, also it was felt that this same limit would constitute an appropriate maximum on the number of levels of abstraction at which the RVM was permitted to execute.

With these considerations in mind, eighteen bits of information (four for the containing segment name and fourteen for the offset within a segment) are required in order to access each object. This fact, together with the sizes of other pieces of information to be stored in a word, led to the selection of a word size of thirty two bits.

The segment tables, which provide the mapping of object names in one environment to names in the next, less abstract, environment, must specify such information as the type of segment, the access permitted to objects within the segment the base address of the segment and the length of the segment. In order to allow a segment to start at any location within its containing segment, and to keep all this information within one word, the length of a

segment is measured in units of sixty four words. A description of a segment table entry can be seen in Figure 4.2.

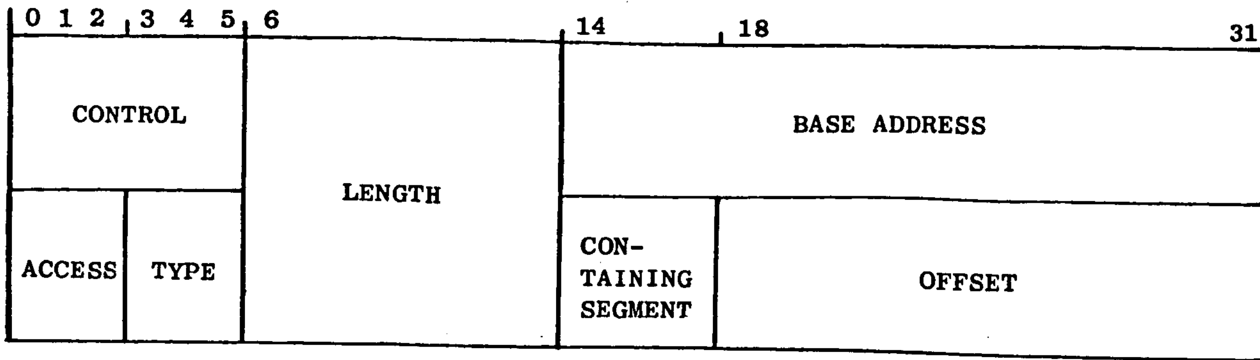


Figure 4.2

4.4.2 The RVM Instruction Set

For simplicity, a fairly conventional, low-level instruction set was chosen, however, the effectiveness of a block structured machine was also considered important. The result is a machine instruction set with an assembler code which resembles PL360 [Wir 68] , each assembler code statement having a direct one-to-one correspondence with a machine instruction.

The RVM instructions are of the conventional assembly level format with an operation code and list of operands. The conventional operations of 'add', 'subtract', 'load', and 'store' etc., are enhanced by operations which transfer program control between environments at various levels of the system. In an attempt to encourage a 'structured' approach to programming the RVM, the typical 'Jump' and 'Skip' instructions have been abolished. Instead a stack mechanism has been adopted, similar to that described by Organick

and Cleary in their discussion of the Burroughs B6700 computer system [OC 71].
The resulting set of operations, which perform program control within an environment, are of a 'high-level' nature; eg 'call', 'cycle', 'case', 'if then else', instructions have been implemented, together with a controlled 'exit' operation which allows program control to be passed to the end of a 'block' of code. It is intended that programs be written in a block structured low-level language which can then be easily compiled into machine code. In fact, although a compiler for this machine has not been written it has proved very simple, albeit tedious, to hand translate AlgolW programs into this machine code.

In an effort to keep the RVM as suitable and general as possible, eight hardware registers are available to the currently executing environment. Most RVM instructions have two operands, one of which is a hardware register, and the other is obtained from the name of an object in memory. This second operand may be either the name of the object, a variable number of bits of information addressed by the object or the object itself. Also the name may be given indirectly through as many memory references as required, indexed via one of the hardware registers, or directly as named in the instruction. Not all of these options are available to each instruction as this leads to a contradiction in some cases. This flexibility in addressing the virtual memory of a given environment will provide a straightforward mechanism for setting up the segment tables which define new environments.

4.4.3 RVM Program Control

The instructions which govern program control, eg 'cycle', 'call', etc., also have associated with them information regarding the RVM memory locations at which execution is to resume. Each item of information gives the address and length of a fragment of code to which control may be transferred.

A 'call' instruction for example has one such fragment descriptor, but an 'if then else' instruction will have two fragmented descriptors, one for the 'then' part of the instruction, the other for the 'else' part. Instructions are thus variable in length and take the form shown in Figure 4.3

RVM Instruction Format

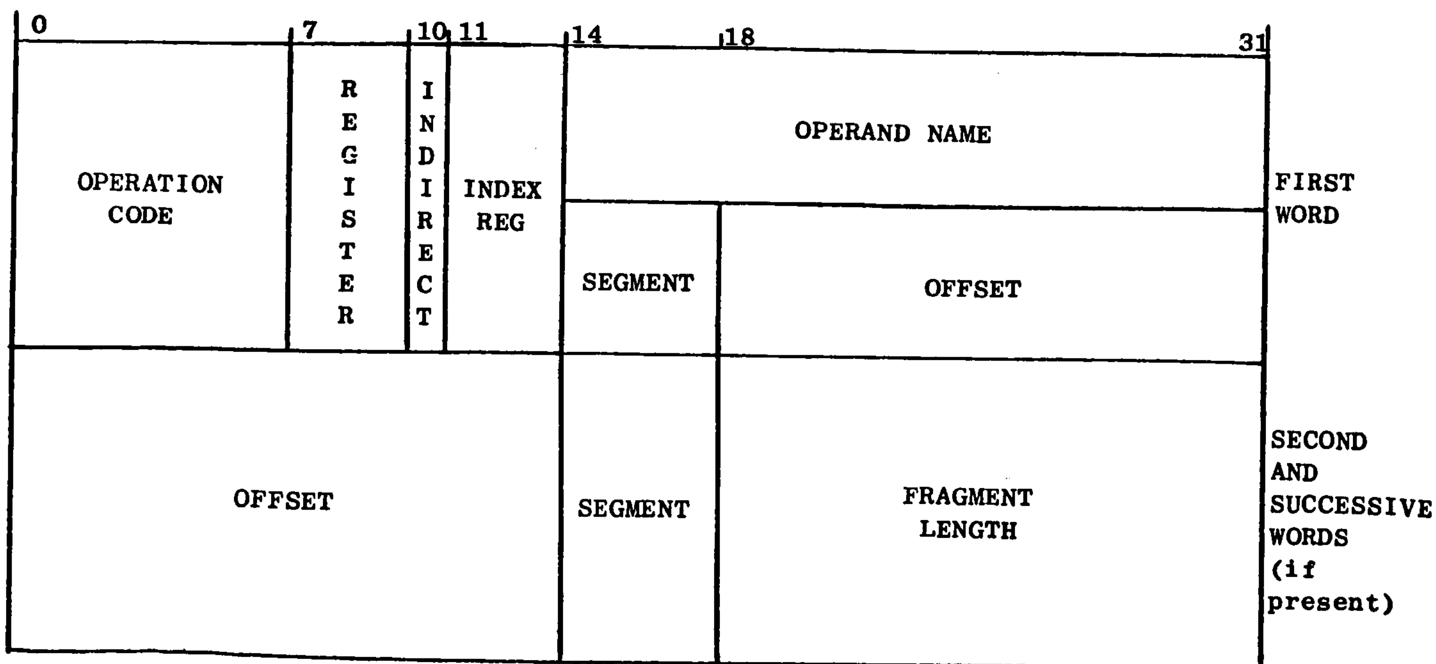


Figure 4.3

On interpreting one of these control instructions, the RVM emulator places the current next instruction pointer, suitably updated for the next instruction in the current fragment, on a stack. A new next instruction pointer is then set up from the required fragment descriptor. Depending upon the type of fragment which is being executed, on reaching the last instruction in a fragment, control is either passed back to the start of the current fragment, eg 'cycle' type fragment, or the top item on the control stack is removed and replaces the next instruction pointer, eg. 'call' type fragment. The

'exit' instruction causes as many items as required to be removed from this stack, the last of these being used to replace the next instruction pointer.

The fragment descriptor, which defines the fragment of code to which control is about to be passed, is extremely flexible. Fragments may be of any length, from one to 16k words, and furthermore different instructions may access the same fragment in a different manner. Thus if in one part of the program a section code is required to be exercised conditionally upon some test this can be accessed via an 'if then' instruction. If in another part of the program it is required to be accessed unconditionally, then a 'call' instruction can be used, and if it is required to execute the same piece of code repeatedly then a 'cycle' instruction could be used, all of which use an identical fragment descriptor.

One consequence of this approach is the requirement of a segment, of variable size, in each environment, which contains the stack of next instruction pointers. However such an approach has the added advantage that recursive procedures can be implemented trivially. The segment name which contains the control stack in each environemnt is fixed by the RVM emulator. Should the segment provided to an environment for use as a control stack prove inconvenient, the structure of the RVM is such that a process within this environment is permitted to set up a new, more abstract, environment with the desired attributes.

Thus it is suggested that the architecture is more flexible than that of a conventional machine, while forcing programs to be written in a structured fashion.

4.4.4 The RVM Data Stack

To facilitate the use of procedure calls, one of the registers is distinguished and always contains the top item of a stack of data objects. Any attempt to reference this register causes the top item of this stack to be accessed. The register has the same name in every environment, and is known as register seven.

The use of this data stack therefore requires a further segment for each environment, again variable in size, in which items of this stack may be placed.

4.4.5 The Environment Save Area

One further requirement when setting up a new environment is the provision of a segment, which need only be the minimum size, for the storage of the hardware registers, next instruction pointer and other information pertaining to a particular environment. Any attempt to access the locations housing these items will, in fact, cause the relevant item in the hardware to be accessed, thus allowing register to register operations directly.

These three segments, essential for the existence of an environment at any level of abstraction are named as follows:-

Segment Ø - contains the copies of the hardware registers etc.,
Segment 1 - contains the control stack items,
Segment 2 - contains the data stack itmes.

Details of all the operations available, together with their various modes

of working are given in Appendix 1, and Appendix 2 contains details of the programs used in the experiments described later in this chapter. A study of these two sections should give the reader an extensive insight into the mechanisms available when programming the RVM.

4.4.6 Programming the RVM Interpreter

Initial development of the interpreter for the RVM was performed using the Burrough's proprietary Micro-Implementation Language, MIL. However this was quickly discarded when the compilation times for the interpreter became prohibitive. A further disadvantage was the COBOL like structure of MIL. As a result it was decided to adopt BML as the standard compiler for the B1700 micro-code.

BML is a language which was initially described by De Witt et. al. [DE 73] in order to evaluate the performance of the B1700. A basic BML compiler had been written at Newcastle and this was modified and improved in order to make BML a practicable alternative to MIL. These modifications permitted sections of the micro-code to be compiled independantly, and enabled these segments of code to be 'linked' together prior to loading a new version of the interpreter. This considerably improved throughput during testing of the interpreter, for now it was only necessary to recompile the invalid segment of code, rather than the whole interpreter, each time an error was detected.

Enhancements were also made to the compiler which permitted the association of identifiers with constants and also strings of text to be compiled later. It is this package that was used to provide the implementation

of the RVM interpreter; examples of the segments of code produced are included in Appendix 3.

4.4.7 Hardware Features of the RVM

Having already discussed in chapter three the 'level', 'display' and 'maximum segment number' mechanisms necessary to provide the recursive virtual machine functions, and the operations which will permit the switching of processes from one environment to another, it remains to describe the manner in which they have been implemented.

The 'display' is held as an array of fifteen 24 bit words in main memory, with each item containing the absolute bit address of the segment table being referenced. The 'level' is stored in a four bit register which is reserved by the interpreter for this purpose. The array of 'maximum segment numbers' is stored as a fifteen 4 bit word array in main memory, and the stack of 'called environments' is also held in main memory with a pointer which addresses the top item on this stack.

4.4.8 An Associative Store for the RVM

One refinement has been added to the implemented RVM. In addition to the basic requirements for a RVM as described in chapter three, an associative store which provides the mappings of objects in any environment into objects in the absolute machine is supplied.

Associative stores are not a new concept, they have been used on machines such as the IBM 360/67 and the GE 645 to improve the performance of paging algorithms. Extensive discussion has taken place on the use of such

devices, Parmelee et. al. [Pa 72] in relation to CP-67, Schroeder [Sc 71] in relation to the operation of MULTICS, and Buzen and Gagliardi [BG 73] in relation to virtual machine systems such as the RVM.

The comprehensive experiments of Schroeder, who monitored the operation of MULTICS with different configurations of the size of associative memory used, led to the adoption of a sixteen item store in MULTICS, since this was sufficient for a general purpose programming environment. Currently the RVM implementation also has a sixteen item associative store, though no experiments have been performed to discover whether or not this is optimum in this case. This may indeed be an area for further research, since the pattern of memory accesses in the RVM will be very different to that encountered in a more conventional machine, due to the necessity to access one or more segment table entry per attempt to access any object.

The choice of a single associative store to increase the overall efficiency of the RVM is an important one. A more obvious approach would be to provide several hardware registers (up to 16 in the case of the RVM implementation described) which map the base of each segment in the current environment into the absolute terms of the real machine. This is the approach adopted in the GEC 4000 series computers [GE 74] .

Within the GEC 4000 an environment is constructed of several segments, and while a process is executing within an environment it is constrained to access only those items of memory which are mapped by its four current segment table registers. In order to access other segments in its environment the process must first load a suitable current segment table register. This attempts to ensure that a process does not unnecessarily

change current segment table items, since the programmer is made aware of the implications of such a decision.

However, in the RVM, it was felt that these details should be hidden from a programmer of a process, and, more important, since it was wished to permit the rapid switching of environments, possibly every instruction, then the hardware register approach would become inefficient. Users of the GEC 4000 series computers are not expected to rapidly change environments. However it can be seen that any time this occurs the current segment table must be reloaded, an overhead which was considered unacceptable if the RVM is to operate in the manner originally envisaged.

The use of an associative store, together with a least recently used algorithm for the replacement of items, ensures that items currently being accessed regularly, no matter from which environment, remain in the store provided that the store is large enough with respect to the locality of the programs being run. Items are only replaced in the store when they have not been accessed for a considerable time. It is therefore expected that this approach will prove more effective and that memory references should take the same order of time at each level in the hierarchy.

Objects within the store are addressed in the following manner.

i) Local name within an environment, given as segment number and offset; together with

ii) Environment name, given by the level at which this environment exists, the absolute address of the segment table defining this environment, and the number of segments available to this environment.

On finding the name of an object in the associative store, its absolute address, again given by segment number and offset, together with the permitted access are returned. A description of the associative memory can be found in Figure 4.4

Associative Memory

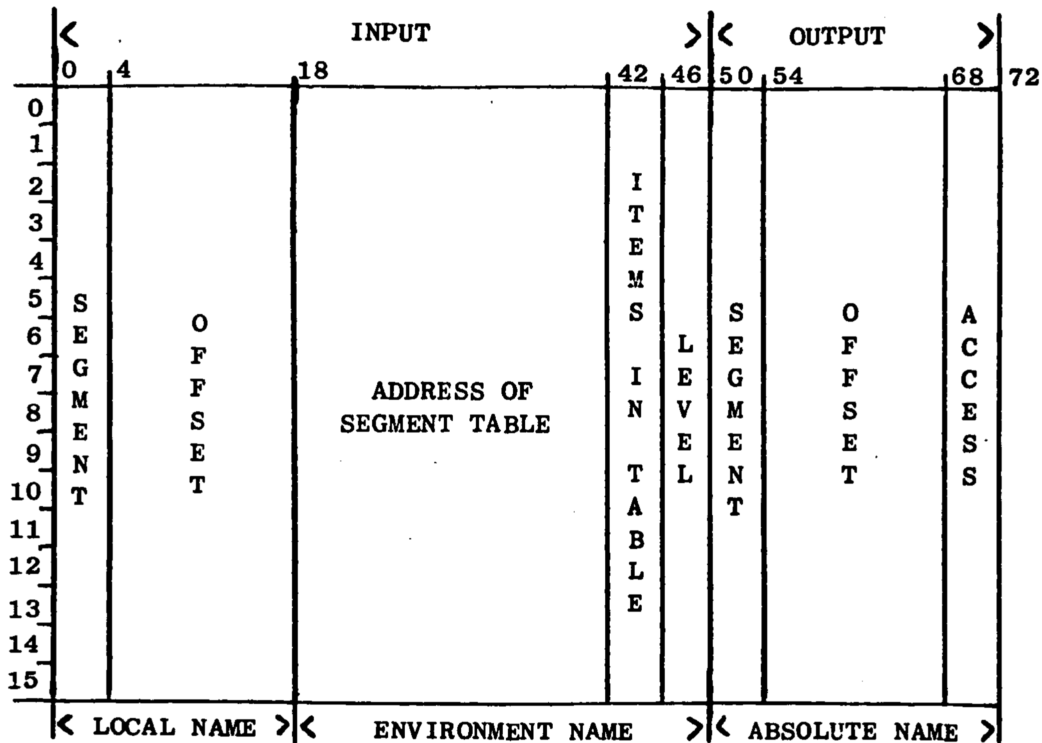


Figure 4.4

It should, of course , be noted that the approach adopted here differs from that used in current cache stores. In a cache store the object itself, rather than its name in absolute terms, is produced. Since the machanism was only being simulated in software, it was decided not to implement this extra refinement. The major reason for this being the extra overhead involved in writing changed items back into main memory and the extra checks involved when they are removed from the associative store.

4.4.9 Conclusions to RVM Design

In this section a detailed description of the design of the RVM has been undertaken. The micro-code used to implement the various algorithms essential to the performance of the RVM is included, as listings of the BML code used, in Appendix 3. It is to be noted that attempts to pass control to a level of abstraction below that of the hardware, results in the machine halting, displaying the reason for the halt. Thus no instructions require special privileges, and it is claimed that the implementation meets the design aims described in Chapter three.

4.5 Performance of the Recursive Virtual Machine

In order to evaluate the performance of the RVM two sets of statistics were gathered. These concerned the time spent at each level of abstraction in the system and the number of addresses which were found in the associative store.

An initial set of experiments was designed to discover how the RVM address translation algorithm performed at different levels of abstractions and also how the performance of the RVM was affected by the inclusion of the associative store.

Later, a second experiment was designed to assess the performance of the RVM when its environment crossing properties were utilized.

4.5.1 Performance of RVM's Address Translation Algorithm

Under these initial experiments it was important to assess the viability of executing programs at different abstract levels of the RVM's environment hierarchy. As a result a program was initialized in several environments, and no Input or Output was performed while the statistics were being gathered. Each program was then executed in turn, on completion of the program at one level of abstraction the program at the next level of abstraction was initiated until the program at the most abstract level was completed. Having completed all the programs, control was passed progressively down the levels until the least abstract level was reached and the RVM halted. At this point the statistics were printed for subsequent analysis.

Since an identical program is executed at each level of abstraction and there are no input or output processing overheads involved, the statistics obtained relating to the time spent at each level of the RVM indicate the performance of the RVM's address translation algorithm at different levels of abstraction.

These initial experiments were designed to illustrate the generality of the RVM and the manner in which new more abstract environments could be entered and then control returned to an immediate ancestor environment. The only feature of the RVM which is not illustrated by these experiments is that of direct transfer from one environment to a less abstract one and return to the calling environment. This feature is illustrated by the second experiment which is described in the following section.

For each test, results were obtained of the execution time taken both with and without the associative memory. Also, from these figures, an estimate was obtained of the time required to execute the same program at each level of abstraction if no display mechanism was available. Analytically it can be shown that if it takes 't' seconds to perform a memory reference in an environment at the lowest level of abstraction, then it will take $t*(2^i - 1)$ seconds to perform a memory reference at the ith level of abstraction.

Performance of RVM

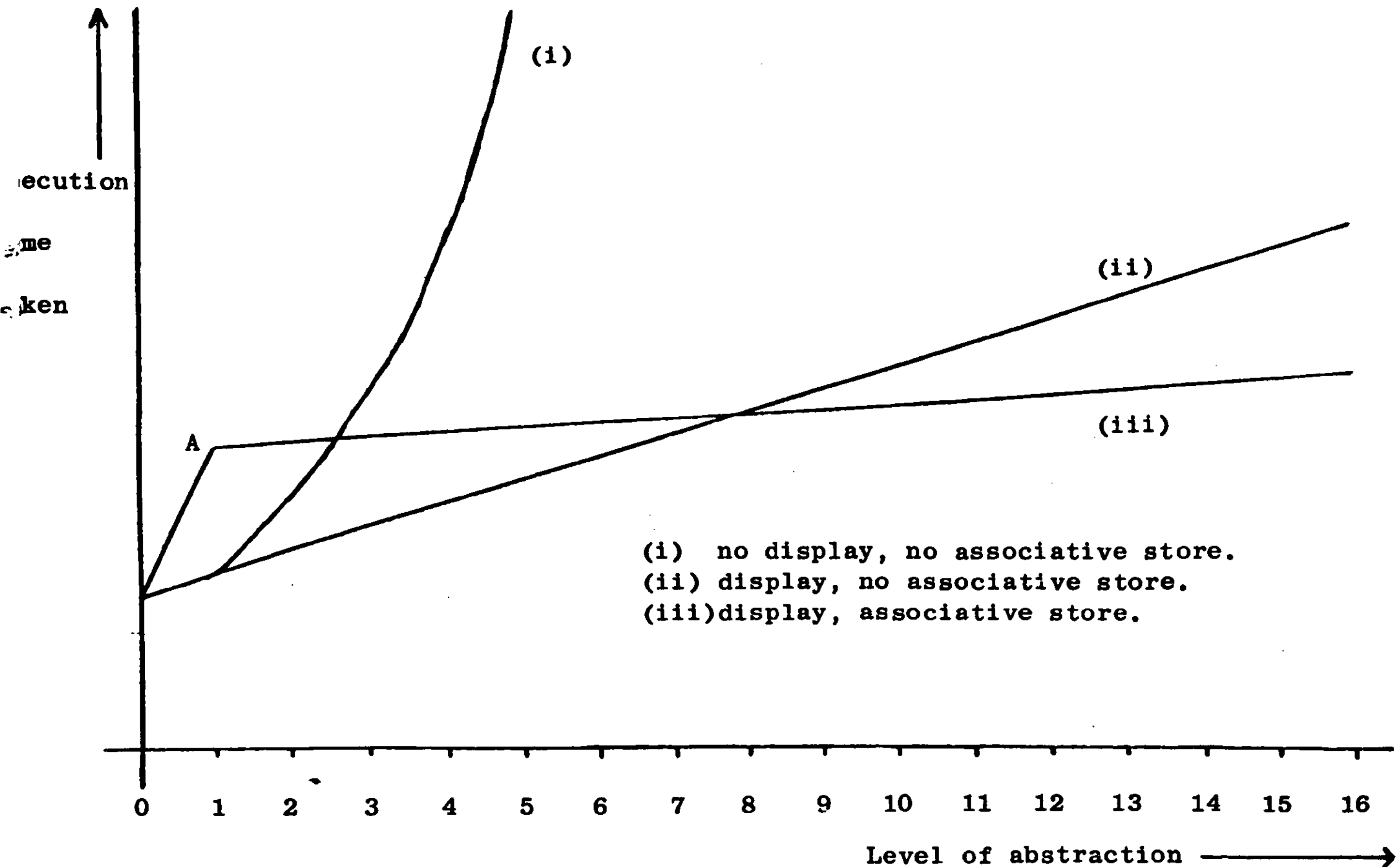


Figure 4.5

A total of four tests were performed, each producing performance figures which followed the same pattern, and this is summarized in Figure 4.5. The tests chosen were as follows:-

i) A program to calculate all the solutions to the eight queens' problem.

ii) A program to sort data into ascending order using a bubble sort method. Three versions of this test were executed.

ii.i) 69 items of data, initially in descending order.

ii.ii) 131 items of data, initially in descending order.

ii.iii) 131 items of data, initially in random order.

With the bubble sort experiment, the use of data initialised in descending order proved that the associative store could be utilized very effectively and thus the data was initialized in random order for one test. The use of the 69 item sort enabled statistics to be gathered for levels Ø to 11 whereas in all the other tests statistics could only be gathered for levels Ø to 9. However the results of these experiments indicate nothing unexpected by executing the program at these two higher levels of abstraction.

The programs which provided these tests are described in Appendix 2, and a summary of the important details of each test is included in Figure 4.6

Summary of Performance Tests

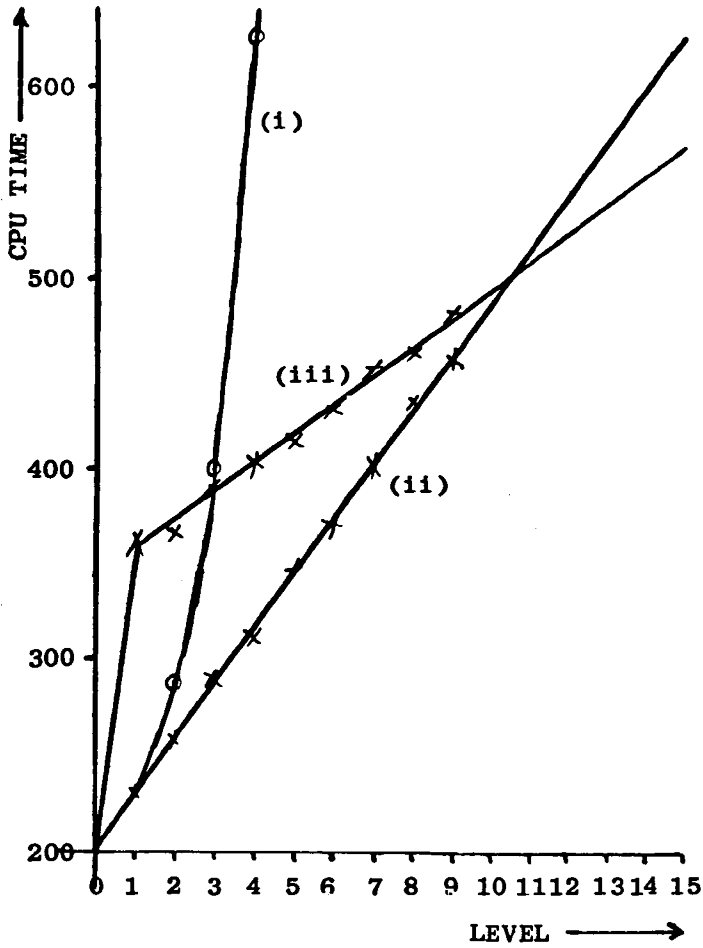
TEST	Memory References per level	Hit Rate in Associative Memory	Number of levels of abstraction used
A	139345	45.6%	10
B	33192	92.0%	12
C	123841	92.4%	10
D	69685	85.8%	10

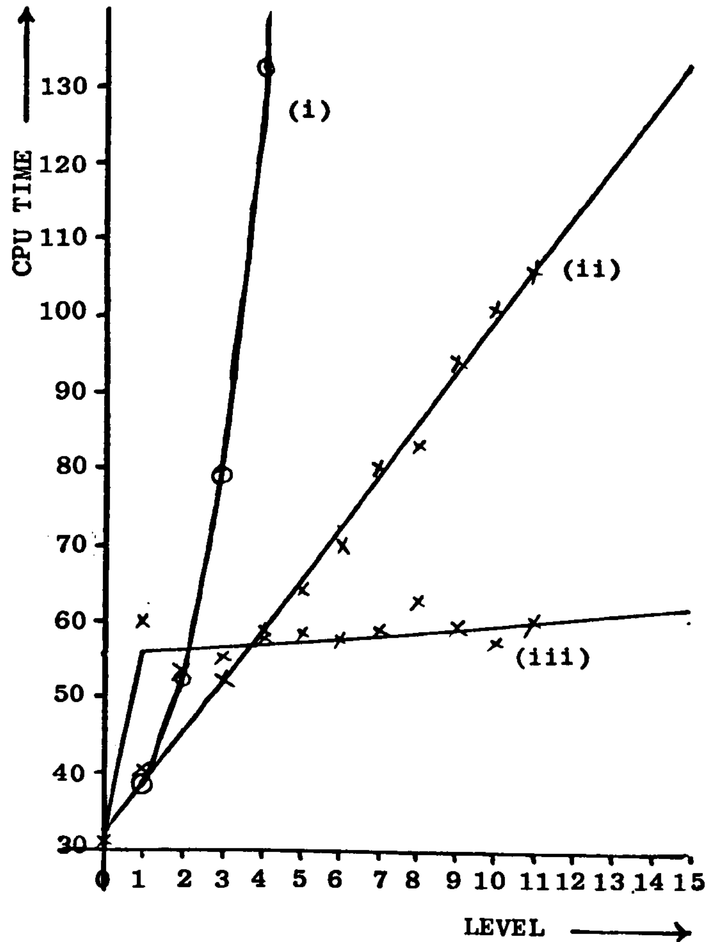
Figure 4.6

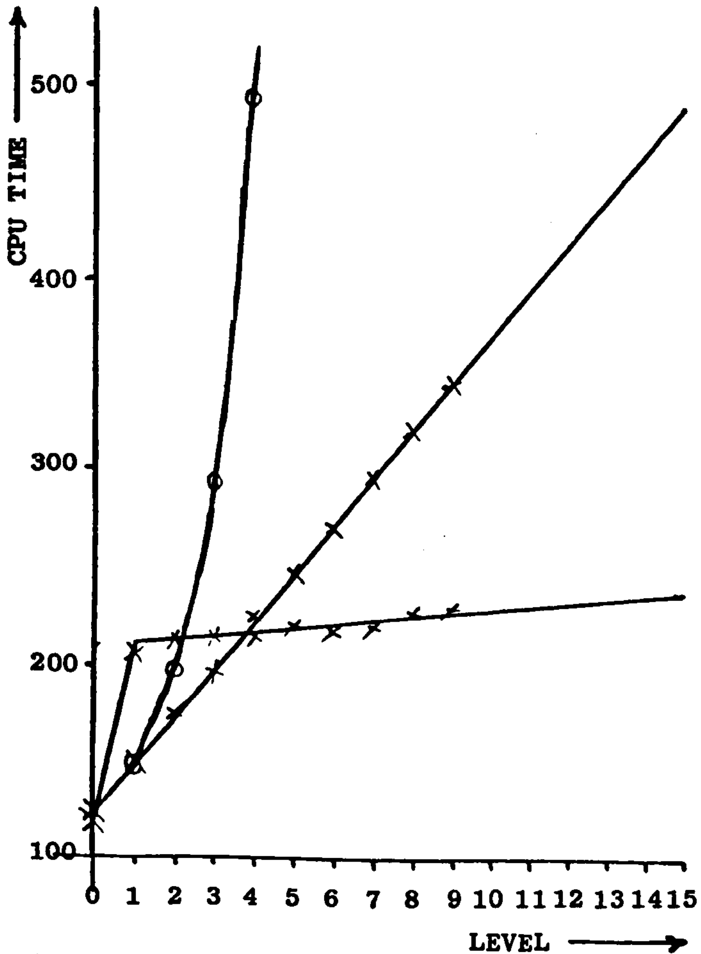
Since these results of all these tests followed the same pattern, the major details will be discussed in terms of Figure 4.5 and the individual details of each test will be mentioned later.

As can be seen from the graph, if there is no display mechanism, execution in environments at not very great levels of abstraction soon becomes impossible due to the prohibitive amount of time required to calculate an address within that environment. As expected the display mechanism does reduce the amount of time spent in the address translation algorithm to a linear scale, and the gradient of this line is dependant upon the number of memory accesses per level and the efficiency of the algorithm. The introduction of the associative store can dramatically reduce the gradient of the graph, however it is interesting to note that the software implementation of the associative memory introduces a "knee" point 'A', which is substantially higher than the equivalent point on graph (ii). This can be explained as the overhead required to access the associative memory. Since it is never accessed at the lowest level of abstraction there is no overhead here, but upon executing a program in an environment at the next, and all more abstract levels, a further amount of c.p.u. time is used in searching for each item in the associative store. A hardware mechanism should cause this 'knee' point to be considerably reduced, thus greatly improving the overall performance of the recursive virtual machine.

The gradient of line (iii), after the first level of abstraction, is directly related to the 'hit rate' achieved in the associative memory and the gradient of line (ii). A hit rate of 0% will produce a line with the same gradient as that of (ii) and a 100% hit rate will produce a gradient of zero, ie. a line parallel to the x-axis.







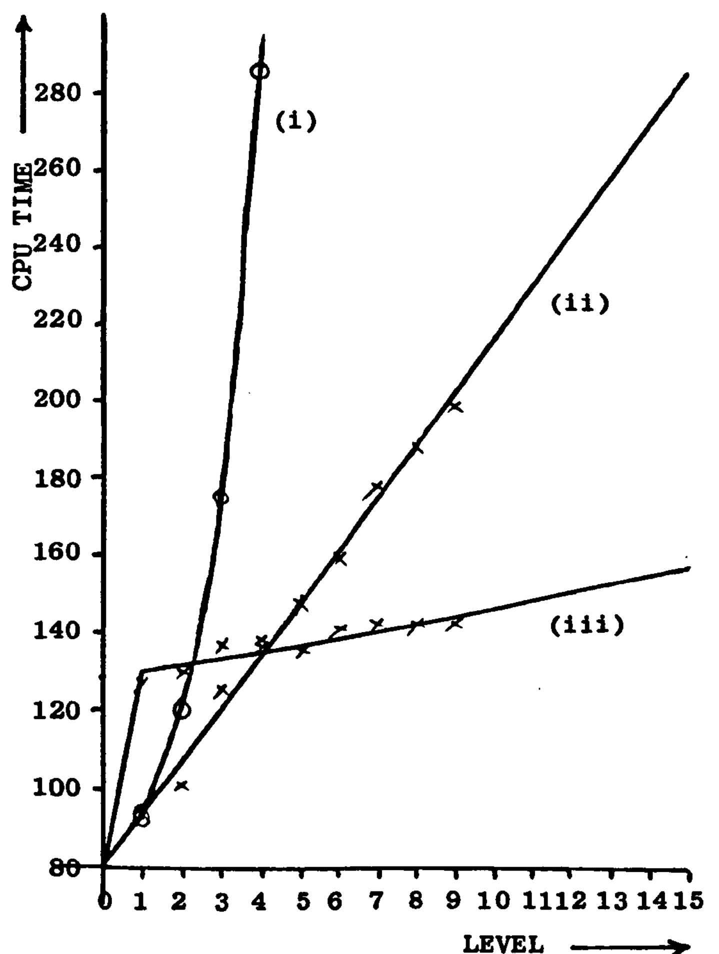


Figure 4.7

The set of graphs for each test are shown in figure 4.7, each individual graph being interpreted as follows:-

i) This line indicates the performance figures to be expected with no display mechanism and no associative memory.

ii) This line indicates the performance figures obtained with a display but no associative memory.

iii) This line indicates the performance figures obtained with a display and an associative memory.

The graphs are all of execution time against level of abstraction at which execution is taking place. Since each program is executed identically at all levels of abstraction, these graphs indicate the overhead incurred by the address translation algorithm of the RVM. The results from each test for the graphs (ii) and (iii) were fed into a polynomial curve fitting program and in all cases it was discovered that the best match was obtained by using a least squares approximation to a straight line. This therefore is the way in which the lines have been plotted, graph (i) being claculated from figures obtained by using this approximation to graph (ii).

The tests used gave hit rates of between 45% and 95% in the associative memory. This indicates that the gradient of line (ii) can be reduced considerably by the introduction of a simple sixteen item associative memory. The low figure obtained for the Queen's Problem may be partially due to the fact that an unnecessary number of variables were introduced in order to prevent an unrealistically high hit rate in the associative store. On the other hand the high figures obtained for the 'worst case' Bubble Sorts may be caused by unnaturally consistent accesses from the

store. The Bubble Sort algorithm works by examining adjacent items in an array, swapping any in the wrong order. Thus in the 'worst case' situation all items are swapped on each scan of the array holding the data. A more accurate figure for a 'typical' program will probably lie nearer the figure of 85% obtained for the Bubble Sort where the items were in random order.

As has already been mentioned, no experiments were made regarding the optimum size of associative memory for the RVM. The experiments performed illustrate the viability of an associative store in the RVM, but further improvements may be possibly by changing either the size of the memory, the information it contains, or both.

4.5.2 Experiments with the Environment Crossing Property of the RVM

Two further tests were performed on the RVM. These were designed to prove that the implementation permits environment crossing in order to perform 'privileged' operations, the RVM trapping to the less abstract environment as defined in chapter three.

The first test was very basic and caused a line to be printed indicating the level of abstraction at which the program was currently being executed. This illustrates the manner in which a program can cause control to be passed to a less abstract 'supervisory' environment and then have control returned with no intervention from intermediate environments. No performance figures were obtained since insufficient processing was performed at each level of abstraction. The program used is described in Appendix 2 and the output produced is also included there.

Performance of RVM when crossing Environments

Bubble Sort (126 items)

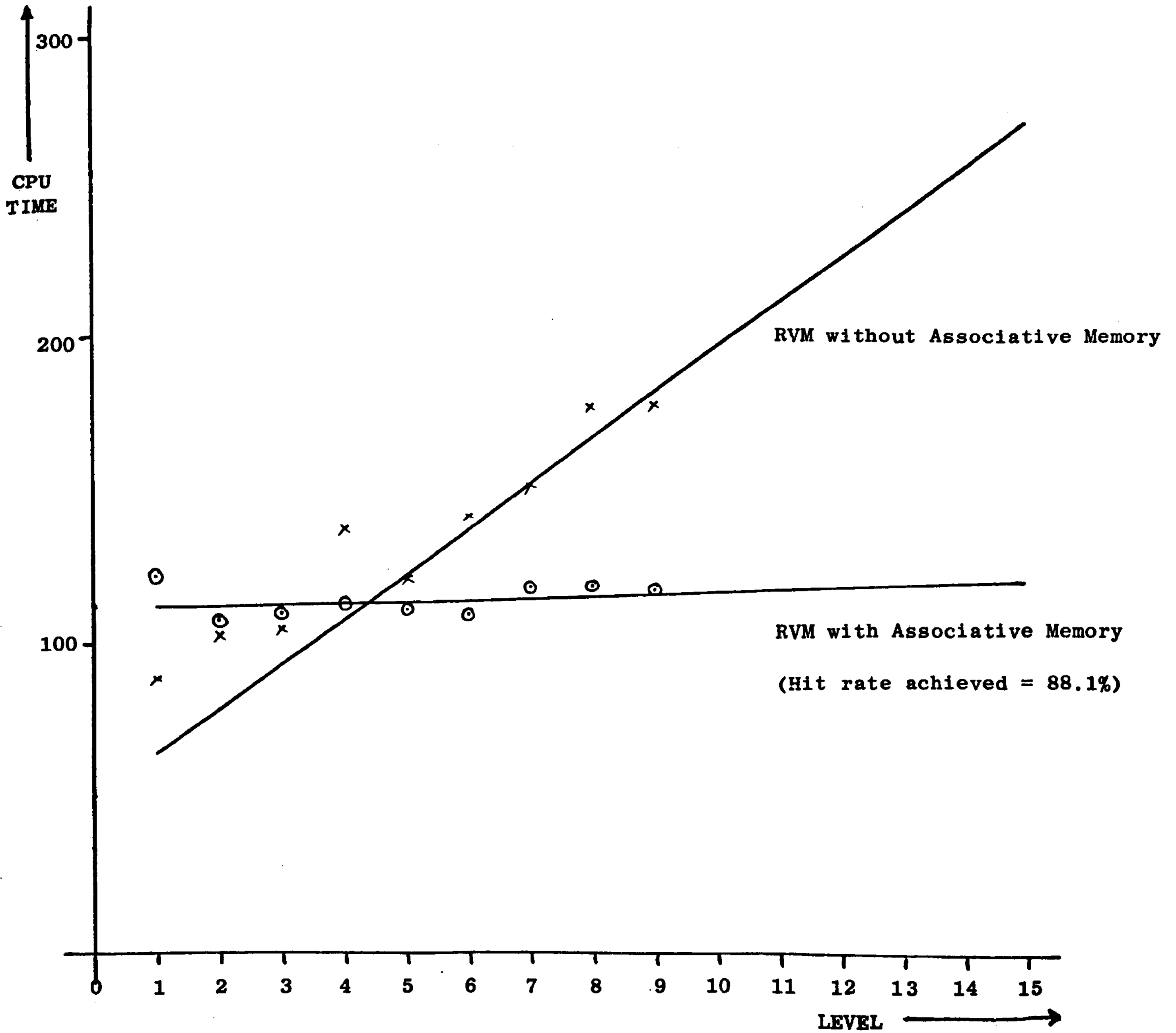


Figure 4.8

The second test is designed to show the performance of the RVM while crossing between environments at different levels of abstraction. A program executing in the least abstract environment provides a 'SWAP' operation to all programs in more abstract environments. Programs in all more abstract environments then perform a 126 item 'Bubble-Sort' with the data in the 'worst case' order. The programs at level Ø and more abstract levels are described in Appendix 2 and the results obtained from the test are shown on the graph, Figure 4.8.

The execution time taken by the program at level Ø is not shown in this case as these figures are rather meaningless. The program at level Ø purely provides the 'swap' operation for all more abstract levels and in fact this is where the majority of processing takes place. The two graphs drawn are once again least squares approximations to straight lines through the points plotted. It is encouraging to note that the effect of crossing between environments certainly does not add an exponential overhead as could have been possible with some implementations. In fact when the results of this test are compared with those of Test C from the previous set of experiments, the indications are that the gradient of the graphs have decreased rather than increased.

Unfortunately there is no exact way of actually assessing the overhead of crossing between environments. With any two programs, designed to perform the same function in different ways, a different set of primitive operations will be performed. Clearly some of the improvement takes place because several operations less are performed at each level in the latter test. This is because the 'swap' operation is performed at level Ø. However these two experiments are so similar that it is

reasonable to assume that any overhead incurred by crossing between environments is minimal, possibly less than that incurred when performing one of the primitive machine operations.

One further point, however, remains to be discussed and that concerns the extra overheads incurred by the program providing the 'swap' operation. The current implementation requires that this 'supervisory' program calculates the addresses of all resources it requires to access, and this can mean tracing up the segment tables in order to access an object in the calling environment. In the general case this may involve the execution of several basic machine operations, however it should be noted that the addressing overhead per instruction is less in the supervisory environment than that of the calling environment.

In chapter six a mechanism is proposed which permits asynchronous processing in the RVM. The mechanism suggested provides a means of mapping resource names between any two levels in the hierarchy of the RVM, and it is intended that such an approach will lead to a considerable reduction in the mapping overheads incurred by 'supervisory' programs.

4.6 Conclusions

In this chapter the implementation of a purely synchronous Recursive Virtual Machine has been described. The results of experiments undertaken to determine the efficiency of this RVM when executing instructions at different levels of abstraction and also when transferring control directly by passing environments, illustrate the feasibility of such an architecture for providing a hierarchically structured operating system. The overheads involved in performing operations at different levels of abstraction increase in a strictly linear fashion which conforms to the criteria for efficiency defined in Chapter one. Furthermore it has been demonstrated that the introduction of a simple sixteen item cache store can dramatically reduce this overhead.

The experiment to evaluate the performance of the RVM when crossing between environments illustrates that the cost involved is again linear and compares favourably with that of performing a basic machine instruction. Comparing this mechanism with those employed by VM370, CAP, HYDRA etc., it can be seen that the RVM really does provide a means whereby operations for a process executing in one environment can be interpreted equally efficiently by a process in any other less abstract environment, or on the hardware of the bare machine.

In order therefore to produce a RVM implementation which can be used sensibly, it remains to produce a mechanism which permits a useful degree of asynchronous processing. The complexities of this problem and a possible solution are discussed in the following chapters, and thus it is proposed that the RVM implementation undertaken on the B1700 could

form the basis of a Recursive Virtual Machine Architecture capable of supporting a general purpose operating system.

CHAPTER 5 - THE ASYNCHRONOUS PROCESSING PROBLEM

5.1 Introduction

In the previous chapters a hierarchically structured virtual machine system has been described which provides all the desirable features of a computer system as defined in chapter three. In order to make 'real' use of this system some asynchronous processing facilities must be provided without affecting any of the 'desirable features' already available.

When the problem of providing asynchronous processing is considered in general terms, the asynchronous interaction of any operation, at any level in the hierarchy, with any other operation must be studied carefully. It is a requirement that each virtual machine operation appears as an atomic entity, but some 'programmed' operations which may be provided, a 'sort' operation for example, clearly will be provided as a further sequence of 'atomic' operations. Obviously therefore, in reality all operations cannot be truly indivisible. In particular if the machine operations of 'add', 'load', etc., are examined at a less abstract level than that of the machine hardware as seen by the users of the system, and the system is considered at a micro-code level, then even those operations normally considered 'atomic' are split into further operations.

In this manner, from the basic transistor level of a computer system, right the way through progressively more abstract levels, to the level of a particular user interface, a set of virtual machines is provided in all computer architectures.

In fact it is argued that by strictly regarding each level of a computer system as an increasingly more abstract virtual machine, all operations, be they I/O operations, complex 'programmed' operations, across-level procedure calls (eg the conventional 'supervisor call'), or the basic machine operations, all can be considered in the same manner.

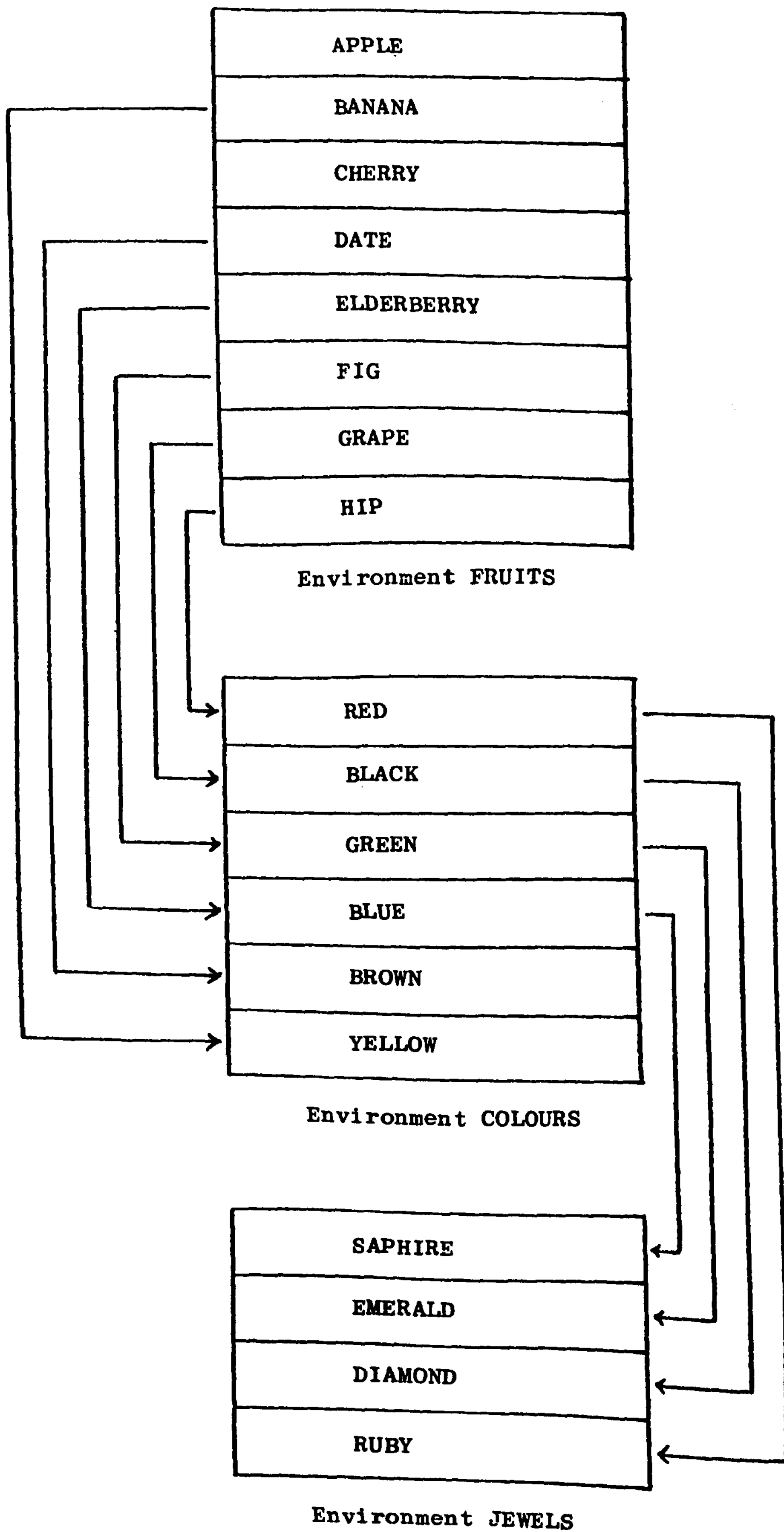
For these reasons the asynchrony problem is discussed in the general terms of two co-operating processes executing asynchronously. It is also assumed that, in general, any parameters passed between the two processes cannot always be of type 'value' or 'result' and that there is a requirement for the processes to share data in such cases. For example, consider a process which sorts data into ascending order, then the only practical manner in which the data can be passed to the 'sort' program is as a shared segment; using ALGOLW terminology, 'value' and'result'parameters will prove highly inefficient in this case.

This approach, of discussing asynchrony in general, has the advantage that it avoids any pre-conceived notions as to how Input and Output processing could be included in a particular computer system. Also if it is shown that a problem exists in the general case then a solution to the general problem must be found if a viable computer system is to be produced. Furthermore if asynchronous processing can be provided in the general case then it becomes logically trivial to provide asynchronously executing I/O processors.

In this chapter the problem of providing asynchronous processing in virtual machine systems is examined. Also, capability systems are examined, since a dual between protection and virtual machine systems was established in chapter two. This examination of protection systems establishes the

mechanism which permits asynchronous processing in these systems and shows that the introduction of 'revocable' capabilities will illustrate a similar asynchrony problem.

Object Mappings Between Environments



SAPHIRE → BLUE → ELDERBERRY
EMERALD → GREEN → FIG
DIAMOND → BLACK → GRAPE
RUBY → RED → HIP

Figure 5.1

5.2 Virtual Machine Systems

In chapter two several virtual machine systems were examined, namely VM370, the RVM, Hardware Virtualizer and Virtual Machine Monitor systems. Since it was concluded that all these systems exhibited identical properties it is only necessary to discuss the asynchrony problem with respect to one of these systems, and the RVM will be used for this purpose.

The only other implementation of a hierarchically structured virtual machine system, VM370, has been shown to have problems with asynchronous input and output operations. However, because of the extra complexities of this system, due to the basic two-state nature of its underlying architecture it is inappropriate to use this system in these discussions.

5.2.1 Asynchronous Processing in the RVM

Within the RVM a mapping function is supplied which provides objects in one virtual machine in terms of objects in the next less abstract virtual machine at each level in the hierarchy. It is therefore possible, see figure 5.1, for any process providing the mappings of objects at a more abstract level to rename these objects in its own terms. Eg a process within Fruits could change the mapping RED → HIP into RED → CHERRY.

This ability for a process to redefine any descendant environment causes a serious problem if one of the objects being renamed is being shared with a process in an ancestor environment to that causing the redefinition. Consider, for example, the following two situations both of a similar nature, the second highlighting the extreme seriousness of the problem.

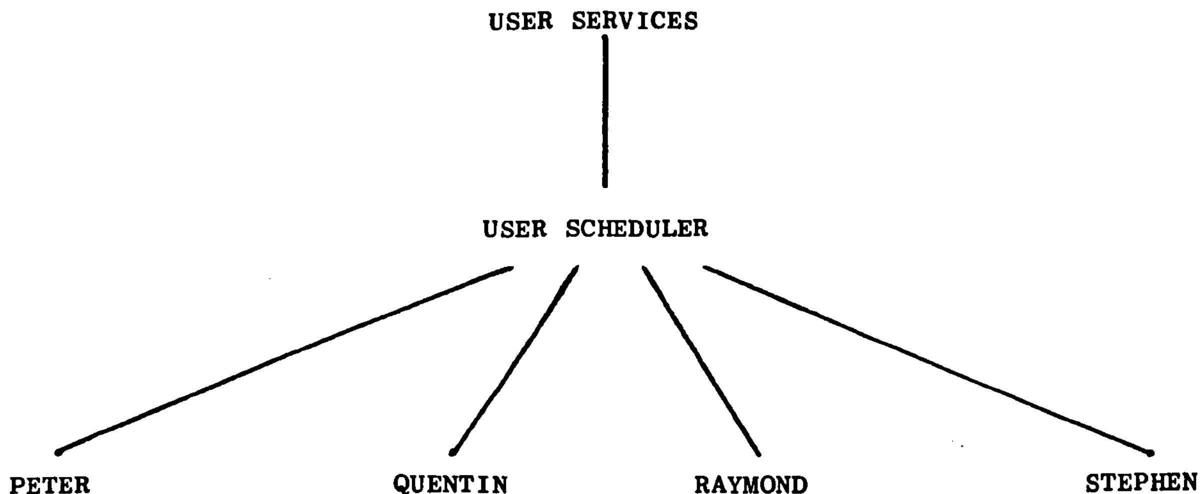


Figure 5.2

i) Referring to Figure 5.2, a process within the environment PETER may invoke a 'Translate ISO to EBCDIC' operation on some data 'File Record', the operation being provided by a process within the environment USER SERVICES. The process within USER SERVICES is now manipulating the object'parameter'in its terminology, this object having been mapped from the object 'file record' within the environment PETER. Concurrently a process within USER SCHEDULER might decide that it is time RAYMOND was permitted to execute and the objects constituting PETER may be moved onto backing store thus endangering the 'Translate ISO to EBCDIC' operation. The problem arises because the process within USER SERVICES can only refer to the data, 'File Record', in its own terms, and the process within USER SCHEDULER, being unaware of USER SERVICES dependence on this object, changes its use. Processes within USER SCHEDULER have insufficient knowledge to know when it is safe to manipulate the resources of PETER and thus this may cause a failure in the protection system .

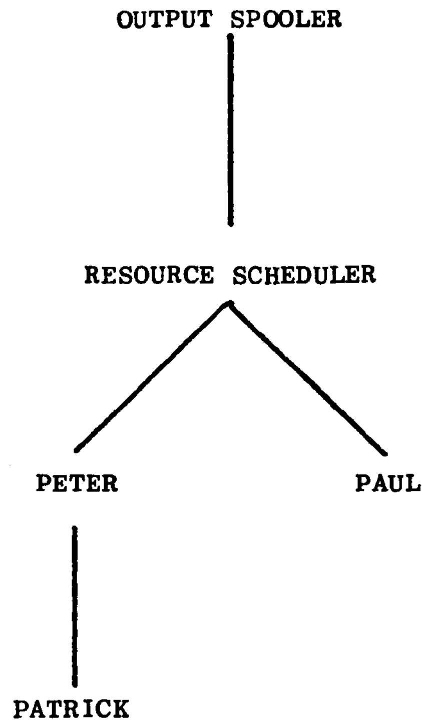


Figure 5.3

ii) Referring to Figure 5.3, a process within PETER creates a subsidiary environment PATRICK containing the segment OUTPUT FILE which is to be spooled by a process within OUTPUT SPOOLER. The spooling operation, requested by the process within PATRICK, requires the use of the resources named OUTPUT FILE, which in turn are named FILE in the environment PETER and Disc in the environment RESOURCE SCHEDULER. The process within PETER may then inform a process within RESOURCE SCHEDULER that it has finished with the resource FILE, although the OUTPUT SPOOLER is still accessing this same resource on behalf of PATRICK. The called process within RESOURCE SCHEDULER may, in ignorance of the spooling operation, make use

of the resources DISC itself or it may pass them to a process within environment PAUL. This will thus mean that information may be copied from PETER to PAUL or RESOURCE SCHEDULER enabling PETER to sabotage these environments. Alternatively if input spooling was being requested by PATRICK then PETER may be able to read confidential information belonging to PAUL or RESOURCE SCHEDULER. In either case a serious breach in the protection system is exposed.

If the RVM system is to remain 'well-protected' as defined in chapter one, while providing the ability for processes to execute asynchronously, it must prohibit the renaming of objects while they are required by the 'called' environment. The solution whereby control is passed through each successive environment can be dismissed on grounds of inefficiency, for exactly the same reasons that the CAP inter environment process migration scheme, discussed in chapter two, was regarded an inefficient.

As a result of these discussions two important questions arise:-

i) How is a process within one environment, eg RESOURCE SCHEDULER, to be prevented from (unwittingly) re-allocating the resources currently being shared between one of its ancestor environments, eg. OUTPUT SPOOLER, and one of its descendant environments, eg. PETER ?

ii) How can the RVM assure a called process within some less abstract environment, eg USER SERVICES, that the resources it is manipulating on behalf on the calling process will remain stable and accessible to the calling environment once they have been located ?

Clearly it is highly inefficient to expect the called environment to check the validity of the shared resources prior to any access of them. Even to achieve such a check would require that the called process was non-interruptable while performing the check and this point contradicts the premise that if asynchronous processing is to be permitted then the mechanism must be able to be invoked between any two primitive RVM instructions.

The notion that environments between called and calling process must be informed of the requirement that certain resources must remain stable is invalid. Only occasionally are resources reallocated. Preferably the overhead of ensuring that the resources are not involved in a concurrently executing operation should be incurred at the time of re-allocation rather than as each operation is invoked.

5.3 Capability Systems

Having discussed the reasons for the problem of providing asynchronous processing in virtual machine systems, capability systems will be examined in order to discover the reason why these systems seem to provide an effective asynchronous processing mechanism.

It is interesting to note that although the basic reason why capability systems permit asynchronous processing is the same in all cases, a similar problem to that exposed in virtual machine systems is exibited by GEC 4000 systems when other processors are involved, and also in relation to the use of capability lists and extended core storage in the CAL-TSS system.

This basic reason why capability systems permit asynchronous processing is the fact that once a process has been granted the capability for some resource, it cannot be revoked by any other process in the system. This philosophy is common to all the capability systems discussed in chapter two, and is described by the following example.

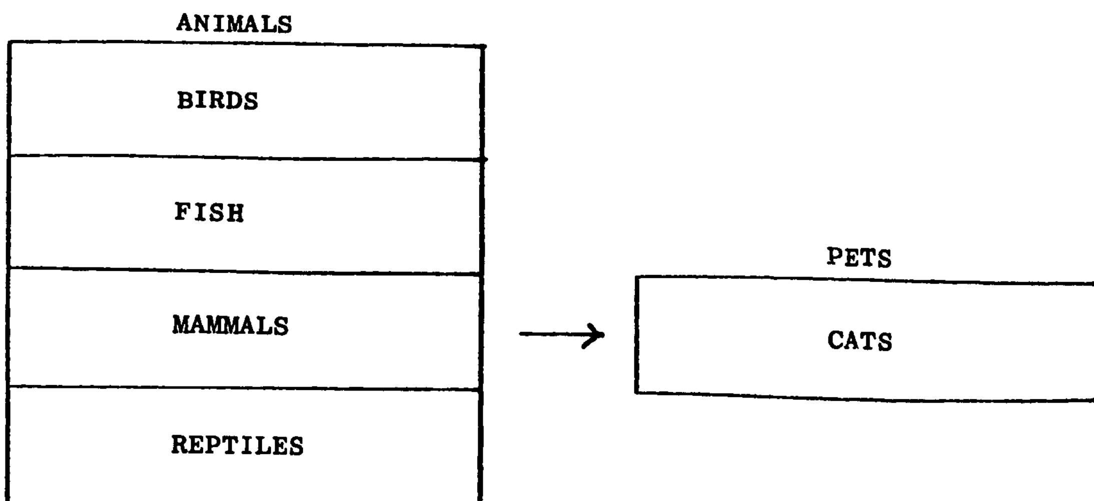


Figure 5.4

Referring to Figure 5.4, if a process within the environment ANIMALS passes the resource MAMMALS to a process within the environment PETS. This resource may be named CATS in the environment PETS. Because it is impossible for any process to revoke PETS' capability to the resource CATS, a process within ANIMALS cannot even replace the resource CATS with a new resource DOGS.

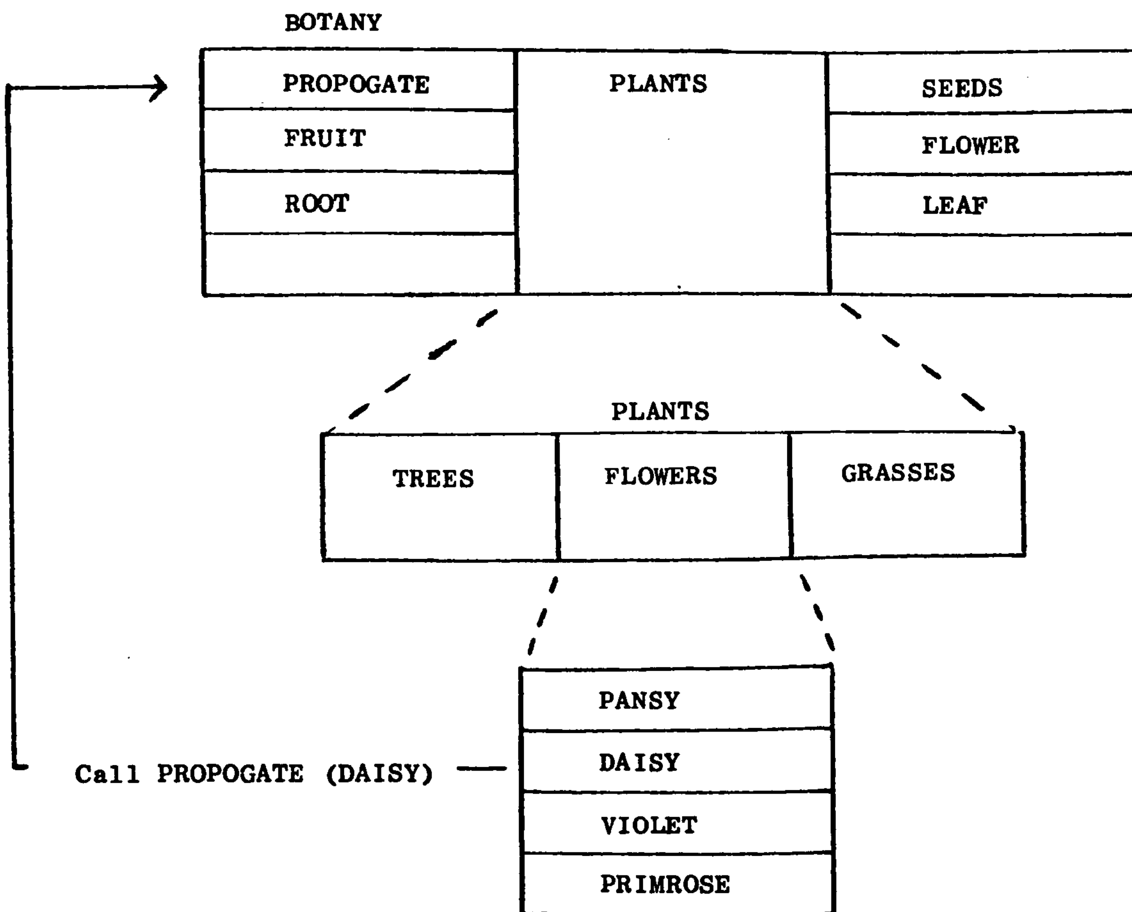


Figure 5.5

The ability to revoke access to resources is fundamental to virtual machine systems, however the lack of revocation permits processes to execute asynchronously within a capability system. For, referring to figure 5.5, consider a process within the environment FLOWERS which passes the resource DAISY to a process within the environment BOTANY requesting the operation

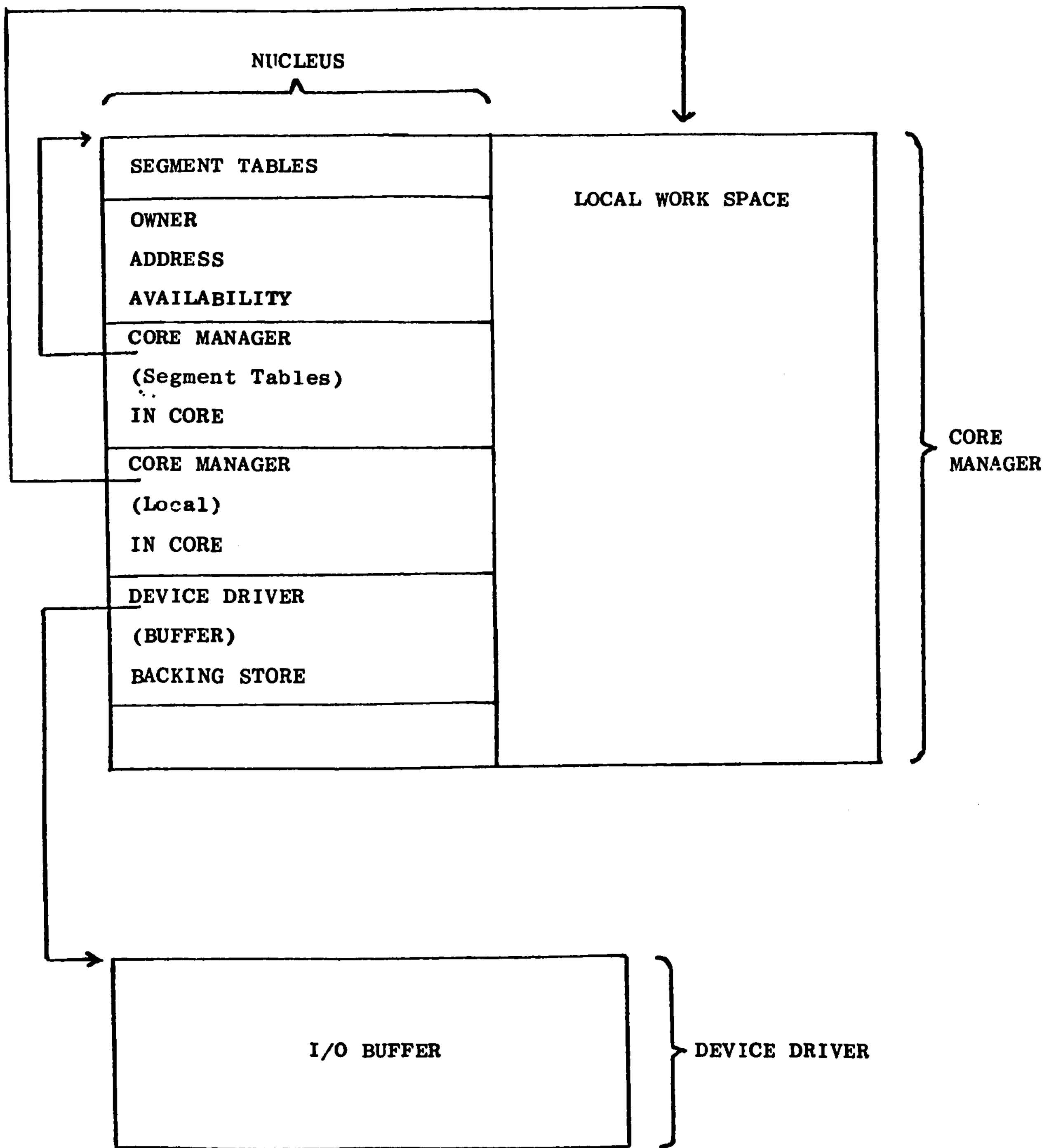


Figure 5.6

PROPOGATE. Within a capability system it is impossible for any process within the environment PLANTS to change the resource DAISY in any way, and the process within BOTANY is assured that DAISY will remain stable for the duration of the PROPOGATE operation.

It is postulated therefore, that if a system can be constructed which never requires to revoke access to objects then this will permit asynchronous processing within a well-protected computer system. Further study of the GEC4000 system and the CAL-TSS system reveals that at some level an element of revocation exists and in the HYDRA system a revocable capability mechanism has been proposed [CJ 75] .

5.3.1 The GEC 4000 System

The GEC 4000 system ensures that the 'Nucleus' within the central processor is always aware of the availability of each segment within the system, ie. in core or on backing store, see Figure 5.6. Any attempt to access a segment currently backing store will cause the Nucleus to schedule the CORE MANAGER in order to retrieve the necessary segment. Because of this approach, any process which is directly controlling an I/O device must ensure that any segment to be shared with the I/O processor is 'held' in core for the duration of the I/O operation. Consequently the I/O Device Driver must inform the CORE MANAGER if it is known that the shared segment is eligible for removal to backing store.

This situation arises because the CORE MANAGER could decide to move the shared segment during the I/O operation , thus causing the I/O processor to overwrite a segment now belonging to a different process.

In the case of the GEC 4000 this solution is considered satisfactory, for the architecture is non-hierarchial and only the more privileged environments contain I/O device drivers. However the RVM does not lend itself to such a solution since this would require any segment being shared to be 'held' prior to each operation that might possibly cause some asynchrony in the system. If a general solution is to be provided it must be capable of being utilized between each primitive machine operation, and only rarely in this case will objects require locking in memory.

5.3.2 The CAL-TSS System

In the description of CAL-TSS in chapter two it was mentioned that once a capability for an object has been passed to another domain it cannot be revoked. Also that the CAL-TSS designers felt that such a feature could not be implemented in he system, partly because of the global name space approach which has been adopted [Lam 68, LAM 69b].

More recently a paper by Lampson and Sturgis [LS 76] discusses the actual CAL-TSS system finally implemented and the problem areas that were met while developing the system. The CAL-TSS system is implemented as a layer on top of the software kernel and was designed to have the following properties :-

i) User system code executes within environments of kernel processes.

ii) User objects in extended core storage are represented as kernel objects. This is so that frequent actions on user objects can be implemented as the more efficient kernel actions on the representing kernel objects.

iii) The movement of user objects from extended core storage to the disc is performed by using kernel operations which read the state of the kernel object and set up the description of the state of the represented user object. The user system can then write the description on the disc, by the appropriate use of kernel operations.

A major difficulty arose while trying to satisfy the third property, in particular with regard to capability lists, as these could not be represented on the disc. The only objects which could be given a disc representation were those which had no direct kernel representation. This is illustrated by discussing the movement of a capability list from disc to extended core storage. The representation in extended core storage is to be by a kernel capability list, the user capability list contains capabilities for various objects, however some of these objects may have existing kernel representations, others do not. Further there may exist other user capability lists which contain capabilities for the user capability list which is being moved and which already have a kernel representation. Upon attempting such an operation one of the kernel actions must be to create the new capability list, however with the kernel that was constructed it was not possible for a capability in a pre-existing kernel capability list to point to this new kernel object, nor for a capability in the new capability list to point to a non-existent kernel object.

As a result of this problem a compromise was reached, whereby user objects were divided into classes, those which could be moved in the manner indicated and those which could not. However, as development of the system proceeded other problems arose because two classes of objects had been used.

This problem is very similar to that of providing asynchronous processing within the RVM, and indeed an initial attempt at a solution, within the RVM, was made by grouping different classes of objects together in a similar manner to that just described. Segment table entries could not be moved and other objects could. This was considered to be a very inflexible and expensive way of providing a solution to the asynchronous processing problem since there are occasions when a process will wish to alter the segment table entries of a descendant and the philosophy of the RVM is to treat a segment table as 'special' only for the environment which it defines. Furthermore there was no guarantee, as discovered in CAL-TSS, that such an approach would provide a comprehensive solution.

5.3.3 The HYDRA System

The fact that capability systems do not permit capabilities to be revoked provides the mechanism for allowing asynchronous processing in these systems. The designers of the GEC 4000 and CAL-TSS systems do not regard the lack of revocable capabilities as a serious problem in their systems, however it has been demonstrated that where an element of revocation is involved then asynchrony problems are highlighted in both cases.

Cohen and Jefferson [CJ 75] discuss the revocable capability in relation to the HYDRA system, and although they do not regard it as serious they do propose that a mechanism is introduced which permits the revocation of capabilities and another is introduced which prevents it.

The revocable capability problem in HYDRA is regarded of low importance since many other HYDRA mechanisms, associated with the lack of hierarchial structure, eliminate the need for such a feature. For this reason it is

not considered essential that the HYDRA revocable capability mechanisms are highly efficient. This is in direct contrast to the RVM system where revocation is an essential feature and an integral part of the segment table mechanism which must be maintained as efficient as possible.

5.4 Revocable Capability Mechanisms

The importance of providing a facility which permits capabilities to be revoked from a more abstract environment is a matter that has been discussed for some time. Fabry [Fa 74] , in a paper which discussed possible hardware implementations of a capability based addressing scheme, also discussed the reasons for wishing to allow capabilities to be revoked, and from this discussion a proposal for a revocable capability arose. Redell and Fabry [RF 74] discuss a mechanism which permits a process to create an environment defined by a capability list, and this capability list may contain a combination of actual capabilities and pointers to capabilities which can be used by the environment. The pointers and their associated capabilities are defined as revocable capabilities.

Within a conventional capability system a single level of indirection is involved when mapping an object in one environment into an absolute resource of the real machine. This single level of indirection is provided by the capability associated with the abstract object. Within the revocable capability scheme further levels of indirection may be introduced as required, these being provided by the pointers to the actual capability for the object involved.

Both ordinary and revocable capabilities, when mapping from capabilities to objects, are considered in the same fashion by any process. Each process exists within an environment which is defined by a capability list containing either or both types of capability. Such a process has the ability to set up further environments with capability lists constructed of either ordinary or revocable capabilities. Thus as each more abstract environment is created there is the possibility of adding a level of

Example of Revocable Capabilities

Environment JOHN Defined by Capability list containing ordinary capabilities.

GRANDAD	TOM
FATHER	DICK
UNCLE	HARRY
BROTHER	BERT

Capability List

TOM	DICK
HARRY	BERT

Environment JOHN

Environment HARRY defined by Capability list containing a revocable capability for NEPHEW.

FATHER	TOM
BROTHER	DICK
NEPHEW	BROTHER'S SON
DAUGHTER	BETTY

Capability List

TOM	DICK SON: JOHN
JOHN	BETTY

Environment HARRY

Environment BETTY

GRANDAD	FATHER'S FATHER
FATHER	HARRY
UNCLE	FATHER'S BROTHER
COUSIN	UNCLE'S SON

Capability List

HARRY FATHER: TOM BROTHER: DICK	TOM
DICK SON: BERT	BERT

Environment BETTY

Figure 5.7

indirection by making the capability to a new object revocable, the capability for the object in the parent environment being either ordinary or revocable, see Figure 5.7. It is thus possible to introduce revocable capabilities defined by revocable capabilities repeatedly at each level in the system thus a very similar structure to that defined by the RVM segment table mechanism in constructed. Of course it is also possible to pass on ordinary capabilities between environments in the system repeatedly. Using this technique asynchronous processing could be performed as in a normal capability scheme.

The problem which has been exposed in the implementation of the RVM, that of preserving the mappings of objects between several environments in the system for significant periods of time, does not appear to have been considered in the design of revocable capabilities. Redell [RE 74] , in his thesis, goes to great lengths to illustrate the flexibility of the scheme for supporting and creating objects of different types and briefly demonstrates, in a trivial example, that some asynchronous processing is possible. However a closer study of the more generalized problem reveals that exactly the same problems occur as in the RVM.

Example: Consider the environment structure of Figure 5.7. The three environments are non-hierarchial in nature and each environment shares objects with other environments. If a process within the environment BETTY starts to perform some operation involving data copying into the object COUSIN then this may involve the reading from or writing to the object BERT. Asynchronously with this operation a process in the environment HARRY may decide to rename DICK's SON as JOHN. This may cause the object JOHN to be overwritten. As a result the process within

BETTY may be corrupted or any other process with access to JOHN could be forced to divulge privileged information or to crash.

As in the RVM, the problem arises because the process within HARRY which supplied the capability COUSIN to BETTY, is unaware of BETTY'S dependance upon this remaining constant.

5.5 Conclusions

In this chapter the problems of providing asynchronous processing in virtual machine systems and capability systems have been discussed. It has been established that the lack of a revocable capability permits processes to execute asynchronously in a capability system but it is questioned whether a real system can be built which does not require revocation at some level. The GEC 4000 and CAL-TSS systems illustrate that an element of revocation exists at some level of the system and the HYDRA system now permits the use of revocable capabilities.

An investigation of the revocable capability mechanism demonstrates that the complete interchangeability of both ordinary and revocable capabilities causes a replication of the asynchronous processing problem encountered in the RVM. A process at any level of the hierarchy cannot know if a process within one of its subordinate environments is dependant upon a particular capability it may wish to revoke.

In the following chapter a mechanism is proposed which overcomes the asynchronous processing problem in the RVM. It is expected that this mechanism will prove generally useful and extendible to revocable capability systems, thus ensuring the integrity of object mappings in both cases. A key requirement of this mechanism is to prevent or delay revocation during use by a subsystem, and this is the point at which the HYDRA mechanism fails. Of course direct use of the RVM mechanism to be proposed will not be possible in most revocable capability systems due to their non-hierarchical structure. This, however, should not prevent an extension to the mechanism permitting it to be employed in both virtual machine and capability based systems.

CHAPTER 6 - A MECHANISM TO PERMIT ASYNCHRONY IN THE RVM

6.1 Introduction

Having described in the previous chapters the mechanisms and implementation of a synchronous RVM, and discussed the problem of providing asynchronous processing, it remains to examine how asynchrony might be permitted in the RVM. This chapter describes a mechanism designed to permit asynchronous processing, insisting that this mechanism must prove sufficiently efficient to be used for each operation. Further, the mechanism must permit any two processes within the RVM to interact in a sensible asynchronous manner so that none of the original design criteria, as described in chapter three, are broken.

Initially the criteria for providing a solution to the problem are discussed. A mechanism is then described which conforms to these criteria. Finally the efficiency of the mechanism is examined and some refinements which could further increase the efficiency of the scheme are described. Fundamental to this chapter is the criteria that any such mechanism which permits asynchronous processing in the RVM must prove sufficiently efficient to be utilized for every operation. For if this criterion is fulfilled then the mechanism can be used in all cases where asynchronous processing may be involved. Thus including multiple processors and 'supervisor call' type operations.

6.2 Requirements for a Solution

Consider the environment structure illustrated in Figure 6.1, and the requirements necessary to ensure that the system is well protected. For the system to perform consistently and reliably it is sufficient that a process contained in one environment (V say), having an operation fully interpreted in an environment at a less abstract level (C say), must have the mappings between the intervening levels preserved for each resource which is required by the process at the less abstract level, eg the mappings P→V and E→P. It may be desirable to preserve the mappings C→E but it should be observed that any attempt to change this mapping by a process within environment C can be regarded as an error in the process which is providing the requested operation, since such a process must be aware of the resources it is currently manipulating.

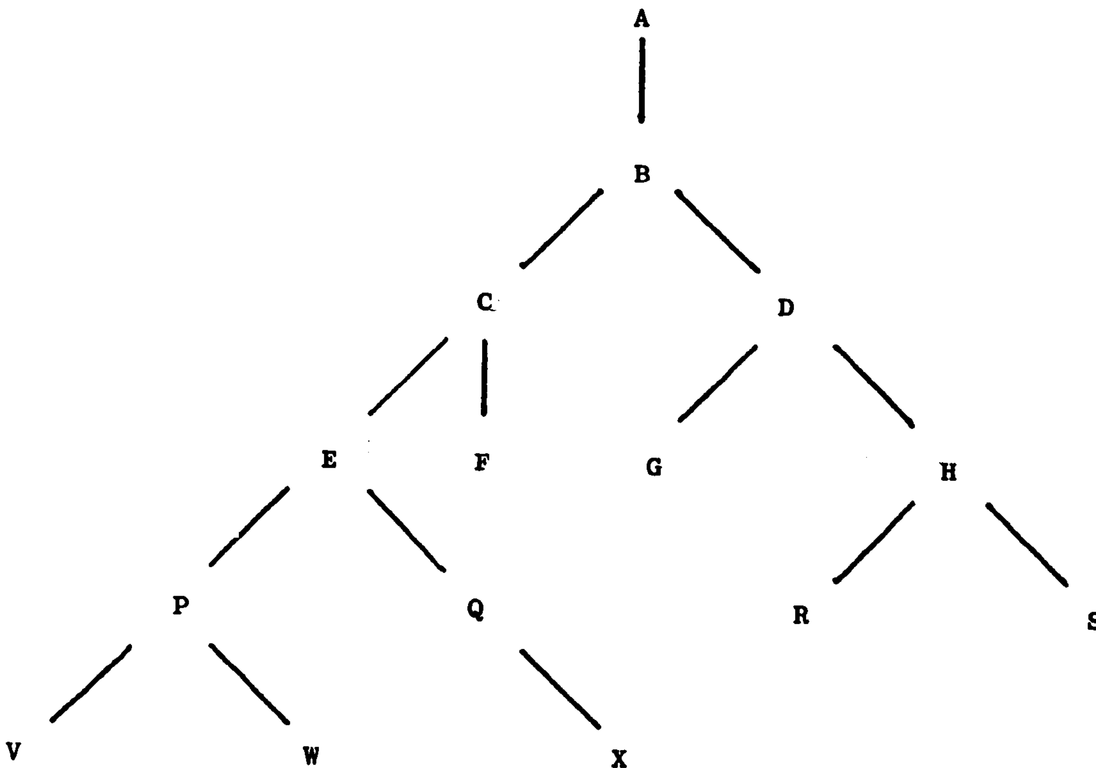


Figure 6.1

A requirement such as this can be fulfilled by preventing any further processing in the sub-branch of the tree which contains the calling process. For example, environments E,P,Q,V,W and X are locked out while an operation is being provided for V in environment C. Processes within environments R,F,G,H etc could therefore continue processing asynchronously with that in C, there being no possibility of the shared resource mappings between C and V being corrupted. However as it is the principal requirement that these mappings be protected for each operation, including those of the bare machine at level Ø, then clearly very little processing could take place in such a system.

As already discussed, the protection of the shared resource, against its overwriting, or the assurance of it remaining constant, can be achieved by conventional 'semaphore' or 'monitor' techniques. Thus the protection of the resource itself will not be considered as the domain of a hardware protection scheme. Instead it is a requirement that the process, within the environment which is providing an operation, performs any necessary protection of the shared resources. Thus two or more processes will be permitted to communicate via a common data area without any loss of protection. Should a protection violation occur then it is asserted that this can only propogate as far as the process, in the least abstract environment, which is sharing the common resources.

For example, referring to figure 6.1 if processes within X and C are co-operating via some shared data, then incorrect use of this data cannot cause a fault in any process within A,B,or D and its sub-branches.

Thus if a process in one environment wishes to co-operate with a process in a more abstract environment, it must ensure that this process co-operates in an orderly manner, by the use of 'monitors' for example, or accept the consequences of any error that may occur within the subordinate process.

C1
C2
C3
Segment Table
Defining Environment C

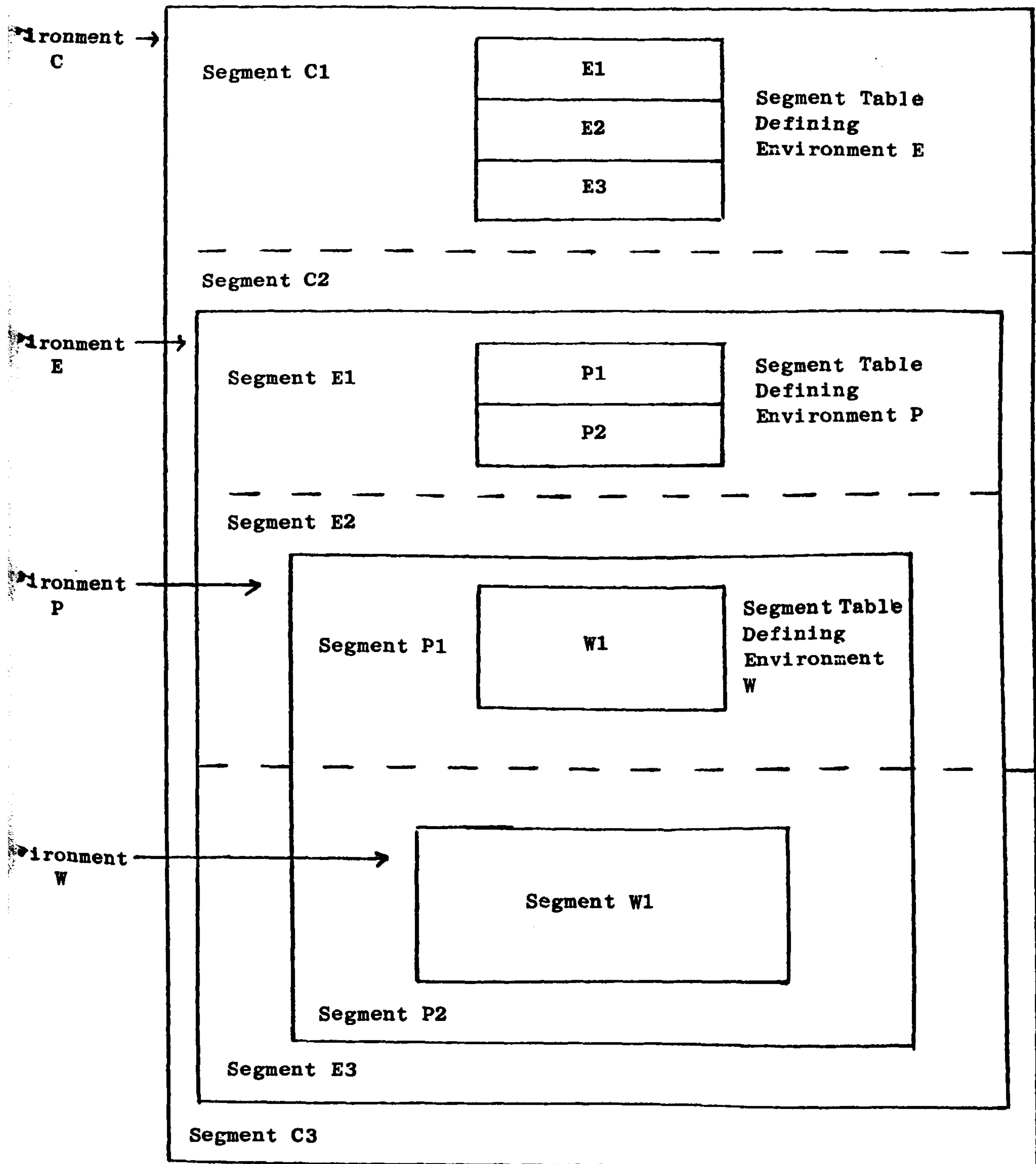


Figure 6.2

These notions of requirements necessary for ensuring a well-protected system are fairly intuitive, and it is easy to see that if these conditions can be maintained then the problems discussed in the previous chapter will be resolved.

At this point it should be noted, again referring to Figure 6.1, that while V is having an operation performed at C, a process within B is permitted to remove a common resource from C (and its descendants). Such an action is quite acceptable, for although C may be executing on a separate processor and will inevitably produce a fault when it next attempts to access the resource, since B has removed the resource it is able to replace it, even mapping it into different resources of its own environment if appropriate. This will not affect C's operation in any way, since the resource shared between C and V will still be named consistently in terms of both C and V.

A mechanism which is incorporated into the RVM in order to provide asynchronous processing must therefore fulfil the following requirement. The mechanism must protect the mappings of any objects required by a process within an environment at one level of the hierarchy, which is interpreting an operation for some process in an environment at a more abstract level. Furthermore since the intervening environments know nothing about these actions this protection must be provided automatically.

Example: Referring to Figure 6.2, if a process within environment W requests an operation, on a set of objects W1, which is provided by a process in environment C then the following mappings must be preserved.

i) W1 in environment W into P2 in environment P

ii) P2 in environment P into E3 in environment E

iii) the segment table defining environment W, contained in environment P, into its realization in environment E,ie P1 in environment P into E2 in environment E.

These mappings must be preserved automatically since processes within environments P and E are unaware of the current operation being undertaken for the process in environment W.

6.3 Considerations of Efficiency

In order to satisfy the requirements of the previous section, it might be envisaged that an 'indicator' be maintained, for each basic machine object, in the micro-program of the base machine. This indicator could enable the micro-program to establish whether or not the contents of a particular basic machine object could be changed by a process executing within any environment of the system.

A call of a process in a less-abstract environment would thus cause certain objects to be identified as 'locked' to processes in intermediate environments for the duration of the call. Such an approach will permit a sensible level of asynchronous processing, however a closer examination reveals that the cost of providing such 'indicators' will prove prohibitively expensive if they are to be used at the start and end of each operation.

The major reasons for this high cost are two-fold. Firstly the extra storage required to implement such a scheme is of the order of a basic machine object. Each indicator must contain information regarding the type of the object (segment table or data), the number of environments which require this item to remain constant, and which environment in particular has access to the item.

A more important reason for rejecting this approach is the fact that the cost of setting up these indicators is not linear with respect to the distance between called and calling processes. The cost function is, in fact, exponential and this stems from the fact that setting an indicator on a segment table entry for a resource at one level implies that two segment table entries require manipulating at the next, less

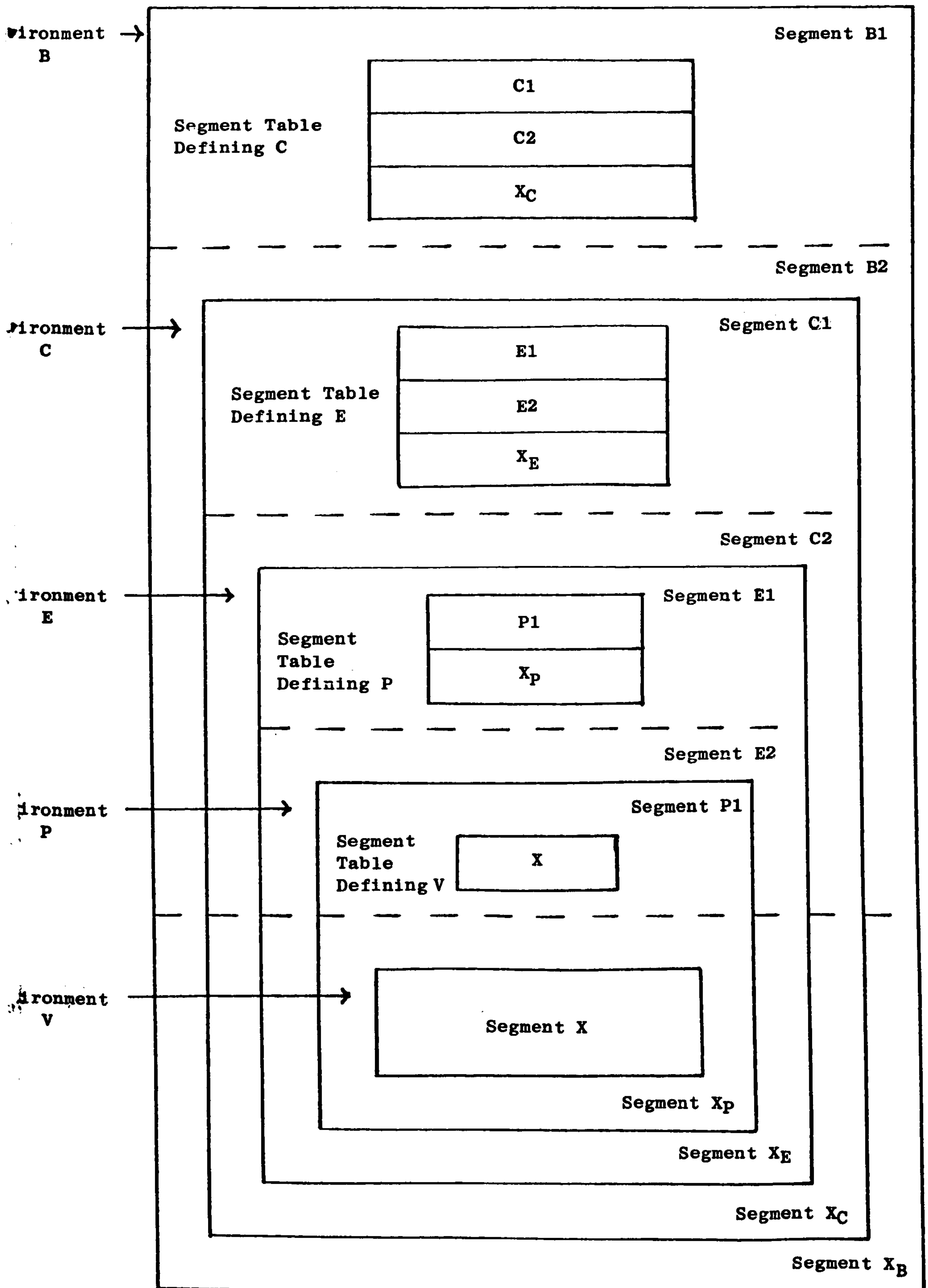


Figure 6.3

abstract level. One of these entries will be mapping the resource in question, the other will be mapping the resources containing the segment table for the previous level.

Example: Referring to Figure 6.3 consider the information which requires to be preserved for the duration of an operation which is being performed in environment B for a process in environment V, using the shared resource X.

i) Segment table entry in P for X must not be changed,

ii) Segment table entry in E for X must not be changed,

iii) Segment table entry in C for X must not be changed,

iv) Segment table entry in E for P's segment table for V, must not be changed, ie segment P1,

v) Segment table entry in C for P's segment table for V, must not be changed, ie segment E2,

vi) Segment table entry in C for E's segment table for P, must not be changed, ie segment E1.

ie six items must be protected when interpreting an operation at four levels of abstraction less than the requesting process, and sharing only one set of resources.

In order for the mechanism to be efficient it has been asserted that the cost of its utilization must remain linear with respect to the distance between any two environments containing processes which utilize the mechanism. For this reason a different strategy must be adopted.

One further point, which indicates the unsuitability of protecting individual resources, is that any resource which requires protecting for the duration of an operation can only be protected as it is identified. Such a scheme may cause the overall protection of the system to become exposed, since it may be possible for a process within an intermediate environment to change the mapping of an as yet unidentified resource unwittingly. The proper approach is to provide this protection during the call of each operation, and this can be provided if the complete environment is protected, rather than the individual resources contained within the environment.

6.4 The Proposed Mechanism

If the example of the previous section is returned to, and if, rather than attempting to protect the resource X for the duration of an operation performed in environment B, the whole environment V is protected then the following items need protecting.

Referring to Figure 6.3:-

i) The environment V ie segment X

ii) The environment P ie segments P1 and X_p

iii) The environment E ie segments E1, E2 and X_E

Thus the number of items requiring protection is now directly proportional to the distance between the two communicating processes.

Clearly this is a superset of the condition for providing asynchrony given in section 6.2. The segment tables for any environment are contained in the environment at the next, less abstract, level and are thus always protected by this approach.

One further advantage of such an approach is the fact that the protection can be provided during the call of a process at the less abstract level, and once provided it remains valid until the completion of the call. Therefore there is no requirement to provide the protection for each shared resource as it is identified.

A mechanism which protects the environments by-passed in an across level call is therefore proposed. The fact that the number of items to be

protected is linear with respect to the distance between called and calling environment plus the fact that this protection need only be applied once indicates that such an approach will conform to the efficiency criteria defined in chapter one.

The following sections describe the extra information to be maintained by the RVM interpreter in order to implement such a mechanism. This extra information falls into four sections as follows:-

i) Information required to ensure an environment remains protected for the duration of an operation.

ii) Information defining the route from a calling to a called process

iii) Extensions to the Associative Memory mechanism of the synchronous RVM

iv) Table identifying those environments in the system which are protected.

6.4.1 The Environment Protection Semaphore

Since any environment may be required to be protected by several asynchronous processes, a multiple reader/single writer semaphore, of the type described by Courtois et al. [Co 71], must be associated with each environment in the current system. The 'writer' semaphore indicates that a process wishes to change, or is changing, the segment table defining a particular environment. The 'reader' semaphore indicates the number of processes currently dependent upon the environment remaining constant. Thus an environment can only be redefined when the number of 'readers' is zero; further no 'readers' can become dependent on the segment table while the 'writer' is non-zero.

In this way any attempt to change a segment table which is providing the mapping which defines a protected environment will not be permitted, also any attempt to protect an environment which is having its mapping changed will be suspended until the completion of the operation which is causing the mapping to be changed.

6.4.2 The 'Called Route' Segment

On completion of a particular inter-environment 'procedure' call, in order to release the semaphores on all the environments protected for the duration of the call, a list of the 'route' between called and calling environments must be maintained. However, this information is already recorded in the synchronous RVM implementation as a stack of environment descriptors; and this is done in order that control can be returned correctly to the calling environment.

The positioning of this information is solely concerned with any implementation of the mechanism. However it is likely that some alternative to the global scheme employed in the synchronous RVM implementation will have to be used. The main reason for this difference is the possibility of a variable number of asynchronous operations outstanding at any instant. In addition, since across-level procedure calls are considered as an extension of general asynchronous operations, it will be impossible to associate this information with a particular processor in the system.

It is therefore proposed to introduce a new segment associated with each environment which is created. This segment could be named - 1, and made inaccessible to any process executing within the environment, and would contain a list of the currently outstanding calls on processes within that

B's called route segment when performing operations for both F & J. (-1 indicates the end of a list)

Reference	Environment	Next in route
1	C	2
2	F	-1
3	E	4
4	J	-1

ie the route C,F is maintained as Reference 1

and the route E,J is maintained as Reference 3

Environment Structure for above example:

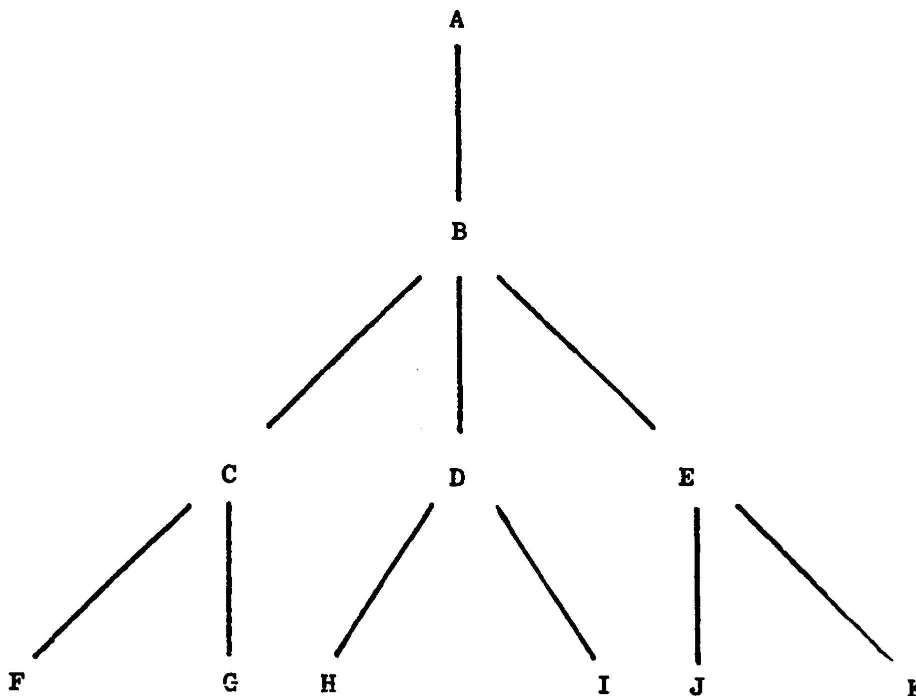


Figure 6.4

environment. Again it will probably prove most appropriate to store the complete route of called and by passed environments, so that when the requested operation has been performed, control can be returned to the calling process.

Example: Referring to Figure 6.4, consider a process within an environment F which requests an operation provided at B. B's 'Called Route' segment would then contain the environment names F and C. While this operation is being performed a process within J may request another operation also provided at B. B's 'Called Route' Segment would then contain the environment names F, C and J, E.

Clearly there is a requirement to associate each call from a descendant environment with the 'called route' which has just been saved. Consequently on request of an operation it is proposed that the called process is supplied with a reference to the 'called route' so that a successful return can be established when appropriate.

By storing the 'called route' information in a segment inaccessible to the called environment, called processes would be protected against overwriting the return routes. Of course a parent could permit one of its subordinates to access this routing segment. However, an action of this kind can be regarded as an error in the parent process. It will not permit the propogation of any error to a process within a less abstract environment than that of the parent process.

A further feature of such a scheme is the flexibility it affords. Although it is necessary for each parent process to provide a segment in order to contain this routing information, it may be of variable size,

even a null segment depending upon how much control the parent process wishes to exercise over processes within a subordinate environment.

Having established a scheme for protecting environments while any operation is being performed it must be noted that the process providing the operation will have to map resource names in the calling environment into resources of the called environment. If the operation is being provided by the hardware of the bare machine then this mapping is performed automatically by the micro-code. However in order for a process in some abstract environment to calculate, in its own terms, the names of resources it needs to manipulate then it will have to perform this mapping iteratively by using the segment tables defined in the segment containing the 'route' of called to calling environments.

The major problem here is that although a process in the calling environment requests some operation on a set of resources X, these are unlikely to have the same name in the called environment. Thus when the process within the called environment realizes that the resources to be used are called X in the calling process's terms, the called process must determine from the intervening environments the name of X in their terms in order to determine the name of X in its own terms.

As already noted this procedure is available in the micro-code of the RVM in order to map names in any environment into objects of the bare machine. Clearly a similar function is required to map the names of any environment into those of the called environment.

6.4.3 Associative Memory Considerations

Essentially the mapping of resource names between any two environments

Item \ Info	Resource Name	Environment	Maps Into Resource	Maps Into Environment	Access Allowed
1	X_F	F	X_E	E	Minimum
2	X_D	D	X_A	A	Full
3	Y_F	F	Z_D	D	Minimum
4	X_C	C	X_A	A	Full
5	X_F	F	X_C	C	Minimum
6	Z_E	E	Z_B	B	Full
⋮					
N-1	X_F	F	X_A	A	Minimum
N	X_E	E	X_C	C	Full

Environment Hierarchy

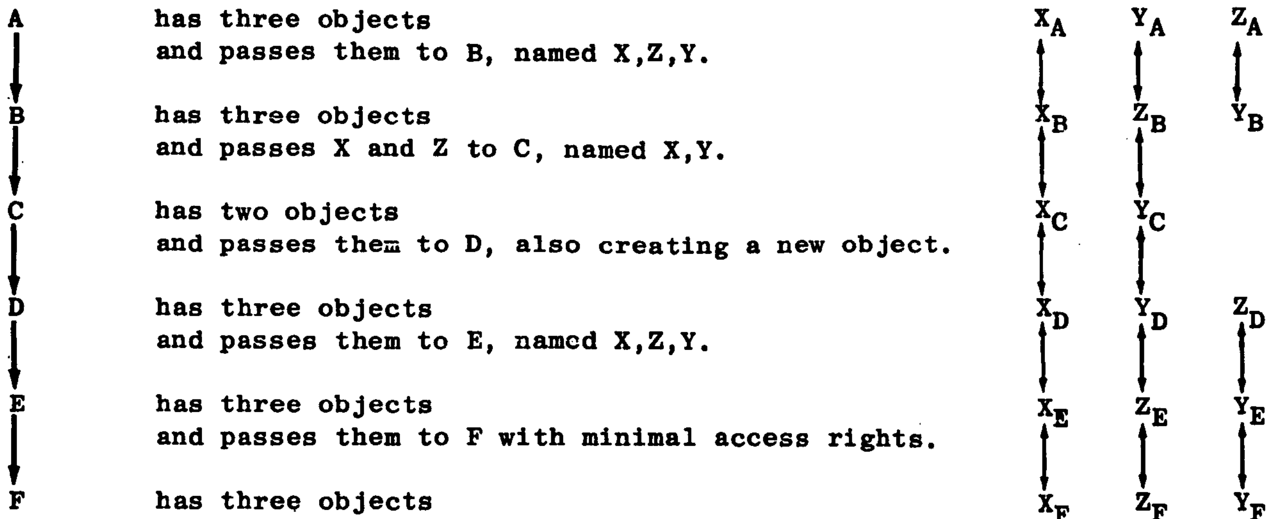


Figure 6.5

is the same. The mapping of resources into those of the bare machine is a special case. The next consideration is therefore the manner in which this information can be accessed both safely and efficiently.

At any given instant the information regarding the various resource mappings required must be available to an indeterminable number of processes. For this reason it must be kept in a separate memory from the total available resources, and the micro-program of the RVM must keep the information updated. Clearly there is too much information for it all to be held in a single high speed store, and there is no obvious way to decide on an allocation policy for placing some of it in such a store. An alternative approach would be to address the information directly but again there is no obvious way this could be achieved.

As already mentioned, in chapter four, the problem of efficiently accessing repeated items, at any level in the hierarchy, from basic objects of the bare machine can be achieved by employing associative memory techniques. Since it has already been asserted that the problem of addressing objects of the bare machine is an essentially similar problem of that of providing address mappings between any two levels in the hierarchy, it is proposed to employ similar associative memory techniques to help solve this efficiency problem. The associative memory will therefore contain the mappings of resources between any two levels in the system, rather than just the mappings between a level and the least abstract one. Also since both operations and objects are mapped between levels of the system, a single associative memory which fulfils this purpose will be discussed.

Figure 6.5 illustrates the format of the proposed associative memory together with a possible set of values and environment structure.

Given the results of the experiments in chapter four, where an associative memory mechanism was shown to have a marked effect on the efficiency of the synchronous RVM, it seems realistic to suppose that by careful selection of the size of a memory which permits the extended functions described, then the mapping of resource names between any two levels could be performed efficiently.

It may appear that the use of such a mechanism might enable a process to overcome the protection scheme enforced by the RVM. For referring to figure 6.5, consider a process in environment F which requests an operation on some object X_F to be performed in environment C. The object X_F may map into the object X_C, to which C has full access rights. If however the process within environment F has only minimal access rights because rights have been denied by an intermediate environment then this information will be maintained within the associative memory, thus preventing the calling environment from overcoming the protection mechanism. The fact that access is now performed on an object in a privileged environment is of minor concern. For even if C is permitted to access X_C in a more privileged manner than F may access X_F, these extra privileges may not be invoked since the associative memory records the maximum permitted access to any object as it is mapped between environments.

Of course all of these mappings may not necessarily be present in the associative memory, but those that are can be used directly. Those that are not must be obtained by tracing up through the mapping tables, and may then be entered in the associative memory.

The proposed mechanism must now be considered in more detail. Using semaphores to 'protect' environments which have been by-passed suffers from the disadvantage that the semaphores must be claimed and released for every operation. A more desirable solution would require that the semaphores are only claimed and released as necessary. It is to be assumed that if the protection of the mappings of the shared resources is insisted upon for each operation, then for each consecutive operation in any environment, unless the operation is provided in the next less abstract environment, then at least one environment will require protecting. Furthermore, in a typical case, where the operations are to be interpreted by the micro-code of the RVM, all intervening environments will require protecting at the start of the operation and unprotecting on completion.

Clearly it may be possible, for each instance that creates the existence of a particular environment, to perform the protection operation once only for the first operation performed in that environment, removing that protection again once only for the last operation performed.

6.4.4 Table of Protected Environments

The suitability of the associative memory mechanism discussed in the previous section clearly depends on the mappings recorded in the memory remaining valid over useful periods of time, and upon their removal from the memory when they are changed. Of course information in the associative memory may disappear in time, in which case its reliability ceases to be of any concern. Alternatively a process may wish to modify the resources of its environment, in this case all items dependant upon this environment must immediately be ejected from the associative memory.

Table of Protected Environments

Environment	Readers	Flag set if to be changed	List of Associated Protected Environments
K	1	FALSE	
F	2	FALSE	K, M
C	2	FALSE	F
M	1	FALSE	
B	2	FALSE	C, E
H	1	FALSE	
D	1	FALSE	H
J	1	FALSE	
E	1	FALSE	J

Asynchronous operations

K calls B
M calls A
H calls B
J calls A

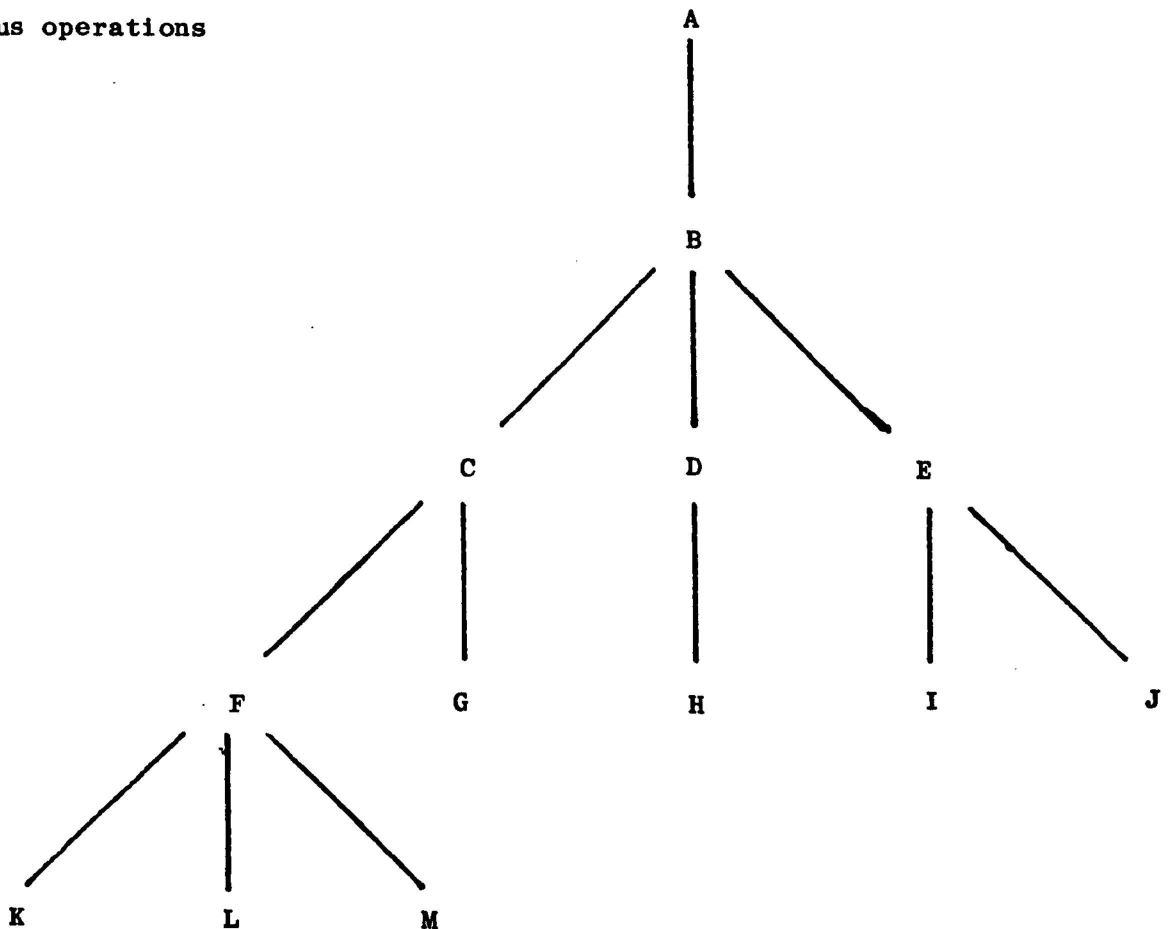


Figure 6.6

Consequently a mechanism is needed which ensures that environments remain protected until there is a definite requirement to remove the protection, and once this requirement has arisen the mechanism must ensure that the environments be unprotected as quickly as possible.

With the above criteria in mind it is proposed to provide a mechanism, based on the following design principles, which will ensure environments are locked as required. The proposed mechanism employs a table which is global to the complete RVM system, and this table contains the names of each environment currently protected. Figure 6.6 illustrates the construction of such a table.

Associated with the name of each environment currently protected is an integer which indicates the number of asynchronous processes currently dependant upon the mappings provided by this environment. When interpreting an operation this table can be examined quickly to determine if the necessary environments require protecting. If the environment name is not present this indicates that a full protection operation has to be performed, otherwise the integer 'READERS' associated with this environment is simply updated. The existence of an environment name in this table thus indicates that this environment has not been changed since its name was placed there, and that all mappings within the associative memory which rely on this environment are valid.

Modification of an environment can now only take place when the number of processes relying upon this environment is zero. When this occurs the environment name must be removed from the table and related items in the associative memory must also be ejected.

In order to eject all the items in the associative memory which are dependant upon an environment to be altered, and also to ensure that the removal of one item from the table of locked environments also forces the removal of all associated, more abstract environments the following information must also be maintained. Each protected environment must have associated with it a flag indicating that it is to be removed when its 'READERS' count reaches zero. If this flag is set no more processes will be permitted to by pass this environment until the pending request to change it has taken place.

Furthermore each entry must contain a list of the environments at the next, more abstract, level which are also protected and dependant upon the current environment for their resources. Thus an attempt to change one environment can cause all associated environments to be flagged as awaiting change.

It must now be noted that the existence of a mapping in the associative memory implies the immediate environments are currently protected. The protection procedure therefore need only take place if a new item is to be placed in the associative memory. Also environments are now only unprotected if, on completion of an operation no processes are dependant upon this environment, and some process has requested to change this environment.

6.4.5 Appraisal of Mechanism

The scheme discussed has considered ensuring that environments remain constant for the duration of each operation at any level of abstraction, these operations being interpreted at some less abstract level. Also

the removal of this protection has been avoided except when specifically requested and therefore the reprotection of environments will often be avoided. As a result the validity of an associative memory mechanism can be assured for longer periods than initially estimated.

Since by protecting whole environments, rather than resources the number of items requiring protecting /unprotecting for any operation is linear with respect to the distance between called and calling environments, the mechanism meets the criteria for efficiency defined in chapter one. It is thus expected that with an associative memory mechanism capable of providing the mappings of the most commonly used operations and objects, and a protection mechanism which adds little overhead to the basic instruction execution cycle, the RVM should be able to work efficiently when performing any form of asynchronous processing.

It has been shown that this proposed mechanism satisfies the criteria for allowing sensible asynchronous processing which were discussed at the beginning of this chapter. It remains only to comment on the envisaged efficiency of the proposed solution. As has been mentioned, the protecting and unprotecting of environments is now only performed when necessary, ie after an environment has been or is to be altered. The frequency of this occurring is envisaged to be small compared with the total number of instructions executed, and since the amount of work involved in this protection and removal of protection has been reduced to a minimal quantity, it is clear that any time spent initiating and removing the protection will be very small in comparison with the time spent performing the actual operations of the machine.

6.5 Refinements to the Proposed Scheme

The original proposal has been to perform any protection of environments for each operation at every level of the RVM hierarchy. However, it has already been noted that such a mechanism is unnecessary when interpreting certain primitive machine operations such as 'load' and 'add' etc. It must therefore now be asserted when the full protection and unprotection of environments is required and when it is possible to make some shórt cuts in the scheme proposed. Clearly these short cuts are possibly only when interpreting primitive machine operations (interpreted on the bare machine, by micro-program; at level Ø) but even then the amount of protection required depends upon the operation being interpreted. A complete quantification for each of the RVM operations is rather pointless since any other implementation will probably choose a more complex I/O system, a different set of primitive operations, and possibly several central processors all with access to the shared memory objects of the complete system.

In the case of extremely basic instructions where the objects being acted on can be obtained within a single memory sycle, then it is apparent that no protection is required at all. However when the objects to be accessed require more than one memory cycle then the number of central processors becomes important, expecially if it is possible to change the objects in between operations. I/O operations will always require that some protection of environments is performed, because an intermediate environment can perform several basic operations during a single I/O operation. It is because of these complex considerations, which are involved when attempting to assess the necessity of protecting environments for any operation, that it is proposed to ensure that on each operation the necessary environments are protected.

In the particular case of the RVM implementation described in chapter four, it is clear that the only primitive machine operations which will require environments to be protected are the I/O operations, and modification of the instruction execution cycle to reflect this fact would further improve the RVM's efficiency.

CHAPTER 7 - CONCLUSIONS

7.1 Introduction

In the previous chapters the design and implementation of a synchronous Recursive Virtual Machine Architecture has been discussed; also a mechanism has been proposed which will permit any kind of asynchronous processing within this machine. It now remains to put this work in perspective and discuss the applicability of such an architecture in relation to other structured computer systems. Also in this chapter the further areas of study, already touched on in this thesis but not fully explored, will be summarized with some indication of the sort of results which might be expected in each case.

7.2 The RVMs Relationship With Other Structured Computer Systems

In chapters two and five various problem areas encountered in other structured computer systems were described. Certain of these systems, notably MULTICS, CALL_TSS and the GEC 4000, do not provide the full range of facilities offered by the RVM. In particular it is not always possible to permit the renaming or creation of objects when mapping them between environments.

The capability based systems, it was suggested, can permit processes to interact asynchronously due to the fact that once a process has been given a capability for an object this capability cannot be revoked by its owner. Thus object mappings are always preserved for the duration of all asynchronous operations. However, in these capability systems, the action of crossing between environments is often extremely complex and inefficient. As a result of this inefficiency other mechanisms are introduced which enable systems to be built and avoid the necessity of crossing between environments. This approach has the undesirable result that further complex protection problems are evolved for which solutions have to be found. The HYDRA system is an example of this approach and in this case solutions to most of these further protection problems have been produced.

The strictly hierarchical structure adopted in the RVM avoids certain of the protection problems encountered in HYDRA, and permits a more logical structuring of an operating system. Also it has been shown that it is possible to cross between environments efficiently in the RVM. However, since it is always possible for a process in one environment to attempt to revoke the access of a process to an object in a more abstract environment, there arises a problem in providing asynchronous processing.

7.2.1 Revocable Capabilities and Asynchronous Processing

There is a dual problem in capability based systems to that of providing asynchronous processing in the RVM. This is the inability of a process to revoke the access of a subordinate process to any object defined in its environment. This is a problem which has been studied both in the HYDRA system and by Redell and Fabry. The solution adopted in both cases permits an environment to consist of revocable capabilities as well as ordinary capabilities. To a process within the environment both revocable and ordinary capabilities appear identical, however the underlying structure is very different. The ordinary capability fully defines the object to which access is being granted; the revocable capability points to another object which in turn defines the object to which access is being granted.

Typically each environment 'owns' the capability list of which it is constituted, however if a capability is revocable then it only owns a pointer to the actual capability. In this way a process is able to use revocable capabilities to define objects in a subordinate environment, and without changing the capability lists can change the capabilities by altering the pointers. This is essentially a similar approach to that adopted in the RVM whereby an environment is defined by its segment table which points to capabilities owned by its ancestor environment.

Because revocable capabilities can be passed as ordinary capabilities there is no reason why one environment should not be defined by a long chain of revocable capabilities through several intermediate environments. For this reason it is considered that the problem encountered in the RVM, that of preserving object mappings when two processes are co-operating using shared

resources, will also occur when revocable capabilities are used. It is clear to see that if two processes sharing data are separated by several intermediate environments, then a process within one of these intermediate environments could easily revoke the calling process's access to the shared data.

The question remains as to whether the mechanism proposed for protecting environments in the RVM in order to overcome this problem can be applied to revocable capabilities in capability based systems. The major difference between the RVM and capability based systems is the lack of a strict hierarchical structure in most capability based systems. The non-hierarchical approach will lead to more complex traces through the environment structure being maintained in order to provide mappings of objects and protection of environments. As a result the associative memory and table of protected environments must become more complex in structure. Also the algorithms required to unprotect environments, when a process wishes to change an environment under its control, will become much more complicated. However it seems feasible that since there is an essential similarity between the RVM segment table structure and the revocable capability, any problems encountered with asynchronous processing using revocable capabilities could be solved by adopting a similar mechanism to that proposed in this thesis.

7.3 Topics For Further Study

7.3.1 Implementation of an Asynchronous

Clearly the most important point concerns the need to implement a fully asynchronous RVM using the mechanisms described in this thesis. It has been shown that the synchronous RVM is capable of providing an efficient means for structuring a computer system but it remains to show that the asynchronous RVM is capable of supporting a general purpose operating system. The design of the mechanism to support asynchronous processing has been such that it should be possible to use the mechanism for each operation performed at any level in the RVM, without adding a significant overhead to that required when crossing between environments in the synchronous RVM. Furthermore the experiment undertaken to investigate this overhead in the synchronous RVM illustrates that this is of the same order as that required to perform a basic machine operation; and thus it is suggested that the order of this overhead in the asynchronous RVM will remain the same. For these reasons it is suggested that the implementation of an asynchronous RVM should illustrate that a hierarchically structured computer system can execute efficiently. The use of such an architecture will permit an operating system to be well structured and provides a high level of protection between environments in the system. Thus it is proposed that the use of a recursive virtual machine architecture will enable a reliable computer system to be produced more easily.

7.3.2 Optimum Associative Memory Size

A further area for study concerns the optimum size of associative memory within the RVM. The current synchronous implementation provides a check on the number of times an item was found in a particular position in the memory. Items are stored in the associative memory in relation to when they were last accessed. Thus if four objects have been accessed since a particular item was last referenced then this item will occupy position four in the memory. This simple mechanism permits a least recently used algorithm to be implemented in order to eject items from the memory, and also enables a check to be kept on the position in the memory that most items are found when they are referenced again.

The figures obtained from the test programs described in chapter four indicated that there was considerably more utilization of items one to seven than of the remainder of the store. The significance of these figures from basically only two distinct programs cannot be estimated. Many more tests would have to be undertaken with much more general use being made of environment crossing in the RVM. It is felt that in general the RVM may perform more memory accesses than a conventional computer architecture in order to execute a basic operation. If this is the case there could be a considerable improvement in the performance of the RVM if the size of the associative memory is increased from 16 to 32 items say. Of course all the experiments so far have concerned the synchronous RVM and the use of the more complex associative memory in an asynchronous RVM could introduce yet further variables in deciding the optimum memory size. The fact that the items most utilized in the current synchronous RVM have been contained in the first half of the store is itself interesting, for this may show that the results of Schroeder [Sc 71] in relation to MULTICS

have a wider significance, and will certainly go a long way to illustrating how any computer system can be optimized considerably.

7.3.3 Alternative Environment Protection Algorithms

A further topic for study concerns the mechanism by which an environment mapping is permitted to be changed. It was suggested, in the previous chapter, that attempts to change an environment mapping should be suspended until no processes were dependant upon the mapping. Also that once a process requests to change a mapping, processes which later attempt to rely upon this mapping should be suspended until the mapping has been changed. This approach permits a general solution to the problem and avoids processes being prevented indefinitely from changing the environment.

An alternative, slightly less expensive, solution may be to always permit processes to rely upon a mapping unless it is actually being changed. A process would thus be able to change a mapping only when no processes relied upon it; and there would in this case never be a queue of processes waiting to rely upon the changed mapping. However there would now be a requirement for a further primitive RVM operation which enabled a process to force an environment mapping to be changed. By experimenting with different systems in the RVM it could then be discovered which is the optimum approach in terms of efficiency. For if it were only occasionally that a process found it necessary to force a mapping to be changed, then this alternative approach may produce a further increase in performance due to the extra simplification of the protection mechanism.

7.3.4 The Problem of Only Protecting Shared Objects

Finally there remains the problem of protecting only the resources used during an operation, as opposed to whole environments. This is an extremely complex area, for as already mentioned in chapter six, a major reason for the efficiency of the proposed solution lies in the fact that an environment can be protected on the initiation of an operation, whereas the resources used can only be protected as they are encountered and thus identified. Thus there is always the possibility of performing several locking operations when attempts are made just to protect resources. In contrast the protection of environments requires a maximum of one locking operation.

It may be considered that protecting resources rather than environments will produce a solution more easily able to provide a general purpose operating system. However if the MULTICS system is considered then it must be noted that this permits only synchronous processing in each main branch of the environment structure. This is a considerably more restrictive approach to that proposed in this thesis, and yet MULTICS is able to provide an extremely sophisticated and reliable time-sharing system. It is therefore proposed that the RVM design mechanisms described in this thesis will prove perfectly adequate for providing a generalized asynchronous computer system. Furthermore, if the RVM is utilized sensibly then it should permit extremely complex operating systems to be built in such a way that they are reliable, well protected and efficient.

7.4 Conclusions

In chapter three of this thesis six requirements were proposed which were deemed desirable in future computer systems. These are summarized as follows:-

i) There should be no supervisor state,

ii) The system should be extremely reliable,

iii) The system should provide a sound protection mechanism,

iv) Objects within the system should be renameable,

v) The system should permit generalized asynchronous processing,

vi) The system should be efficient,

This thesis has advocated the use of a Recursive Virtual Machine architecture in order to provide these factors. It has been shown that the initial problems of providing asynchronous processing can be overcome and a mechanism has been described which fulfils this objective. A synchronous RVM implementation has been undertaken which illustrates that such a system is capable of efficiently executing in environments at all levels of abstraction.

These pieces of work illustrate that a Recursive Virtual Machine architecture is a practical alternative to the conventional two-level architectures currently employed and as such should be considered seriously when designing future computer systems.

REFERENCES

At 72a Atwood J.W. (Editor) 1972, "Project Sue Status Report", Computer Systems Research Group Technical Report CSRG-11, University of Toronto, April 1972.

At 72b Atwood J.W. et. al. 1972, "Project Sue as a Learning Experience", Computer Systems Research Group Technical Report CSRG-19, University of Toronto, September 1972.

BG 73 Buzen J.P. and Gagliardi U.O. 1973, "The Evolution of Virtual Machine Architecture", Proc. AFIPS National Computer Conf. Vol 42 1973, pp 291-299.

BH 75 Belpaire G. and Hsu N. 1975, "Hardware Architecture for Recursive Virtual Machines", Proc. ACM annual conf. Minneapolis, Minnesota, October 1975, pp 14-18.

Br 70 Brinch Hansen P. 1970, "The Nucleus of a Multiprogramming System", CACM Vol 13 No 4 (April 1970), pp 238-241.

Br 73 Brinch Hansen P. 1973, "Operating System Principles", Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

Bu 72 Burroughs Corporation 1972, "B1700 Reference Manual-Preliminary Edition".

Bu 75 Burroughs Corporation 1975, "B1700 Systems Master Control Program (MCP) Reference Manual".

BW 74 Belady L.A. and Weissman C. 1974, "Experiments with Secure Resource Sharing for Virtual Machines", Proc. International Workshop in Protection in Operating Systems, IRIA Rocquencourt, August 1974, pp 27-33.

CHP 71 Courtois P.J., Heymans F. and Parnas D.L. 1971, "Concurrent Control with 'Readers' and 'Writers'". CACM Vol 14 No. 10 (October 1971), pp 667-668.

CJ 75 Cohen E. and Jefferson D. 1975, "Protection in the HYDRA Operating System", Proc 5th Symposium on Operating Systems Principles, University of Texas at Austin, pp 141-160.

CV 65 Corbató F.J. and Vyssotsky V.A. 1965, "Introduction and Overview of the MULTICS System", Proc. AFIPS 1965 Fall Joint Computer Conf. Vol 27, pp 185-196.

DDH 72 Dahl O.J., Dijkstra E.W. and Hoare C.A.R. 1972, "Structured Programming", Academic Press 1972.

De 70 Denning P.J. 1970, "Virtual Memory" ACM Computing Surveys Vol. 2 No. 3 (September 1970), pp 153-187.

Di 68a Dijkstra E.W. 1968, "Co-operating Sequential Processes", in Programming Languages (Ed. F. Genuys), Academic Press 1968, pp 43-112.

Di 68b Dijkstra E.W. 1968, "The Structure of the THE Multiprogramming System", CACM Vol. 11 No. 5 (May 1968), pp 341-346.

DSA 73 DeWitt D.J., Schlansker M.S. and Atkins D.E. 1973 "A Microprogramming Language for the B1726", Preprints of the sixth Annual Workshop on Microprogramming, ACM SIG MICRO, University of Maryland 1973, pp 21-29.

DV 66 Dennis J.B. and Van Horn E.C. 1966, "Programming Semantics for Multiprogrammed Computations", CACM Vol. 9 No. 3 (March 1966), pp 143-155.

En 74 England D.M. 1974, "Capability Concept Mechanisms and Structure in System 250", Proc. International Workshop on Protection in Operating Systems, IRIA Rocquencourt, August 1974, pp 63-82.

Fa 74 Fabry R.S. 1974, "Capability Based Addressing", CACM, Vol. 17 No. 7 (July 1974), pp 403-412.

GEC 74 GEC Computers Ltd. 1974, "User Hardware Handbook - Computer (CPU Nucleus)" GECCL Dec 1974.

Go 73 Goldberg R.P. 1973, "Architecture of Virtual Machines", Proc AFIPS 1973 National Computer Conf. Vol. 42, pp 309-318.

GS 76 Goldberg R.P. and Schwenk H.S. Jr. 1976, "Virtual Machines for the Honeywell 6000 Family", Proc. Compcon 76, Washington D.C. September 1976.

Ho 74 Hoare C.A.R. 1974, "Monitors: An Operating System Structuring Concept", CACM Vol. 17 No 10 (October 1974), pp 549-557.

HR 73 Horning J.J. and Randell B. 1973, "Process Structuring", ACM Computing Surveys, Vol. 5 No. 1 (March 1973).

Jo 73 Jones A.K. 1973, "Protection in Programmed Systems", Phd. Thesis, Carnegie-Mellon University 1973.

JW 74 Jones A.K. and Wulf W.A. 1974, "Towards the Design of Secure Systems", Proc. International Workshop on Protection in Operating Systems, IRIA Rocquencourt, August 1974, pp 121-135.

Lam 68 Lampson B.W. 1968, "A Scheduling Philosophy for Multiprocessing Systems", CACM Vol. 11 No. 5 (May 1968), pp 347-360.

Lam 69a Lampson B.W. 1969, "Dynamic Protection Structures", Proc. AFIPS 1969 Fall Joint Computer Conf. Vol 35, pp 27-38.

Lam 69b Lampson B.W. 1969, "An Overview of the CAL Timesharing System", Computer Centre, University of California Berkeley, 1969.

Lam 71 Lampson B.W. 1971, "Protection", Proc 5th Annual Princetown Conf. on Information Sciences and Systems, 1971, pp 437-443.

Lau 74 Lauer H.C. 1974, "Protection and Hierarchical Addressing Structures", Proc. International Workshop on Protection in Operating Systems, IRIA Rocquencourt, August 1974, pp 137-148.

Le 75 Levin R. et.al. 1975, "Policy/Mechanism Separation in HYDRA", ACM Operating Systems Review, Vol. 9 No. 5, Proc. 5th Symposium in Operating Systems, November 1975, University of Texas at Austin, pp 132-140.

LS 72 Lauer H.C. and Snow C.R. 1972, "Is Supervisor State Necessary?" University of Newcastle upon Tyne Computing Laboratory, Technical Report 30.

LS 76 Lampson B.W. and Sturgis H.E. 1976, "Reflections on an Operating System Design", CACM Vol. 19 No. 5 (May 1976), pp 251-265.

LW 73 Lauer H.C. and Wyeth D. 1973, "A Recursive Virtual Machine Architecture", University of Newcastle upon Tyne Computing Laboratory, Technical Report 54.

MS 70 Meyer R.A. and Seawright L.H. 1970, "A Virtual Machine Time-Sharing System", IBM Systems Journal, Vol. 9 No. 3 (1970) pp 199-218.

Nee 72 Needham R.M. 1972, "Protection Systems and Protection Implementations", Proc. AFIPS 1972 Fall Joint Computer Conf. Vol. 41 pt. 1, pp 571-578.

Neu 74 Neumann P.G. et.al. 1974, "On the Design of a Provably Secure Operating System", Proc. International Workshop on Protection in Operating Systems, IRIA Rocquencourt, August 1974, pp 161-175.

NW 74 Needham R.M. and Walker R.D.H. 1974, "Protection and Process Management in the 'CAP' Computer", Proc. International Workshop on Protection in Operating Systems, IRIA Rocquencourt, August 1974, pp 155-160.

OC 71 Organik E.I. and Cleary J.G. 1971, "A Data Structure Model of the B6700 Computer System", SIGPLAN notices Vol. 6 No. 2 (February 1971), pp 83-145.

Or 72 Organik E.I. 1972, "The MULTICS System, an Examination of its Structure", MIT Press Cambridge Massachussets (1972).

Pa 72 Parmelee R.P. et.al. 1972, "Virtual Storage and Virtual Machine Concepts", IBM Systems Journal Vol. 11 No. 2 (1972), pp 99-130.

PH 76 Perry J.G. Jr. and Hein R.R. 1976, "Virtual Machines for a Tactical Environment", Proc. Compcon 76, Washington D.C., September 1976.

Ra 69 Randell B. 1969, "Towards a Methodology of Computing Systems Design", Software Engineering, NATO Science Committee Conf. October 1968, Gurmisch Germany (Ed. P. Naur and B. Randell), pp 204-208.

Re 74 Redell D.D. 1974, "Naming and Protection in Extendible Operating Systems", Phd. Thesis, Massachussetts Insitute of Technology Project Mac TR 140, 1974.

RF 74 Redell D.D. and Fabry R.S. 1974, "Selective Revocation of Capabilities", Proc. International Workshop on Protection in Operating Systems, IRIA Rocquencourt, August 1974, pp 197-209.

Sc 71 Schroeder M.D. 1971, "Performance of the GE-645 Associative Memory while MULTICS is in Operation", Proc. ACM Workshop on System Performance Evaluation, New York, April 1974, pp 227-245.

SS 72 Schroeder M.D. and Saltzer J.H. 1972, "A Hardware Architecture for Implementing Protection Rings", CACM Vol. 15 No. 3 (March 1972), pp 157-170.

Wa 73 Walker R.D.H. 1973, "The Structure of a Well-protected Computer", Phd. Thesis, University of Cambridge.

Wil 72a Wilner W.T. 1972, "Design of the Burroughs 1700", Proc. AFIPS 1972 Fall Joint Computer Conf. Vol. 41, pp 489-497.

Wil 72b Wilner W.T. 1972, "Burroughs B1700 Memory Utilization", Proc. AFIPS 1972 Fall Joint Computer Conf. Vol. 41, pp 579-586.

Wir 68 Wirth N. 1968, "PL360, A Programming Language for the 360 Computers", JACM Vol. 15 No. 1 (January 1968), pp 37-74.

WLP 75 Wulf W., Levin R. and Pierson C. 1975, "Overview of the HYDRA Operating System", ACM Operating Systems Review, Vol. 9 No. 5. Proc. 5th Symposium in Operating Systems, November 1975, University of Texas at Austin, pp 122-131.

Wu 74 Wulf W. et. al. 1974, "HYDRA: The Kernel of a Multiprocessor Operating System", CACM Vol. 17 No. 6 (June 1974) pp 337-345.

Wy 76 Wyeth D. 1976, "On the Comparison of Protection Systems", Phd. Thesis, University of Newcastle upon Tyne. July 1976.

ZR 68 Zurcher F. and Randell B. 1968, "Iterative Multi-level Modelling - A Methodology for Computer Systems Design", IFIP Edinburgh 1968, pp D138-D142.

Appendix 1

Recursive Virtual Machine Operations

Basic Operation	Other Forms	Operands Used
LOAD	Byte, Literal	Register, Operand
STORE	Byte	Register, Operand
ADD	Byte, Literal	Register, Operand
SUBTRACT	Byte, Literal	Register, Operand
MULTIPLY	Byte, Literal	Register, Operand
DIVIDE	Byte, Literal	Register, Operand
AND	Byte, Literal	Register, Operand
OR	Byte, Literal	Register, Operand
EXCLUSIVE OR	Byte, Literal	Register, Operand
COMPARE	Byte, Literal	Register, Operand
SHIFT (operand +ive = left, -ive = right)	Byte, Literal	Register, Operand
ROTATE (operand +ive = left, -ive = right)	Byte, Literal	Register, Operand
CASE operand OF	Byte, Literal	Register, Operand, Fragment descriptors
DO LOOP	Byte, Literal	Register, Operand, Fragment descriptors
TRANSER TO SON (Literal)		Register, Operand
IF register EQUALS operand THEN	Byte, Literal	Register, Operand, Fragment descriptor

IF register EQUALS operand THEN ... ELSE	Byte, Literal	Register, Operand, Fragment descriptors
NO OPERATION	-	-
CALL	-	Fragment descriptor
CYCLE	-	Fragment descriptor
RETURN TO FATHER	-	-
IF CONDITION CODE EQUALS operand THEN	Byte, Literal	Operand, Fragment descriptor
IF CONDITION CODE NOT EQUALS operand THEN	Byte, Literal	Operand, Fragment descriptor
IF CONDITION CODE EQUALS operand THEN ... ELSE	Byte, Literal	Operand, Fragment descriptors
IF CONDITION CODE NOT EQUALS operand THEN ... ELSE	Byte, Literal	Operand, Fragment descriptors
EXIT FRAGMENT (Literal)	-	Operand
RETURN CONTROL	-	-
START I O	Byte, Literal	Operand

Appendix 2

Sample Recursive Virtual Machine Programs

Although no compiler exists for the following code, and the syntax is undefined, the standard conventions of a block structured language apply. There is only a small amount of I/O shown here as all the data has been initialized prior to the start of each program. Apart from certain "comment" statements, each statement is equivalent to one instruction of the emulated machine. The following notes may help readers to further understand the program.

Underlined words	-eg. begin, these are envisaged as "reserved" words. Some such as begin and end generate no code, merely acting as code fragment delimiters. Others such as true, false and greater etc., take a predefined literal value which is inserted in the current instruction.
Capitalized words	-eg. CALL and CYCLE, these indicate actual instructions used.
Registers	-eg. R(1), these are defined by R followed by an index value.
Comments	- are prefixed by the basic symbol comment and generate no code.
Comparison	- is performed by the COMPARE instruction. This compares the two operands and sets the condition code in the current next instruction pointer. This is then tested by an instruction of the form IF condition code (¬)= operand THEN (... ELSE).
Array subscripting	-eg. a[R(1)], is performed by the use of an index register, this is placed within square brackets and suffixes the array name. Any memory operand may be referenced in this manner.

Indirection	-eg. @B, the "@" indicates that the indirect bit is to be placed in the current address.
Segment specification	-eg. 16<2>, this indicates that the address required is offset 16 in segment 2. Addresses specified by name, eg. sum, will have their segment number implicitly associated with them.
Statement delimiters	- as in 'algol', by a semi-colon or an end.
Labels	- blocks of code may be labelled, eg. l1: begin, the instruction EXIT (l1) will then cause control to be passed to the instruction following the end associated with the named begin.
Macros	-eg. define push (op) = "R(7) := op", these have been introduced in an effort to improve the readability of the programs. Thus certain operations may be redefined, and names of registers can be given more meaningful identifiers.

Program 1

```
begin  comment program to perform a bubble sort;
       define push (op)                  =   "R(7) := op";
       define pop(op)                    =   "op := R(7)";
       define array_size                 =   "R(2)";
       integer no_of_items;               comment number of items to be sorted;
       begin array a[Ø::no_of_items-1];   comment items to be sorted
             array segment_table [Ø::3];  comment segment table for environment
                                                  at next level;
             procedure sort;
             begin comment sorts the contents of 'a' into ascending orders;
                   define count          =   "array_size";
                   define index          =   "R(1)";
                   define interchanges   =   "R(3)";
                   define x              =   "R(4)";
                   procedure swap;
                   begin comment exchange the contents of two adjacent items
                             of 'a', 'index' indicates the offset of the
                             first item;
                         define first  =    "index";
                         define y      =    "R(2)";
                         define z      =    "R(3)";
                         push (y) ; push (z) ;
                         y := a[first] ; z := a+1 [first] ;
                         a [first] := z ; a+1 [first]  := y;
                         pop (z) ; pop (y)
                   end swap;
                   push (index); push (count); push (interchanges); push (x);
                   interchanges := true;
                   CYCLE;
```

```
loop1:              begin COMPARE count, 1;
                          IF condition code = greater THEN x :=true x :=false;
                          x := x AND interchanges;
                          COMPARE x, true;
                          IF condition code = equal THEN
                          begin interchanges := false; count := count -1;
                                index := Ø;
                                CYCLE;
loop2:                          begin x := a[index] ;
                                      COMPARE x, a+1 [index] ;
                                      IF condition code = greater THEN
                                      begin interchanges := true;
                                            CALL swap
                                      end;
                                      index := index +1;
                                      COMPARE index, count;
                                      IF condition code = equal THEN
                                            EXIT (loop2)
                                end loop2_cycle
                          end
                          ELSE  EXIT (loop1)
                    end   loop1_cycle;
                    pop(x); pop(interchanges); pop(count); pop(index)
              end sort;


              array_size := no_of_items; CALL sort;
              R(1) := 3; TRANSFER TO SON R(1), segment_table;
              RETURN TO FATHER
         end
 end program_to_perform_bubble_sort;
```

Program 2

```
begin  comment program to produce all the solutions to the eight queens'
             problem.  The algorithm used is that described by Dijkstra
             [Di 72];
       define n                 =    "R(2)";
       define no_of_solutions   =    "R(3)";
       define k                 =    "R(1)";
       define push (op)         =    "R(7) := op";
       define pop (op)          =    "op := R(7)";
       array x, col [ø :: 7];
       array up, down [-7 :: 7];
       array segment_table [ø :: 3];
       comment x contains the current solution,
             col contains the columns attacked by queens already placed,
             up contains the UP diagonals attacked by queens already placed,
             down contains the DOWN diagonals attacked by queens already placed,
             segment_table contains the segment table for the environment at
                   the next level;


       procedure generate;
       begin comment this procedure generates all the solutions to a (8-n)
                   queens' problem. The total number of solutions found so
                   far is held in no_of_solutions.  The procedure manipulates
                   items of x, col, up and down as required;
             define h    =    "R(1)";
             define x    =    "R(4)";
             define y    =    "R(6)";
```

```
          push(h); push(x); push(y);
          h := Ø;
          CYCLE;
loop:     begin
               define z    =    "R(5)";
               push(z);
               z := col [h]; x := n; x := x-h; z := z AND up [x];
               y := 7 [n]; y := y+h; z := z AND down [y];
               comment z = col [h] AND up [n-h] AND down [n+h+7];
               COMPARE z, true; pop(z);
               IF condition code = equal THEN
               begin comment set queen on board;
                    x [n]:= h; col [h] := false; up[x]:= false;
                    down [y] := false; n := n+1;
                    COMPARE n, 8;
                    IF condition code = equal THEN
                         comment a solution has been found;
                         no_of_solutions := no_of_solutions +1
                    ELSE CALL generate;
                    n := n-1;
                    comment remove queen from square [n,h];
                    down [y] := true; up[x]:= true; col[h]:= true
               end;
               h := h+1; COMPARE h, 7;
               IF condition code = greater THEN EXIT(loop)
          end loop;
```

```
        pop(y); pop(x); pop(h)
    end generate;

    k := -7; n := true;
    CYCLE;
1:  begin comment initialize arrays;
        COMPARE k,Ø;
        IF condition code ¬= less THEN col [k] := n;
        up [k] := n; down [k] := n; k := k+1;
        COMPARE k,7;
        IF condition code = greater THEN EXIT(1)
    end 1_cycle;

    n := Ø; no_of_solutions := Ø; CALL generate
end;
```

Program 3

```
begin comment  this is the 'supervisory' program which executes in the lowest
               level environment of the RVM.  It performs the 'write'
               operation for programs executing at all more abstract levels;
      define level = "R(1)";
      define temp  = "R(2)";
      define base  = "R(3)";
      define interrupt_ptr = "9";
      define address_out_of_segment = "hex(202)";
      define printer = "5";
      define ex_only_access = "2";
      define rd_only_access = "1";
      define full_access = "7";
      define seg_Ø_base = "224";
      define current_instruction_ptr = "8";
      integer correction factor init hex(40002);
                                   comment value to be subtracted from next
                                   instruction pointer in calling environment on
                                   completion of 'write' operation;
      array segment_table [Ø::6] init
            seg_table_entry (full_access, 3,1,seg_Ø_base),
            seg_table_entry (full_access, 3,1,288),
            seg_table_entry (full_access, 3,1,352),
            seg_table_entry (full_access, 3,36,416),
            seg_table_entry (ex_only_access, 3,1,160),
            seg_table_entry (full_access, 3,2,64),
            seg_table_entry (full_access, 5,1,Ø);
```

```
procedure  error_check;
begin
      comment  This procedure checks the reason for a return to this environment.
               If it is to have the 'write' operation performed then this takes
               place and control is passed back to the calling environment;
      base := interrupt_ptr; level := <3> 191;
      comment  Calculate reason for return to this environment;

      CYCLE;
11:   begin
            CMPR level, 1;
            IF condition_code ¬= greater  THEN EXIT (11);
            level := level -1;
            base := base + seg_Ø_base
      end 11_cycle;

      temp := <3> 256 [base] ;
      CMPR temp, address_out_of_segment;
      IF condition_code ¬= equals THEN EXIT (error_check);

      comment  Now perform write operation;
      START_I_O printer;

      comment  Now set up control stack index for current environment;
      base := current_instruction_ptr; level := <3> 191;
```

```
        CYCLE;
12:     begin
                CMPR level, 1;
                IF condition_code ¬= greater THEN EXIT (12);
                level := level -1;
                base := base + seg_Ø_base
        end 12_cycle;

        comment  Now adjust the control stack in the calling environment so
                 that on return execution continues normally;
        temp := <3> 256 [base]; tecp := temp-correction_factor;
        <3> seg_Ø_base [base] := temp;

        RETURN_CONTROL; CALL error_check;
        comment  Return is to this point on any occasion other than the first;
end     error_check;

        temp := Ø;  <3> 191 := temp;
                                comment set value of 'level' for program at next
                                level of abstraction;
        temp := 6; TRANSFER_TO_SON  temp, segment_table;
        CALL error_check;
        RETURN_TO_FATHER
end     program_at_level_Ø;
```

```
begin comment this program executes in all environments at more
          abstract levels than those of the bare machine.  The
          procedures 'print-stars' and 'clear-line' are located
          in segment four of each environment and constitutes
          shared code;
    define level = "R(1)";
    define temp  = "R(2)";
    define write = "<4>128";
    define full_access = "7";
    define ex_only_access = "2";
    define rd_only_access = "1";
    array segment_table [Ø::6] init
        seg_table_entry (full_access, 3,1,32),
        seg_table_entry (full_access, 3,1,96),
        seg_table_entry (full_access, 3,1,160),
        seg_table_entry (full_access, 3,32,224),
        seg_table_entry (ex_only_access, 4,2,Ø),
        seg_table_entry (full_access, 5,2,Ø),
        seg_table_entry (full_access, 6,1,Ø);
    comment the length field, '32', of the segment  table entry for
          segment three is decremented by 4 for each environment
          at a more abstract level.  It should be noted that in order
          to ensure that attempts to address 'Write' trap to level
          Ø, segment 4 is given a size of 128 words as opposed to
          the 64 words given to the level 1 environment;
    external Clear_line, Print_stars;
    temp := <5> 127; temp := temp +1; <5> 127 := temp;
```

```
    comment update level for next, more abstract environment;
    CALL Clear_line; CALL Print_stars; CALL Write;
    temp := 6; TRANSFER_TO_SON temp, segment_table;
    RETURN_TO_FATHER
end program_at_more_abstract_levels;
```

```
begin comment shared procedures 'clear_line' and 'print_stars'
          which reside in segment 4 of each environment at more
          abstract levels than that of the bare machine;
   define level = "R(1)";
   define temp  = "R(2)";
   define i     = "R(3)";
   define push (op) = "R(7) := op";
   define pop (op) = "op := R(7)";
   define print_line = "<6> 2";
   procedure print_stars;
   begin comment this procedure places a number of "*"s in the
                 output line as defined by the current level of
                 abstraction at which the program is executing.
                 If level = n then (2n - 1) "*"s are output;
      define number_of_stars = "R(4)";
      define number_of_spaces = "R(5)";
      define temp 2 = "R(5)";
                    comment only used when number_of_spaces is
                            no longer required;
      define line_ptr = "R(6)";
      integer stars init "****";
      array spaces_stars [1::3] init "Δ***", "ΔΔ**", "ΔΔΔ*";
      array stars_spaces [1::3] init "*ΔΔΔ", "**ΔΔ", "***Δ";
      push (temp); push (i); push (number_of_stars);
      push (number_of_spaces); push (line_ptr);
      number_of_spaces := 16;
      number_of_spaces := number_of_spaces MINUS -1[level];
```

```
comment calculate number of leading spaces as 16 - (level -1);
number_of_stars := -1[level]; SHL number_of_stars, 1;
number_of_stars := number_of_stars +1;
comment calculate number of "*"s to be printed as (level -1)*2+1;
temp := number_of_spaces; SHR temp, 2;
line_ptr := 7[temp];
comment set up line_ptr to index first word which is to
        contain an "*";
temp2  := number_of_spaces AND hex (3);
comment number of spaces modulo (4);
CMPR temp2, Ø;
IF condition code = greater THEN
begin number_of_stars := number_of_stars PLUS -4[temp 2];
      comment number of "*"s remaining to be printed =
              original number + (temp 2 - 4);
      temp2 := spaces_stars +1[temp 2];
      print_line [line_ptr] := temp; line_ptr := line_ptr+1
end;
temp2  := number_of_stars;
SHR number_of_stars, 2; i := 1; temp := stars;
CYCLE;
1:    begin comment print blocks of "****" as necessary;
      CMPR i, number_of_stars;
      IF condition code = greater THEN EXIT (1);
      print line [line_ptr]  := temp; line_ptr := line_ptr + 1;
      i := i + 1
end 1_cycle_block;
```

```
      temp2 := temp2 AND hex(3);
      CMRP temp2, Ø;
      IF condition code = greater THEN
      begin comment print trailing batch of "*"s;
            temp := stars_spaces [temp 2];
            print_line [line_ptr]  := temp
      end;
      pop (line_ptr); pop (number_of_spaces);
      pop (number_of_stars); pop (i); pop (temp)
end print_stars;
procedure clear_line;
begin comment this procedure inserts a piece of text at the
              start of each line to indicate at which level
              of abstraction the RVM is executing, and then
              clears the rest of the line to blanks.  Also it
              initializes the 'op code' and 'status' fields of
              the 'PRINTER' segment in preparation for the write
              operation;
      define write_op = "3";
      array text [Ø :: 4] init
              "NOWΔ", "ATΔL", "EVEL", "ΔΔΔØ", "ΔΔΔΔ";
      push (level); push (temp); push (i);
      temp := Ø; <5> Ø := temp;
                              comment clear status;
      temp := write_op; ROTATE temp, -3, <5> 1 := temp;
                              comment set up I/O operation code
                                      = hex (6000);
      i := Ø;
      CYCLE;
```

```
11:       begin
                CMPR i, 2;
                IF condition_code = greater THEN EXIT (11);
                temp := text [i]; print_line [i] := temp, i := i + 1
          end 11 cycle;
          CMPR level, 1Ø; level := level + text [i];
          IF condition code ¬= less THEN level := level + hex (B0F6)
                             comment convert to decimal characters,
                                     ie. hex (00F0) + [F100 - A if necessary];
          print_line [i] := level, i := i + 1; temp := text [i];
          CYCLE;
12        begin
                CMPR i, 32;
                IF condition_code = greater THEN EXIT (12);
                print_line [i] := temp, i := i + 1
          end 12_cycle;
          pop (i); pop (temp); pop (level)
     end clear_line;
```

OUTPUT OF PROGRAM 3

```
NOW AT LEVEL    1               *
NOW AT LEVEL    2              ***
NOW AT LEVEL    3             *****
NOW AT LEVEL    4            *******
NOW AT LEVEL    5           *********
NOW AT LEVEL    6          ***********
NOW AT LEVEL    7         *************
NOW AT LEVEL    8        ***************
NOW AT LEVEL    9       *****************
```

Program 4

```
begin

        comment  this program runs in the least abstract environment and

                 provides a 'swap' operation to all more abstract environments;


        define   index    =    "R(1)";

        define   temp     =    "R(2)";

        define   base     =    "R(4)";


        define   interrupt_ptr               =     "9";

        define   address_out_of_segment      =     "hex (202)";

        define   seg_Ø_base                  =     "224";

        define   ex_only_access              =     "2";

        define   full_access                 =     "7";

        define   current_instruction_ptr     =     "8";

        define   num_segs                    =     "5";


        array    segment_table [Ø::num segs] base <3> 31 init


                 segment_table_entry (full_access, 3,1, seg_Ø_base),

                 segment_table_entry (full_access, 3,1, 288),

                 segment_table_entry (full_access, 3,1, 352),

                 segment_table_entry (full_access, 3,2, 64),

                 segment_table_entry (ex_only_access, 3,1, 160),

                 segment_table_entry (full_access, 3,36, 416);


        interger correction_factor init hex (40002);
```

```
procedure error_check;

begin
      comment This procedure checks the reason for a return to this
              environment.  If it is to have the 'swap' operation
              performed then this takes place and control is passed
              back to the calling environment;

      base := interrupt_ptr; index := <3> 191;
      comment Calculate reason for return to this environment;

      CYCLE;
l1:   begin
              CMPR index,1;
              IF condition_code ¬= greater THEN EXIT (l1);
              index := index -1;
              base := base + seg_Ø_base
      end     l1_cycle;

      temp := <3> 256 [base];
      CMPR temp, address_out_of_segment;
      IF condition_code ¬= equals THEN EXIT (error_check);

      comment Now perform swap;
      index := <3> 64; temp := <3> 64 [index]; temp2 := <3> 65 [index];
      <3> 64 [index]:= temp2; <3> 65 [index]:= temp;
```

```
        comment Now set up control stack index for current environment;
        base := current_instruction_ptr; index := <3> 191;

        CYCLE;
12:     begin
                CMPR index,1;
                IF condition_code ¬= greater THEN EXIT(12);
                index := index -1;
                base := base + seg_Ø_base;
        end     12_cycle;

        comment Now adjust the control stack in the calling environment so
                that on return execution continues normally;
        temp := <3> 256 [base];
        temp := temp - correction_factor;
        <3> seg_Ø_base [base] := temp;

        RETURN_CONTROL;  CALL error_check;
        comment Return is to this point on any occasion other than the first;
end error_check;

        temp := Ø; <3> 191 := temp;

        temp := num_segs;
        TRANSFER_TO_SON  temp, segment_table;

        comment Pass control to program in environment at next level of
                abstraction;
```

```
        CALL error_check;

        RETURN_TO_FATHER;

end program_at_level_Ø;
```

```
begin
    comment this program runs in all more abstract environments than that of
            the bare machine. The sort procedure is shared code placed in
            segment 4 of each environment;
    define n                    =     "R(2)";
    define i                    =     "R(3)";
    define temp                 =     "R(4)";
    define interchanges         =     "R(5)";
    define push (op)            =     "R(7)  :=op";
    define pop (op)             =     "op := R(7)";
    define ex_only_access       =     "2";
    define full_access          =     "7";

    define num_segs             =     "5";

    define data_area            =     " <3> Ø ";
    define swap                 =     " <4> 64 ";

    array  segment_table [Ø::num segs] init
            segment_table_entry (full_access, 5,1,32) ,
            segment_table_entry (full_access, 5,1,96) ,
            segment_table_entry (full_access, 5,1,160) ,
            segment_table_entry (full_access, 3,2,Ø) ,
            segment_table_entry (ex_only_access, 4,2,Ø) ,
            segment_table_entry (full_access, 5, 32, 224);

    comment note that the length of segment five is reduced by four units
            for each extra level of abstraction at which the program is
            placed;
```

```
procedure sort;
begin
       comment procedure to perform a bubble sort of the data held in
               'data area'.  The number of items to be sorted is held in 'n';

       push(n); push(i); push(temp); push(interchanges);

       n := n-2; interchanges := true;
       CYCLE;

11:    begin
               CMPR n,Ø;
               IF condition code = greater THEN temp := true ELSE temp:=false;
               temp := temp AND interchanges;
               CMPR temp,true;
               IF condition code ¬= equal THEN EXIT (11);
               interchanges := false; n:=n-1; i := Ø;
               CYCLE;
12:            begin
                    CMPR i,n;
                    IF condition code = greater THEN EXIT (12);
                    temp := data_area +1 [i];
                    CMPR temp, data_area +2 [i];
                    IF condition code = less THEN
                    begin
                         interchanges := true; temp := i+1; data_area := temp;
                         CALL swap;
                    end;      i := i+1
               end 12_cycle
```

```
    end l1_cycle

    pop(interchanges); pop(temp); pop(i); pop(n);

end sort;

            n := Ø;
            CYCLE;
l1:         begin comment set up data area;
                CMPR n,126;
                IF condition code = greater THEN EXIT(l1);
                data_area [n] :=n; n:=n+1;
            end l1_cycle;

            temp := data_area [n]; temp := temp+1;
            data_area [n]; := temp;  CALL SORT;
            temp := num_segs; TRANSFER_TO_SON temp,segment_table;

            RETURN_TO_FATHER;

end program_at_other_levels;
```

Appendix 3

BML Code for Address Calculation and 'Special' RVM Instructions

```
ADDRESSING.EXCEPTION:
     LEVEL := LEVEL + X≥1≥; GOTO CRASH;  % NEED TO INCREMENT LEVEL AS
                                          % CRASH DECREMENTS IT ]]]

HARDWARE.ADDRESS...ENTRY:
HARDWARE.ADDRESS:
%                                                                          %
%        HARDWARE ADDRESS CALCULATION                                      %
%                                                                          %
%    FUNCTION:   CALCULATES THE ACTUAL HARDWARE ADDRESS OF THE             %
%        LOCATION DEFINED BY THE CURRENT LEVEL, SEGMENT AND OFFSET.        %
%        ALSO CHECKS THAT ACCESS REQUIRED OF LOCATION IS VALID.            %
%                                                                          %
%    PARAMETERS: INPUT    LEVEL, POSN, SEGMENT.NO, ACCESS.CODE             %
%                OUTPUT   POSN, SEGMENT.NO                                 %
%                                                                          %
     TAS := S1; TAS := S2; TAS := S3; TAS := S4; TAS := S5;
                                          % SAVE SCRATCH PADS
     TAS := LEVEL;                        % SAVE CURRENT LEVEL;
     IF X≥0≥ = SEGMENT.NO THEN SKP; GOTO HWRE.ADDR.2;
                                          % ADDRESS IN SEGMENT ZERO,
        L := FLAGS; IF NO.HARDWARE.REGISTERS THEN GOTO HWRE.ADDR.2;
                                          % REQUIRE ACTUAL SEGMENT ZERO
                                          % CHECK IF IT IS A REGISTER
        X := POSN(10, 14);                % GET OFFSET
        Y := 10;
        IF X<Y THEN GOTO FOUND.ADDRESS.1;
                                          % IT IS A REGISTER, HOO RAY
                                          % IE. REG 0 - 6 , DATA STACK PTR
                                          % , CONTROL STACK PTR OR
                                          % INTERRUPTS
HWRE.ADDR.2:
     LEVEL := LEVEL -SKP X≥1≥; SKIP; GOTO FOUND.ADDRESS.1;
     X := ADDR.MAX.SEG.NO; Y := LEVEL; Y := SHL Y (2); TAS := XPLUSY;
     X := Y; Y := SHL Y (1); X := XPLUSY; X := SHL X (1);
     Y := ADDR.DISPLAY; TAS := XPLUSY; CALL READ.MAX.SEG.NO.AND.DISPLAY;
     TAS := X≥F≥; S5 := TAS;              % KEEP CHECK ON PERMITTED
                                          % ACCESS
     L := FLAGS; IF ≤ A.MEM.PRESENT THEN GOTO LOOP.ON.LEVEL;
     CALL SEARCH.ASSOCIATIVE.MEMORY; X := LA;
     L := FLAGS; IF ≤ ITEM.IN.A.MEM THEN GOTO LOOP.ON.LEVEL;
     LA := X;
     FA := S4; POSN := S1; X := ACCESS.CODE; Y := MEM(-4, FA-);
     Y := XANY; IF X≤=Y THEN GOTO LOOP.ON.LEVEL;
                                          % INVALID ACCESS
     POSN := MEM(-OFFSET.SZ, FA-); X := MEM(-SEG.NO.SZ, FA-);
     SEGMENT.NO := X;
     X := LA; Y := X; Y := SHL Y(1); X := XPLUSY; X := SHL X(7);
                                          % WHERE FOUND * 24 * 16
     Y := BITS.ADDR.STATS; Y := SHL Y(1); X := XPLUSY;
     Y := ADDR.ADDRESSING.STATS; FA := XPLUSY;
     GOTO FOUND.ADDRESS;
LOOP.ON.LEVEL:
     X := S3; Y := SEGMENT.NO;
     IF X>=Y THEN GOTO HWRE.ADDR.3;
        L := SEG.OUT.OF.RANGE; GOTO ADDRESSING.EXCEPTION;
HWRE.ADDR.3:
     X := S2; Y := SHL Y(5); FA := XPLUSY;
     X := MEM(BITS.SEG.ACCESS); Y := ACCESS.CODE; TAS := X; X := XANY;
     IF X=Y THEN GOTO HWRE.ADDR.4;
        L := ACCESS.INVALID; GOTO ADDRESSING.EXCEPTION;
```

```
HWRE.ADDR.4:
     X := TAS; Y := S5; S5 := XANY;
     FA := FA + BITS.SEGMENT.TYPE; X := MEM(BITS.SEGMENT.LEN, FA+);
     X := SHL X (6);                        % * 64
     Y := POSN(10, OFFSET.SZ); IF X >= Y THEN GOTO HWRE.ADDR.5;
        L := ADDR.OUT.OF.SEG; GOTO ADDRESSING.EXCEPTION;
HWRE.ADDR.5:
     X := MEM(BITS.CONTAINING.SEG, FA+); SEGMENT.NO := X;
     X := MEM(BITS.SEG.BASE.ADDR); Y := XPLUSY;
                                            % NEW SEGMENT AND OFFSET SET UP
     X := POSN (0,8); X := RTR X (8); POSN := XORY;
     LEVEL := LEVEL -SKP X≥12; GOTO HWRE.ADDR.6;
     L := FLAGS; IF ≤A.MEM.PRESENT THEN GOTO FOUND.ADDRESS.2;
     FA := S4; X := S5; MEM(-4, FA-) := X;
     X := POSN (10, OFFSET.SZ); MEM(-OFFSET.SZ, FA-) := X;
     X := SEGMENT.NO; MEM(-SEG.NO.SZ, FA-) := X; GOTO FOUND.ADDRESS.2;
HWRE.ADDR.6:
     CALL READ.MAX.SEG.NO.AND.DISPLAY;
     GOTO LOOP.ON.LEVEL;
FOUND.ADDRESS.2:
     FA := ADDR.ADDRESSING.STATS; GOTO FOUND.ADDRESS;
FOUND.ADDRESS.1:
     X := ADDR.ADDRESSING.STATS; Y := BITS.ADDR.STATS; FA := XPLUSY;
FOUND.ADDRESS:
     X := MEM(24); Y := 1; X := XPLUSY; MEM(24) := X;
     LEVEL := TAS;                          % RESTORE LEVEL;
     S5 := TAS; S4 := TAS; S3 := TAS; S2 := TAS; S1 := TAS;
                                            % RESTORE SCRATCH PADS
     EXIT;
```

```
TRANSFER.TO.SON:
%                                                                       %
%       SET UP A VIRTUAL MACHINE AND TRANSFER CONTROL TO IT             %
%                                                                       %
%   FUNCTION: SAVE THE REGISTERS OF THE CURRENT MACHINE, UPDATE         %
%      LEVEL, DISPLAYS, MAX.SEG.NO, SET UP REGISTERS OF NEW MACHINE     %
%      AND CONTINUE PROCESSING. CONTENT  OF REGISTER INDICATES          %
%      NO. OF SEGMENTS AND OPERAND POINTS TO SEGMENT TABLE  FOR         %
%      MACHINE.                                                         %
%                                                                       %
%   PARAMETERS: INPUT    LEVEL, DISPLAY, MAX.SEG.NO, CURRENT.INSTR,     %
%                        T                 CURRENT OP CODE              %
%                                                                       %
%               OUTPUT   LEVEL, DISPLAY, MAX.SEG.NO                     %
%                                                                       %
    TAS := S1; TAS := S2; TAS := S3; TAS := S4;
                                         % SAVE SCRATCH PADS
    IF ≤TF(2) THEN GOTO ILLEGAL.OP;
                                         % ONLY IMMEDIATE MODE LEGAL
    X := 1; CALL UPDATE.PROGRAM.STATUS.WORD;
    CALL CALC.ADDR.OPERAND...ENTRY;
    T := LEVEL; X := SHL T (3); Y := SHL T (4); X := XPLUSY;
                                         % X := LEVEL * 24
    Y := ADDR.DISPLAY; S4 := XPLUSY;     % CURRENT OFFSET IN DISPLAY
    T := S2; POSN := T(10, LEN.INS.OFFSET); SEGMENT.NO := S3;
    ACCESS.CODE := READ.ONLY.ACCESS;
    CALL HARDWARE.ADDRESS...ENTRY; CALL ABSOLUTE.ADDRESS...ENTRY;
    X := FA; FA := S4;
    MEM(BITS.DISPLAY) := X;
                                         % SET UP NEW DISPLAY ITEMS FROM
                                         % OPERAND OF INSTRUCTION
    CALL GET.REGISTER.OPERAND; T := LEVEL; X := ADDR.MAX.SEG.NO;
    Y := SHL T (2); FA := XPLUSY;
    X := S3; IF X≤=0 THEN GOTO T.T.S.3;
    X := S4; Y := 15; IF X<=Y THEN GOTO T.T.S.4;
T.T.S.3:
       L := MAX.SEG.NO.TOO.LARGE; GOTO CRASH;
                                         % EACH MACHINE IS ONLY ALLOWED
                                         % 16 SEGMENTS
T.T.S.4:
    MEM(BITS.MAX.SEG.NO) := X;           % SET UP NEW MAX.SEG.NO
    CALL SAVE.REGISTERS.ETC;
    LEVEL := LEVEL +SKP X≥1≥; GOTO T.T.S.6;
       L := XFER.TO.LEVEL.16; GOTO CRASH;
                                         % ATTEMPT TO SET UP A MACHINE
                                         % AT LEVEL 16
T.T.S.6:
    FA := ADDR.RVM.TIMER; FA := FA + TIMER.SZ; ADDR.RVM.TIMER := FA;
                                         % ADDRESS OF TIMER AT THIS LEVEL
    FA := ADDR.ADDRESSING.STATS; FA := FA + 24;
    ADDR.ADDRESSING.STATS := FA;
    CALL SET.UP.REGISTERS.ETC;
    CALL TRACE;
    S4 := TAS; S3 := TAS; S2 := TAS; S1 := TAS;
                                         % RESTORE SCRATCH PADS
    EXIT;
```

```
RETURN.TO.FATHER:
                                                                          %
      RETURN CONTROL TO CURRENT VIRTUAL MACHINE≥S FATHER                  %
                                                                          %
    FUNCTION: SAVE THE REGISTERS OF THE CURRENT MACHINE, UPDATE           %
       LEVEL, DISPLAYS, MAX.SEG.NO, RESTORE FATHER≥S REGISTERS AND        %
       CONTINUE PROCESSING.                                               %
                                                                          %
    PARAMETERS: INPUT    LEVEL, DISPLAY MAX.SEG.NO, CURRENT.INSTR,        %
                         T              CURRENT OP CODE                   %
                                                                          %
                OUTPUT   LEVEL                                            %
                                                                          %
    SET.HALT.OK.FLAG;
    X := 1; CALL UPDATE.PROGRAM.STATUS.WORD;
RETURN.TO.FATHER...ENTRY:                   % ENTRY POINT WHEN ERROR HAS
                                            % OCCURRED.
    CALL SAVE.REGISTERS.ETC;
    LEVEL := LEVEL -SKP X≥1≥; GOTO R.T.F.4;
       FA := ADDR.INTERRUPTS;               % ALREADY AT LEVEL ZERO, SO HALT
       T := MEM(8, FA+); L := FLAGS;
       IF HALT.OK.FLAG THEN SKP; GOTO R.T.F.2;
          L := HALT.CODE; SKIP;
R.T.F.2:
          L := MEM(24);
       HALT;
       CLR.HALT.OK.FLAG; LEVEL := 0;
       CALL PRINT.STATISTICS; CALL PRINT.ADDR.STATS;
       CALL DUMPS;
       GOTO CLOSE.OUTPUT.FILE;              % CLEAR UP AND QUIT
R.T.F.4:
    T := LEVEL; X := SHL T (3); Y := SHL T (4); X := XPLUSY;
                                            % LEVEL * 24
    Y := BR; X := XPLUSY; Y := TIMER.STATISTICS;
    ADDR.RVM.TIMER := XPLUSY;
    Y := ADDRESSING.STATS; ADDR.ADDRESSING.STATS := XPLUSY;
    CALL SET.UP.REGISTERS.ETC;
    CALL TRACE;
    EXIT;
END
```