ON THE COMPARISON OF PROTECTION SYSTEMS

David Wyeth

Ph.D. Thesis

University of Newcastle upon Tyne

July 1976

ACKNOWLEDGEMENTS

I should like to express my gratitude to Professor B. Randell who has been a constant source of encouragement, support and helpful suggestions, and to thank him for his critical reading of the manuscript. I should also like to thank the staff and graduate students of the Computing Laboratory, many of whom have provided assistance with this work and have contributed to innumerable stimulating discussions.

The original idea for this thesis arose in a discussion with Dr. R. M. Needham and Professor Randell, following a lecture given by Dr. Needham in Newcastle in 1972.

Procter and Gamble Limited very kindly expended programming effort and computer time in gathering statistics on the use made of their computer in Newcastle during one week. These statistics formed the basis of the study reported in Chapter 6.

I gratefully acknowledge the financial support I received for two years from the Science Research Council and particularly thank Professor E. S. Page for the financial support provided by the Computing Laboratory for the last two years of my research work.

Finally, I should like to thank the members of my family for their forbearance, assistance and encouragement.

ABSTRACT

A methodology is presented for performing quantitative cost-benefit comparisons of protection systems. Protection systems in both programming languages and machine architectures can be understood and described in terms of the concept of a domain, an abstract entity which defines the access privileges of an executing program to objects in a system. Though the issues of protection and addressing can be treated separately, the realisation of the close relationship between protection and addressing can assist in the implementation of domains using addressing techniques and provides a basis for the comparison of protection systems.

Current formal models of protection are seen to aid qualitative comparisons but do not provide an effective yardstick with which to compare protection systems. Based on the ideas of protection through addressing, a protection model is developed from which cost and benefit measures of protection are derived in order to achieve the quantitative comparison methodology.

Two detailed examples of the application of the methodology are presented. The first concerns the protection implemented in various Algol W run-time systems, and the second compares the protection system of IBM's 370 DOS/VS operating system with a proposed alternative protection system.

Finally, the comparison of protection systems which exploit structure to achieve protection is discussed. The notion of a structured domain is introduced and used in an assessment of the protection afforded by programmer defined types and a supporting architecture.

TABLE OF CONTENTS

		PAGE
Chapter 1	INTRODUCTION	1
1.1	Protection	1
1.2	The Comparison of Protection Systems	4
1.3	Summary of Thesis	7
Chapter 2	PROTECTION AND ITS RELATION TO ADDRESSING	10
2.1	Objects	11
2.2	Processes	13
2.3	Domains	17
2.3.1	Concept of a Domain	17
2.3.2	Domains and Locality	20
2.3.3	Examples of Domains	23
2.4	Protection and Redundancy	25
2.5	Name, Identity and Address	27
2.6	Protection and Addressing	30
2.7	Time of Protection Checking	34
Chapter 3	THE COMPARISON OF PROTECTION SYSTEMS	38
3.1	Formal Models of Protection and Their Use in Comparisons	38
3.1.1	Lampson's Access Matrix Model	38
3.1.2	Jones's Environment Model	41
3.1.3	Criticisms of the Models	43
3.2	Suitability Measure of Jones	46
3.3	The Modularisation Problem	48
Chapter 4	A MODEL OF PROTECTION AND A METHODOLOGY FOR COMPARISON	52
4.1	A Simple Protection Model	52
4.1.1	Program Model	53
4.1.2	Set Definitions	56
4.1.3	Transitions Between Sets	61
4.1.4	Extensions to the Model	63
4.2	Value of Protection	64
4.3	Cost of Protection	66
4.4	Comparison of Protection Systems	70

PAGE

Chapter 5	A QUANTITATIVE STUDY OF PROTECTION IN IMPLEMENTATIONS OF ALGOL W	72
5.1	Implementations of Algol W	73
5.2	Benefit and Cost Measures	79
5.2.1	Domains in Algol W	81
5.2.2	Benefit Measures	84
5.2.3	Cost Measures	89
5.3	Method of Gathering Statistics and Evaluating Measures	96
5.4	Evaluation of the Measures	99
5.5	Extensions to the Study	106
5.6	Evaluation of the Experiment	108
Chapter 6	INTER-PROCESS PROTECTION IN A NAIVE OPERATING SYSTEM: A STUDY OF THE IBM SYSTEM/370 DOS/VS OPERATING SYSTEM	111
6.1	IBM System/370 DOS/VS Operating System	112
6.2	An Alternative Protection System	117
6.3	A Cost-Benefit Analysis of DOS/VS and the Alternative Protection System	119
6.3.1	Identification of Domains	119
6.3.2	Benefit and Cost Measures	120
6.3.3	Evaluation of the Measures	131
6.3.4	Evaluation of Other Benefit Measures	141
6.4	Evaluation of the Comparison	149
Chapter 7	PROTECTION AND STRUCTURE	152
7.1	Structured Domains	152
7.2	A Measure of Structure within a Domain	157
7.3	Comparison of Structured Protection Systems	159
7.4	Evaluation of a Type System	163
7.4.1	Types in a Block Structured Language	165
7.4.2	A Computer Architecture for Types	169
7.4.3	Evaluation	175
7.5	Complementary Notion of Structured Relationships between Domains	184
Chapter 8	CONCLUSIONS	191
Appendix: Estimation of Supervisor State Time		196
References		199

Chapter 1

INTRODUCTION

1.1 Protection

Protection in a computer system is concerned with controlling the access of executing programs to resources in the system. Included in the term 'resources' are physical resources, such as main memory and input/output devices, and logical resources such as files and programs.

The prime function of a protection system, in a multiprogramming, timesharing or multiprocessor computer system, is the prevention of unwanted interference between executing user programs. This function has two aspects:

1. isolating user programs from each other during their execution and preventing an executing user program from interfering with the supervisor, and
2. facilitating the controlled sharing of resources and information.

The term sharing has a number of connotations. It includes the true serial use of a resource, as in the allocation of the CPU in a multiprogramming environment, sharing of a divisible resource such as main memory, and cooperative sharing as in sharing a file or data structure. Even in the situation where concurrent execution of programs takes place without communication between programs, there is typically sharing of physical resources and operating system routines on the grounds of economy. To some extent, sharing is a matter of synchronisation as well as protection. For instance, if several users wish to modify or examine the contents of a file, not only must they have authority to perform such actions but the actions must be synchronised.

Protection can be regarded as part of the wider issue of security. Security concerns essentially the integrity of a computer system against any form of unauthorised penetration. Security issues include the

correct identification of users, approved use of terminals, prevention of wire tapping, privacy, the handling of tapes and disk packs, as well as the control of executing programs accessing resources within the computer system. Protection deals with security internal to the computer system. Other types of threats perpetrated outside the computer system, such as the monitoring of communication lines, are not addressed directly by protection, though occasionally protection provides means for detecting illegal probes into the computer system.

Protection systems generally assume that users and their processes are correctly identified. This identification in the case of users usually takes the form of a password scheme. It is clear that however secure the protection system is itself, if the identification mechanism can be subverted, a user will have little difficulty in gaining access to any information he wants. It is also usual for protection systems to assume the reliability of the hardware on which the protection system operates, though some hardware failures may be detected by protection checks. The PRIME project (Fabry 1973) has attempted to take account of the unreliability of hardware and the protection system itself by performing two independent protection checks on every access request.

We restrict consideration of protection to the control of access to resources by a process such that a decision on whether or not to allow a requested access to proceed can be made independently of the information (if any) contained in the resource. Data-dependent access checks, such as only permitting a doctor to view medical records of patients who have had a specific disease, can be implemented in the following manner. A procedure is constructed which performs the necessary data-dependent checking when supplied with the access request. The user is given the authority to invoke the procedure but not to access the file of medical records directly. Thus, all his accesses to the file are constrained to go via the procedure. Hoffman (1969) discusses this and similar issues,

which we would classify under the heading of privacy, and proposes a method of solving many of the problems (Hoffman 1971).

Another problem, sometimes considered to be in the realm of protection but which we exclude, is that of guaranteeing that a process is 'confined'. That is, the process does not retain after completion any of the information that has been supplied to or derived by the process during its execution, nor transmit such information to unauthorised processes. The classic example involving a process which should ideally have these properties is the provision of a proprietary program to calculate income tax. A user of the program supplies data concerning his earnings etc., the program returns details of the amount of tax he owes and reports the cost of using the program to the owner of the program. The program must be guaranteed not to retain nor transmit in any manner details concerning the financial status of the user. A protection mechanism can be used to prevent the direct transmission of information, but Lampson (1973) and Fenton (1974) have demonstrated that much more complicated mechanisms are necessary to prevent indirect transmission of information. For instance, by making abrupt changes in the traffic on a channel a process can in certain circumstances transmit data to a second process.

The need for protection systems arises from the observed fact that programs often contain errors and from the necessity of guarding against the malicious user. Design and implementation techniques such as structured programming (Dahl, Dijkstra and Hoare 1972) and the chief programmer team (Baker 1972), and the efforts being directed towards developing effective program proving techniques, can aid significantly in the reduction of errors but do not guarantee their absence.

A significant proportion of errors manifest themselves as protection violations. Protection systems contribute to reliable computing systems by aiding the early detection of errors, and hence their diagnosis and

correction, and by helping to isolate the effects of an error within one part of a system.

A protection system can contribute to the reliability of a process itself if facilities exist for splitting a process into a number of parts and treating each part as a separate unit for protection purposes. The protection system is then able to detect errors sooner, because protection is applied to a unit smaller than the process, and is able to detect certain errors which previously could not be detected because the process was treated as a whole.

Since the specification of a process, and in particular how a process is to be subdivided, involves the use of a programming language, it is reasonable to take a wider view of protection than has been typical in the past. Protection has usually been regarded as one of the functions of the hardware and operating system, but in this thesis we also consider the protection aspects of programming languages.

With the development of high level programming languages, 'scope rules' were introduced mainly for storage allocation optimisation, but they also act as protection rules defining the allowed use of variables within a program. Recent developments with regard to programmer defined 'types' (e.g., forms in Alphard (Wulf 1974a), clusters in CLU (Liskov and Zilles 1974)) provide further means for a programmer to use facilities of protection systems within his programs. Also, by including programming languages in the study, notions such as 'context' and 'scope of names' can be used to aid our understanding of protection systems.

1.2 The Comparison of Protection Systems

A recent report on the future of real time technology (Dept. of Industry 1975) suggests that:

> "The most significant trend in real time technology will be the implementation of systems as sets of interacting subsystems. The

value of this approach has already been recognised in software with the concept of structured programming; it will also become feasible in the hardware as the improved technology makes distributed computing economic..... Such interacting parts are usually called processes, and the concept of a real time system as a collection of interacting parallel processes is fundamental."

(Dept. of Industry 1975)

The emphasis in many real time systems is on reliability, and distributed computing offers the prospect of enhanced system resilience. Protection has a key role in any system composed of interacting parallel processes in detecting and containing errors, so preventing unwanted interference among processes. Thus the provision of new protection systems with improved effectiveness and efficiency will contribute directly to the development of successful real time systems.

There are many protection systems in use today, ranging from systems which present a homogeneous attitude to protection throughout a computer system (e.g., CAL-TSS in which all protection is based on capabilities (Lampson 1969b)) to those which are formed from a collection of ad hoc and often unrelated mechanisms, each applicable to a specific type of resource and providing differing degrees of protection (e.g., storage keys for physical memory, segmentation for virtual memory, access control lists for files).

With the increasing pressure for secure systems, the need for reliable and effective protection systems is certain to grow. Current work on types ((Wulf 1974a), (Morris 1973a), (Wang 1974), (Liskov and Zilles 1974)) is likely to lead to novel protection methods since it simultaneously addresses the questions of protection within a programming language and within a system of interacting processes. The timeliness of relating programming language constructs and operating system/machine architecture constructs is underlined by the report mentioned above:

"Architecture and language should be two sides of the same coin. In the past there has been a divergence between the two, but the increased power of the hardware and greater understanding of language design is bringing them closer together ..... as operating (system)

facilities are codified and incorporated into language and as the hardware provides improved run-time facilities, this divergence will reduce, until programming languages become the primary interface for the system implementer."

"It is to be expected that the availability of low cost hardware will encourage the introduction of new architectural features... Perhaps the most significant effect for real time systems is that the difference between the program as written and as executed is likely to decrease so that the programmer will have greater control over what the computer actually does."

(Dept. of Industry 1975)

Existing machine architectures are still more heavily influenced by hardware considerations (e.g., word/byte access, arithmetic, etc.) than by the structural and logical characteristics of the tasks they are set. It is anticipated that architectures influenced by more macroscopic considerations such as block structure, control structure, composite data items, etc., will have the effect of eliminating some of the potential sources of error in conventional programming, and of assisting the detection, diagnosis, and the determination of the extent of errors.

In this somewhat confused situation, there is lacking a means of comparing protection systems. When designing a new machine architecture or a programming language and its run-time system, it would be useful to be able to compare various alternative protection schemes to obtain some relative indication of their suitability for the intended environment, as well as estimates of their costs in terms of execution time, storage space and implementation.

Abstract models of protection, such as those proposed by Lampson (1971) and Jones (1973), allow many of the salient features of a protection system to be identified and thus compared with other protection systems in a qualitative manner. This thesis attempts to show that there are, however, useful alternative methods of comparison based on, as far as possible, quantitative rather than a very abstract qualitative comparison.

Underlying the method of comparison proposed in this thesis is the recognition of the relationship between protection and addressing. An

understanding of this relationship is both useful for the development of a methodology for the comparison of protection systems and for the development of new protection methods. The relationship is a unifying factor between programming languages, operating systems and machine architectures, encouraging a synthesis of ideas from each of these areas in the design of new protection systems.

The comparison methodology is a cost-benefit analysis method, based on measures for cost and benefit. The general form of these measures is derived but the precise form will depend on the actual comparison being made, that is the protection aspects one is interested in, the given environment, and the criteria for choosing the best protection mechanism. Examples of the use of the comparison methodology given in the thesis illustrate the variety of comparisons which the methodology includes.

There are also aspects of protection systems not amenable to quantification which should also be considered in order to make a comparison in some sense complete. The effect of the protection system on the way programs are written is such a factor since the provision of protection information by programmers could influence the design of a program.

1.3 Summary of Thesis

Until very recently, many people working in the area of protection have argued that protection and addressing are separate issues and so protection systems can be considered independently of any particular addressing mechanism being used. A great deal of thought and analysis of the concepts underlying protection has resulted in the realisation of the close relationship between protection and addressing. A detailed discussion of this relationship and its relevance to the relationship between machine architecture and programming languages forms the first major topic of this thesis.

Secondly, having brought the issues of protection and addressing together, it was then feasible to develop a proposal for the cost-benefit analysis of some protection systems. The third topic is the successful application of the comparison methodology in two practical experiments comparing different protection systems. Finally, ways in which protection systems exploiting structure can be characterised and compared are considered.

The nature of protection and its relationship to addressing are considered in Chapter 2. Characterisations are given of the concepts of process and domain, and the often unrecognised link between the concept of a domain and notions of scope of names, context, etc., in programming languages is pointed out. Examples of protection in operating systems and programming languages are presented to illustrate that the basic issues of protection are similar in both areas.

Chapter 3 contains brief descriptions of current abstract models of protection which, in principle, offer a feasible means of comparison, at least of a qualitative nature. These models are shown to be inadequate for our purposes though a measure suggested by Jones (1973) points the way to an alternative approach; that of comparing protection systems by means of cost-benefit measures.

The proposed methodology for comparing protection systems is explained in Chapter 4 and Chapters 5 and 6 contain two examples of the use of the methodology. The first example concerns protection in programming languages, specifically Algol W (Wirth and Hoare 1966). Statistics were gathered on a set of 'typical' Algol W programs which were then used to compare the protection aspects of different implementations of Algol W. Protection between processes is the central concern of the second example, illustrating the comparison of protection systems where the issue of the isolation of processes is paramount. Statistics were gathered on programs run under the

DOS/VS operating system (IBM 1973) on an IBM 370/135 computer and used to compare the protection system of DOS/VS with an alternative protection system which also uses the storage key mechanism of the 370/135 (IBM 1972).

Chapter 7 introduces the concept of a structured domain which is used to characterise protection systems which exploit structure to achieve effective and efficient protection mechanisms. A measure of structure is derived, based on the notion of a structured domain, and is used in brief comparisons of some protection systems. The protection advantages afforded by programmer defined types and a supporting architecture, which retains at run-time the structure provided by the programmer, are considered in detail.

The final chapter presents the conclusions of this research and attempts to put the work into perspective.

Chapter 2

PROTECTION AND ITS RELATION TO ADDRESSING

In this chapter, the elements with which protection is concerned are considered: the entities to be protected (objects) and the entities to be protected against (processes). Programming language notions such as the name, identity and address of variables, and context are seen to be of direct relevance to protection. In particular, these notions assist the understanding of the transformation function, which, given an object name issued by a process, produces the identity and the location(s) of its representation. The significance of this transformation function is that protection checks are typically carried out during the application of the transformation function or are actually embodied in the transformation function itself. The transformation function, commonly understood as the addressing mechanism, is thus frequently bound up with protection.

The abstract notion of domain (Lampson 1969a) is seen as fundamental to protection and forms a unit of structuring for protection purposes, allowing a process to be associated with a group of access privileges. Inter-process and intra-process protection can be discussed in terms of domains. The emphasis placed on domains reflects their importance to the cost-benefit analysis which relies on the concept for the definition of cost and benefit measures.

To perform any sort of access check, it is necessary to have some alternate specification against which the intended action can be validated. The domain provides such a specification, but the information describing what a program may access, from which the domain can be established, has to be furnished in some manner. Various ways in which such redundant information can be provided are briefly considered.

The application of protection checking can take place at a number of points in time and space. Attention can be restricted to dynamic checks made between the point of issue of an access request and the actual access being performed on the object, or enlarged to include static checks, performed, for example, at compile-time or load-time. Different attitudes to the time and place of run-time protection checks result in a variety of mechanisms to implement protection. In this thesis, we are principally concerned with the dynamic verification of access requests.

2.1 Objects

We refer to the entities within a computer system which are to be protected against unpermitted access as objects. 'Object' defies precise definition, but the intuitive notion of object is sufficient in any particular case to decide whether or not an entity constitutes an object. An object may be a physical resource, such as a page frame, a terminal, a tape drive or disk track, or a logical resource such as a file, a segment, a user defined data structure or a procedure. An object may also be made up of other objects.

An access to an object is an algorithm, defined by hardware, microcode or a piece of program, to reference the object in order to change the value of the object or to extract information from the object. For a particular object there may be a number of different ways in which the object can be accessed. For example, the possible accesses defined for an object which is a segment could be READ a word, WRITE a word, and INCREASE the size of the segment.

Objects are divided into disjoint classes by type, the distinguishing characteristic being that all objects of a given type have the same set of defined accesses. Thus, for any object which is a segment, the accesses READ a word, WRITE a word and INCREASE the size of the segment would be

defined, though the protection system may limit the access a given process has to a particular segment to READ a word. For any object, its type is known and so the possible accesses which can be made to the object are also known. The set of available types may be fixed at the time of creation of the system or it may be dynamic in that new types can be created or old types deleted during the lifetime of the system as is possible in Hydra (Wulf et al 1974b).

Details of an object, the internal form of its representation, or indeed whether or not it exists as a physical realisation, need not be available outside the defined accesses for that type of object. Indeed, one can conceive of 'virtual' objects, objects represented in the form of a set of procedures rather than data.

For the purposes of protection, the information required concerning an object is the identity of the object, to distinguish it from other objects, its type, since this identifies the accesses which are defined for the object, the accesses permitted to the object by the requesting process, and the requested access. A consequence of the restrictions which were outlined in Chapter 1 is that a protection system is not concerned about the nature of objects, the information content of an object, or the semantics associated with particular accesses. A protection system provides a facility which given the identity of an object and a requested access either permits or prohibits the requested access to the object.

Objects are not necessarily atomic, an object may be formed from a composition of other objects and this process may be recursive. A system will initially provide a certain set of objects and support for a number of types, and possibly a mechanism for defining new types. New objects of certain types can be created, e.g., segments and pages, and existing objects deleted. Dynamic creation and deletion of physical resources will in general not occur, though a system may cater for the

attachment of new devices or the temporary removal of a device from the system.

Though creation of a new object may bring into existence a new object to be handled by the system, at some level this 'creation' may appear as a regrouping and renaming of other objects. Ultimately, all logical resources are forms of information represented as sequences of bits, and except for paper tape, magnetic tape, etc., there is no possibility of creation of storage media.

Similarly, object deletion may not be as simple as the destruction of the object. Parts of an object, that is other objects of which an object is composed, may be intended to continue their existence, and certainly in the case of the deletion of a file there is no intended destruction of the disk tracks on which the file resided.

The choice as to what constitutes objects (or a primitive set of objects in the case of systems permitting programmer defined types) in a particular system is at the discretion of the designer of the protection system, and the determination of objects should be made to meet the requirements of the system as a whole. In the case of memory protection, for instance, one may choose a word, page or segment as the basic storage unit of protection.

2.2 Processes

An object has a passive role; the entities initiating access requests to objects are executing programs, i.e., processes. During execution, a process retrieves information from objects and manipulates objects via the accesses defined for each object.

We take as our basic definition of a process that given by Horning and Randell (1973), where a process is defined as a triple (S,f,s): 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 a sequence of states from the state space obtained by applying the action function f first to an initial state and then to each succeeding state.

Working from a strict low level definition of a process such as this, and using techniques of combination, abstraction and refinement, as discussed by Horning and Randell, to deduce new 'higher' level processes, will not always yield the most appropriate structuring of a system into processes. This is because at the level of user programs and operating systems the notion of a process is somewhat arbitrary. The techniques of combination, abstraction and refinement allow many choices at each stage, thus it is often the case that a system can be structured into processes in a variety of ways.

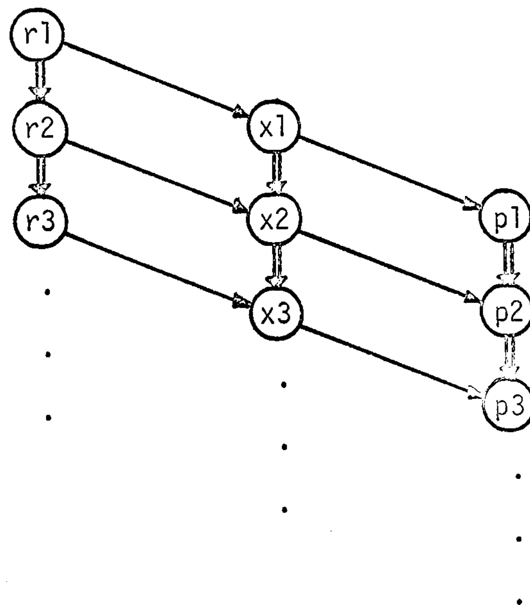


Figure 2.1

The process structure of a system

Consider a spooling system which schedules user jobs one at a time, where each job consists of three phases: reading the program and data, execution of the program and printing of the results. This system can be viewed in

two ways as illustrated in Figure 2.1. (This example is taken from Brinch Hansen (1973)).

The first view, indicated by the single line arrows, is that which would be taken by a user of the system. Each job is considered to correspond to a process; initially it is scheduled to do the input, then the execution phase of the job and finally the printing of the results. Thus a user would recognise the processes:

job 1 consisting of r1 x1 p1

job 2 consisting of r2 x2 p2

. . .

The designer of such a system is likely to take the alternative view indicated by the double line arrows and partition the system into three processes of a cyclical nature each in control of a physical resource, viz:

reader process comprising r1 r2 r3 . . .

execution process comprising x1 x2 x3 . . .

printer process comprising p1 p2 p3 . . .

These three processes progress independently except during short intervals when they must exchange data. The execution process must receive user programs and data from the reader process, or at least an indication of their location, and the printer process must be informed by the execution process where user results are stored.

Both decompositions are useful for a particular purpose, but being abstractions each of them obscures facts about the original system. The first decomposition hides the fact that the jobs share the same reader, processor and printer and from the second decomposition it is not evident that the reader, execution and printer processes execute a stream of jobs. A decomposition of a system into a set of processes is a partial description or an abstraction of that system. How one chooses an abstraction depends on ones purpose.

Though there is this degree of arbitrariness with respect to what constitutes a process, the one aspect which any process structuring must preserve is the degree of permitted parallelism in a system. Thus any act which modifies the degree of permitted parallelism must be regarded as involving an alteration to a process. The essential idea of process structuring has to do with minimising the interactions between parts of a system (Simon 1962), (Myers 1975). In any practical realisation of a process structuring, one wants to choose the basic entities (in this case processes) such that interaction occurs within an entity rather than between entities. If a structuring has been devised in which the majority of interactions occurs between entities rather than within the entities, then perhaps the most appropriate structuring of the system has not been obtained.

In this thesis, we assume that a process is associated with the execution of a user job, though alternative views, such as that illustrated in the previous example, may be more appropriate in other circumstances. Though we associate a process with a user job, we do not want to limit the operating system from having further processes inside of itself, e.g., for spooling. In a particular system, the abstract concept of a process will have some concrete representation, such as a 'process control block'. Thus, we assume that during execution a user job is represented by a particular 'process control block', also that each process created by the operating system corresponds to a particular 'process control block'. If a user job has inherent parallelism, it will cause two or more processes to be created in which case there will be more than one 'process control block' corresponding to the user job.

In a uni-programming system with both supervisor and problem states, the execution of a program can be regarded as a single process executing at certain times in supervisor state and at other times in problem state, or as two processes, one which executes only in supervisor state and the

other which executes only in problem state. As will be seen in Chapter 6, we take the former view, though in this case either view is probably equally valid and the distinction between them will depend on how one wishes to view the system.

2.3 Domains

2.3.1 Concept of a Domain

We define a domain to be the abstract entity which specifies the objects a process can access at a given point in time and the manner in which each object can be accessed. Typically, the permitted accesses to an object will be a subset of those defined for the type of the object. The domain in which a process executes thus defines the access privileges of the process. A process may execute out of a number of domains during its lifetime, though at any particular instant it will be associated with precisely one domain.

The access privileges of a process change when either a process moves from one domain of execution to another, a switch in domains, or the set of access privileges constituting the current domain alter, a domain change. The distinction being drawn here can perhaps be clarified by considering an analogy with a variable in a programming language. A variable has a name and a value, and though its value may change its name remains the same. A switch to a new domain implies a switch from one domain variable to another. Such a switch may well be a temporary one and the previous domain variable can then be reused. A change in the access privileges contained in the domain corresponds to a change in the value of the domain variable. In general, the difference between the contents of two domains is likely to be much greater than the differences within a domain during its lifetime.

The aim of allowing the contents of a domain (i.e., access privileges contained in a domain) to change with time is so that over-frequent

domain switches can be avoided. The contents of a domain can be changed either by the addition of new objects, the deletion of currently accessible objects, or changed types of access including revocation of access privileges by other processes. By a series of such changes the contents of one domain could be altered to coincide with that of another domain, thus avoiding the need for domain switches altogether. However, domain switches are retained as it is sometimes more efficient to make a sudden change in the set of accessible objects by means of a domain switch.

It is expedient to regard domains as objects, since a switch between domains can then be treated as an access to a domain object and hence be controlled by the protection system. In addition, creation of domains and alterations to the contents of a domain can be monitored by the protection system. If certain functions or instructions, e.g., privileged instructions, are to have their use limited, treating them as objects allows them to be included in domains and thus their use can also be controlled by the protection system.

The aim of protection is to restrict a process to those objects it needs to access at any particular time. In the ideal situation, a process would be restricted, at any instant, to accessing a single object, viz. the object it currently wishes to access. This scheme would achieve complete protection, but is impractical and inefficient, it involves a switch in domain prior to each access. In the other extreme, a process is executed in just one domain, in which case the domain must contain all the objects which the process may possibly want to access. As a compromise between these extremes, a number of domains are usually associated with a process. This approach gives much greater flexibility and facilitates tighter protection since the instantaneous name space of the process is restricted to the objects of immediate interest, as parts of a process can be run in more restricted domains than would be the case if the whole process had to be run in a single domain. Against this, however, must be balanced the cost associated with the creation and maintenance of the extra

domains and the cost of switching between domains.

We have emphasised the notion of multiple domains being associated with a process since this facilitates the provision of intra-process protection and includes the simple case where a process is executed in a single domain. Domains are also a suitable abstraction for considering inter-process protection and the notion of controlled sharing of objects. Providing the domains of all processes in a computing system are set up in an appropriate way, interference between processes can be prevented. Sharing of an object is achieved by including access privileges, not necessarily identical, to the same object in the domains of different processes. For instance, one process may be permitted read access to a particular file, whereas another process may be permitted read and write access to the same file.

Though the concept of a domain is well understood, what constitutes a domain in a particular system is somewhat arbitrary, thus one has a choice of what aspects of a system should be linked to the concept of a domain. Some choices are of course more natural than others, but the aim, as with the use of the process concept, is to reflect the structuring of the system. The ways in which programs and operating systems are designed and the manner in which workloads are presented to a system lead to large and abrupt changes in what is going on, specifically the objects being accessed, within the computer system. The relation between processes and domains can be one-to-one, many-to-one or one-to-many. This degree of flexibility can be exploited in order to achieve a structural description of a system which captures the practical realisation in a 'useful' way. That is, it not only describes it well in that large and small changes in what is going on in the system have corresponding changes in the abstract structural representation, but also the associated costs of the changes are minimised. As is discussed later in this thesis, this is particularly

related to obtaining a structural description of a system such that protection can be achieved through addressing.

Domains are a protection structuring device and intuitively it is appealing to tie protection structure directly to some form of program structure. This has the advantage that a change in the program entity being executed then implies a switch in the domain of execution. If protection and program structure are independent, the problem of how to signal a switch in the domain of execution remains to be solved.

Processes are a structuring imposed on activity and domains are a structuring imposed on accessibility, just as procedures, say, are a structuring imposed on a program text. Each such structuring can be done well or badly, and their interrelationship can be set up well or badly. We have enumerated some of the criteria which are likely to be important in structuring activity using processes and accessibility using domains. For the sake of generality, we assume a many-to-many relationship between processes and domains, and assume that each is linked appropriately to some program structuring scheme - e.g., domain linked to procedure, and process to class.

2.3.2 Domains and Locality

The utility of the domain concept depends partly on the notion of 'locality of reference,' that is, the experimentally observed phenomenon that a process tends to access only a small subset of the available objects during a relatively extended period of execution. In many cases, the set of objects which a process actually accesses changes only slowly over a period of time, but studies of reference patterns (e.g., (Morrison 1973), (Hatfield and Gerald 1971), (Hatfield 1972)) have shown that many processes are subject to occasional sudden changes in their behaviour, corresponding to an abrupt and massive change in the set of objects being accessed.

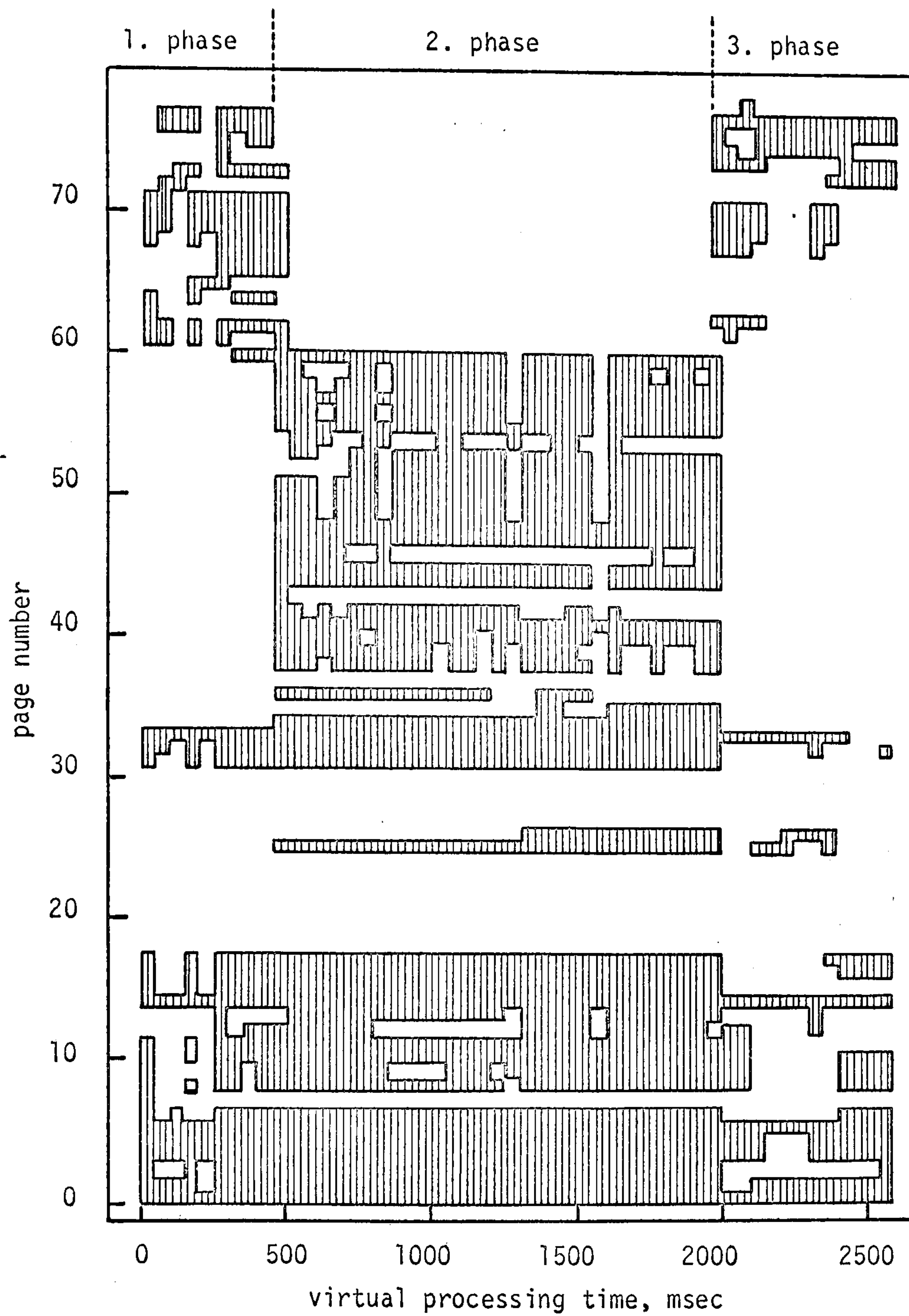


Figure 2.2

Reference pattern of the META7 compiler

Typically, a process will be performing different logical functions in different time periods and thus will tend to concentrate its activity on well-defined subsets of objects. Localisation in space is a desirable characteristic since it can provide limitations on the free access of information and aid recovery in the event of failure (Dept. of Industry 1975).

Example

> Figure 2.2, taken from a paper by Opderbeck and Chu (1974) displays a reference pattern for an execution of the META7 compiler.
>
> "The horizontal axis represents virtual processing time measured in units of 50,000 page references, while the vertical axis represents virtual memory at 512 word resolution. The dark areas show which pages have been used during a given time interval. This reference pattern illustrates the sudden changes of program behaviour as observed during the execution of the META7 compiler. As Figure 2.2 suggests, the execution of META7 consists of three different phases ...." (Opderbeck and Chu 1974).

Evidence such as this suggests that it is quite realistic to associate multiple domains with a process and hence afford the process improved protection. This is not to say that it will be obvious how to allocate access privileges to domains or how to associate the different domains with the different phases of execution of a process.

At this point, it is useful to draw a distinction between the concepts embodied in the words 'possibilities' and 'probabilities'. Possibilities are the objects which may be accessed in the future by a process. Information about possibilities comes, for example, from the explicit provision by programmers of statements which affect domain changes and domain switches. These statements are 'redundant' in the sense that they contribute nothing to the intended progress of the process.

Probability is concerned with the study of accessing patterns, i.e., how programmers actually use objects, to predict what a program will do in the near future based on recent history. Denning (1968), in his work on the working set model has been concerned with 'probabilities'. The notion of locality is directly connected with 'probability'.

The important distinction to be made here is that locality is not an aspect of 'possibility'. 'Probabilities' are assessed over recent history and used to predict some short future interval. Thus, 'probabilities' are not necessarily a subset of the present 'possibilities'.

Since a domain indicates 'possibilities', it says something about which objects can be accessed while the process executes in the domain. A paging mechanism, for instance, might make use of such predictive information. Thus a domain may contain information which could be useful to other parts of an operating system.

2.3.3 Examples of Domains

A capability list (Dennis and Van Horn 1966) associated with a process defines a domain since it denotes all the objects the process may access. Since a process may execute with different capability lists at different times, protection based on capabilities supports multiple domains per process. The use of multiple domains per process can be found in programming languages too. The scope rules of Algol-60, for instance, can be used to define domains consisting of all the variables, arrays, etc., which lie in the scope of the block being executed, together with the parameters which have been passed to any enclosing procedures. If such domains are used, a switch in the current domain of execution will occur on entry to or exit from a block or procedure. In practice, the Algol run-time system designer will choose explicitly what sorts of Algol constructs will involve domain switches, so that an Algol programmer will make implicit choices of domain.

Some systems (e.g. RC-4000 multiprogramming system (Brinch Hansen 1970)) support only one domain per process, often on the grounds that it is too costly to switch domains during execution of a process. In the RC-4000 multiprogramming system, each object in the system is owned by one process and that process alone is permitted access to the object. If a process wants to access an object owned by another process it sends a request, in the form of a message, to the owning process. The owning process decides whether or not to invoke the request, and if it does, it returns the information in a reply message.

CAL-TSS (Lampson 1969b) allows up to 12 domains to be associated with a process. Each domain is directly attached to a 'subprocess' so that, when control passes to a subprocess, the process automatically enters the domain associated with that subprocess. In the Hydra system (Wulf et al 1974b), a 'procedure' includes a specification (template) of the domain in which it is to run on invocation. When the procedure is invoked by a process, the domain in which the procedure is to run (called the Local Name Space) is constructed from the template and the parameters passed to the procedure. In principle, Hydra allows a process to execute in an unlimited sequence of different domains, but the cost of domain switches constrains 'procedures', in the context of Hydra, to be larger than the typical procedure occurring within a program.

Not all uses of the domain concept coincide with our definition. Lampson treats a domain both as a collection of names (i.e., a set of allowed accesses) and as the active entity performing the access. Viewing domains as the active entities initiating access requests within a system conflicts with our intended role of a domain. We want the facility to add access permissions to a domain or delete access permissions already contained within a domain, while still maintaining the identity of the domain. Thus, we subscribe to the view that a domain is a set of allowed accesses.

2.4 Protection and Redundancy

Protection is concerned with checking that the accesses to objects attempted by a process obey certain restrictions on the objects the process may access and the operations which may be invoked. Such checks depend on the provision, by the programmer, of explicit or implicit redundant information. This redundant information forms an alternate specification of what the program may do. In general, this information is not so detailed as the program, but is provided so that errors in the program may be detected when the program is being executed and damage to other processes or information prevented. The term 'redundancy' is used here in the sense of useful redundancy - it adds nothing to the execution of the program, but can be utilised for protection.

Instructions to switch execution to another domain are a very obvious example of information which is redundant, yet can be used to indicate that a process is in error, either in that it has attempted to access an object in an unintended manner, or that previously it had specified an incorrect domain switch.

A less obvious example of redundancy is that provided unwittingly by a programmer when he signs on at a timesharing terminal. The manner of sign-on (e.g., the password used, the identity of the terminal used) may well imply information about what the programmer is allowed to do (e.g., a batch terminal for student jobs), information that can be used by the system in checking his program's actions. The 'JCL' preceding a request to execute a program typically specifies the files and/or devices which are to be used for input and output of data.

The block structure of Algol provides another form of redundancy. The programmer uses the block structure to signify that certain variables can or cannot be accessed within a specific block. The compiler uses this extra information to check that no illegal accesses are attempted.

The union of the information on the restrictions to be imposed on an executing program, provided from sources such as those mentioned above, can be used to define the domain(s) of the executing program. Although potentially useful information may be provided, it will not necessarily all be utilised in the monitoring of the execution of the program. Information provided in the source program may be ignored by the compiler, or used during compilation but not retained in the object form of the source program. Type information is commonly required at compile-time but is often discarded before run-time. At least two machine architectures, the Burroughs B6700 (Burroughs 1972) and the Rice computer (Feustel 1972), however, have been devised to make use of type information at run-time to limit the operations which may be applied to data. Codewords, developed by Iliffe and Jodeit in the Basic Language Machine (Iliffe 1969) and in the Rice computer (Jodeit 1968), were aimed at the retention of structure present in programs written in a high-level language so that it can be utilised for addressing and protection purposes.

If information, which is potentially valuable to a protection system, is discarded before the protection system can view it, the amount of protection which can be provided to the executing program is necessarily reduced. However, the protection system itself may make only limited use of the information with which it is supplied.

It is useful to separate out two aspects of redundancy:

1. What can be provided in the way of an alternate specification of a program's intended behaviour, and
2. The use made by a protection system of the information with which it is provided.

The first aspect is easily understood from manuals describing the programming language and the system being used, though research needs to be carried out on the best form of this specification. This thesis

concentrates on the second aspect; measuring the use made of protection information by protection systems.

2.5 Name, Identity and Address

To invoke an access to an object, the invoking entity (process) must name an object and the access to be made. The object name used by the process must in some manner be translated into the identity of the object, to distinguish the object from other objects. It is useful to distinguish between the name and identity of an object and, where appropriate, its address or physical location, by analogy with programming languages. The definitions of identity, context and universe of discourse given here are due to Dijkstra and Randell.

Even within a single program written in, for example, Algol-60 or PL/1, a particular object may be referred to by different names (called 'identifiers' in Algol-60) and the same name, in different contexts, may refer to different, (simultaneously existing!) objects. A similar situation exists in a multiprogrammed environment in which separate programs may share data or items from a common library.

Different names referring to the same object, or the same name referring (in different contexts) to different objects are relations which only make sense with respect to a universe of discourse in which different objects have different identities, regardless of the various names that may be used to refer to them in various contexts. A context defines how names may be mapped into identities, the latter obviously to be understood in an implied (constant or common) universe of discourse. For instance, at a given stage of execution of an Algol program, the name of a quantity can only be understood in the context provided by the block structure and the depth of nesting of procedure calls.

Taking a step backwards, the universe of discourse of a process

may present itself as a sub-universe among many others in a surrounding universe. What was 'universe' then becomes 'context' and what was 'identity' becomes 'name' to be understood in that context. From this viewpoint, the notion of identification acquires an essentially recursive nature.

In the original von Neumann machine, working in its isolated finite universe, addresses of storage locations serve both as names (occurring in the program text) and as identities of the information held in the locations. Multi-level storage systems force upon us the clearest possible distinction between identity of information and the addresses of the actual locations where the information can be found. As information drifts around the storage system the address associated with a given item of information will no longer be constant in time, it may even be non-unique, viz. when copies of an item exist simultaneously on different storage levels.

These notions apply equally well to general objects in a computer system. For instance, Fraser (1971) has illustrated how the concept of context underlies both the problem of file identification within a computer system and that of the design of programming languages. When a new program is submitted to a system, it will name the context in which it is to run, or the context will be implicit in the manner in which the program is submitted.

A mechanism or <u>transformation function</u> must exist for the translation of an object name issued by an executing program (either explicitly or implicitly) first to the identity of the object and then to its address. In certain cases, it may be possible to bypass the identity in the total transition from name to address.

The nature of names used for objects and the transformation function

can have a significant bearing on the protection system. For instance, one approach to protection is that a process should only be able to name those objects to which it is permitted access. The other extreme, which avoids altogether the problem of the name to identity transformation, is to use unique global names for all objects. In this case, there is a single universe of discourse in the system containing all objects, in which all processes execute.

Many writers on the subject of protection (e.g., (Wulf et al 1974b), (Fabry 1971), (Lampson 1971), (Jones 1973)) make the assumption that all objects within a system are uniquely identified, not only in space (i.e., from other objects which exist concurrently) but also in time, over the whole lifetime of the computer system (i.e., amongst all objects existing in the past or in the future). The mechanism used for such global names is a unique integer of, say, 64 bits, which is assigned to an object on its creation and identifies the object throughout its lifetime. Such global names necessarily pervade the whole system and though they finesse the transformation of name to identity, unique names do not provide an easy means of locating objects, indeed global names give rise to a number of problems in this area (Lampson 1969b).

Systems such as the Rice computer (Feustel 1972) and the Burroughs B6700 (Organick 1973) testify that unique global names for all objects are not a basic requirement of efficient and effective protection systems. We do not deny unique names as an approach to protection, but argue that insistence on such an assumption may preclude potentially useful protection systems. In particular, the exploitation of the notion of context in both programming languages and operating systems offers the possibility of cheap effective protection methods without the need for absolute unique identifiers for objects to be explicit throughout a system.

2.6 Protection and Addressing

Perhaps one of the simplest illustrations of the concept of achieving protection through addressing is the use of a single pair of relocation and bounds registers. Segmentation provides an extension of this idea and enables different types of access (e.g., read, write or execute) to be allowed or denied to each segment included in the address space of the executing program. Both these schemes have the advantage that they can be largely implemented in hardware and so the addressing, and therefore the protection, is efficient.

Lampson (1968) took protection by addressing a stage further and considered how access could be controlled by addressing mechanisms not only to areas of main memory but also to so-called 'privileged' instructions. Privileged instructions are those instructions which could lead to a process accessing information outside of its address space, e.g., I/O instructions, instructions which alter the address mapping hardware, etc. The conventional way of handling privileged instructions is to provide two modes of execution 'supervisor' and 'user'. In supervisor mode, all instructions, including privileged instructions, can be executed, whereas in user mode privileged instructions are prohibited. The prohibition is usually enforced by causing a switch into supervisor mode and transferring to a standard system routine whenever attempted execution of a privileged instruction is detected. Lampson's proposal was to include in a special area of memory provided for each process those privileged instructions which the process would be allowed to execute. The appropriate instructions would be placed in this area by a supervisor routine. The addressing mechanism would allow privileged instructions fetched from this area to be executed, but execution of other privileged instructions would be suppressed.

The DEC PDP 11 computers (Digital Equipment Corporation 1972) achieve

protection of I/O devices in a similar manner. I/O devices are represented by addresses in real memory, reference to an address corresponding to a device causes the reference to be made to the actual device. Devices can then be protected by the addressing mechanism by including or not including them in the virtual memory of a process.

A number of people have argued that conceptually the issues of addressability and protection should be separated. Lampson (1971), in his abstract treatment of protection, assumes that every object ever in a computer system has a unique name. The protection system can then be represented by an access matrix, the rows and columns of which correspond to all the different objects and different domains respectively. Each element of the matrix indicates what types of access are permitted to a particular object by a process when in a given domain. The protection system is then independent of any addressing mechanism used.

Jones (1973) takes a similar view and Needham (1972) argues the case for a lock and key system which is independent of the method of addressing objects. Achieving a protection scheme which is independent of the addressing scheme used is possible, e.g., storage keys on IBM System/360 computers (IBM 1968a), though pragmatic considerations eventually force most designers back to some form of addressing technique.

The importance of addressing techniques in respect of protection is that such techniques can provide a great deal of protection in an efficient manner. Assume a situation in which a process may access objects only through an addressing mechanism provided to it, such that the process cannot alter the mechanism. The mechanism interprets every reference to an object as a name (not an address) in a set of names provided to the program. If that set of names includes only objects which the program is allowed to access, then the addressing mechanism provides a first level of protection since the process has no way of referring to objects for

which no access is permitted. Thus, in a given domain, a process would not 'know' the name of any object for which it did not have some type of access. This is to be contrasted to a system of protection in which the process has a large space of names or addresses, many of which it is not allowed to refer to, so that a check of each access is required to determine if the access is permitted.

In practice, unless the number of bits available to form names permits only the proper names to be formed, a check is still required with addressing schemes, though it is usually possible to arrange that the check is a particularly simple one. The capability mechanism (Dennis and Van Horn 1966), for instance, has the advantage that names may be arranged in such a way as to simplify the check needed with each reference. Usually the check need entail no more than comparing an index with a limit.

With the addressing technique, the function of the protection system has been reduced to that of ensuring that only the right types of access are allowed. In principle, even this function of the protection system can be thought of in terms of addressability. For simplicity, consider just the special case of read or write access to a variable X. Instead of, as above, considering that one has a variable X and operations LOAD and STORE, one can think of having two separate (though of course related) functions LOADX and STOREX. This immediately leads to the possibility of having a name space in which one could specify (i.e., name) the function LOADX but not the function STOREX. So the question of type of access to X has been reduced to that of addressability of functions that can be applied to X. Such a scheme can be immediately extended to the control of peripheral devices along the lines of the DEC PDP 11 manner of addressing I/O devices (Digital Equipment Corporation 1972). There would be problems in implementing such a scheme for achieving protection since the product of the number of functions and the total number of objects will in general be very large. However it demonstrates that all protection can

be thought of in terms of addressability.

A characterisation of protection by addressing is that protection is achieved by a simple check at the cost of a level of indirection. With a lock and key protection system, absolute addresses of objects can be used as names, since protection checking is independent of the access path to the object. However, in many systems, e.g., a segmented virtual memory, a level of indirection already exists, so protection through addressing does not have to introduce an extra level of indirection.

Recognising this relationship between protection and addressing does not immediately solve all protection problems. However, it does allow one to consider protection in terms of existing programming language concepts such as name, value and context, and to consider techniques that have been developed for addressing in structured languages, such as Algol, and at systems such as the Burroughs B6700 (Organick 1973) for pointers to the design of sophisticated protection systems.

In designing a protection system, any reasonable structure, useful redundancy or knowledge concerning the behaviour of programs should be exploited to achieve protection by limiting the name space available to a program during execution. The relocation and bounds registers example above recognises a static name space, viz. 0,1, ..., L-1 (where the bounds register is set to L). The Algol addressing mechanism is a good example of how the name space of a program can change dynamically during program execution. The use of a display and address couples in block-level implementations of Algol-like languages (see Section 5.1) limits the set of names recognised by the addressing mechanism to precisely those which are defined as valid by the language.

A way of increasing the effectiveness of protection achieved via addressing is to exploit structure contained in a system. Chapter 7 concentrates on structure, considering the notion of a <u>structured domain</u>,

and examining such domains in various computer systems.

2.7 Time of Protection Checking

The objective of this section is to consider some of the many mechanisms which have been devised for enforcing protection and to see how they can be characterised in terms of time of protection checking. We are concerned only with run-time checks performed during execution of a process. Static checks performed by a compiler are useful since error indications can be obtained without the need to execute the program. However, to place total reliance on a compiler and run the object code without checks requires that:

- the compiler be correct and trusted always to produce correct object code;
- all names be bound at compile time;
- information describing access authorisations is not changed between compilation and execution;
- the object code cannot be altered before execution.

Similar comments can be made concerning load-time checking.

The safest time to check the validity of an access is immediately before it is performed. Then, concern about whether a compiler was used and its correctness, or whether the object program had been corrupted, is allayed.

Dynamic protection checking, performed at run-time, is also a form of check on the compiler, the loader, the run-time system and the hardware. As such, it contributes towards the reliability of the system. Because of the significance of dynamic protection checking, we concentrate our attention on protection systems which perform explicit checks on every reference to an object or embody protection checks in the addressing mechanism.

In general, there is a time delay between an access request and its

execution. This delay may be due to the address mapping function, queuing of the request due to time delays involved with input/output devices or delays introduced by the fact that the central processor is multiprogramming a number of processes. For similar reasons, there may be a time interval between the time of protection checking and the performance of the access. Thus, the possibility can arise that though an access request was valid at the time of the protection check, it may be invalid by the time the access is performed.

Example

> In IBM's VM/370 system (Belady and Weissman 1974), the protection check and the execution of the action validated by that check are separated by some distance in time in the case of channel programs. It is possible, under certain conditions, such as the occurrence of an interrupt, for the user to change the parameters examined by the check before the action occurs and thus subvert the protection system. This flaw depends on the interruptibility of the check-action sequence and the accessibility of the access request to the user after it has been checked.

To make the protection system as secure as possible, it is desirable that the protection check be carried out as close as possible to the performance of the access, reducing the possibility of the checked request being altered or the specifications of the allowance of that access being revoked before the access is executed.

Dynamic protection checking can be performed during the transformation of an object name to an address either at a single point or at multiple points during the transformation. Since the transformation function is applied to every object name issued by a process, associating protection with the transformation ensures that every access is validated. The obvious places for the application of protection checks are:

- the point at which names are generated;
- the location of the object;
- at some intermediate point(s).

The set of valid object names and accesses is delineated by the domain in which the process is executing so, when considering different times of protection checking, one also considers different representations of domains.

The capability list of Dennis and Van Horn (1966) is an example of protection enforced at the place of generation of access requests. The capability list encodes the domain in a table of capabilities. The names of objects used by a process are simple integers which index into the capability list. A simple test that the object name does not exceed the length of the capability list suffices to determine the validity of the object name. Each capability contains the global name of the object it represents and a specification of the accesses permitted to the object by the process. In this implementation, the information describing a domain is concentrated in one place, at the site of execution, so information is directly available on all the objects which the process may access.

The access control list mechanism, as implemented in Multics (Graham 1968), exemplifies the storage of protection information with objects. The access control list attached to each object consists of a set of pairs, each pair consisting of the identity of a user who is allowed access to the object and the permitted types of access. Such an implementation implies that the object has to be located before the protection check can be carried out. Since the information on each domain is distributed throughout the system it is very difficult to determine all the objects to which a given domain allows access. However, the access control list mechanism enables the set of users who have access to a given object to be determined trivially.

Protection checks are applied at each stage of address translation in the Cambridge CAP computer (Walker 1973) and the Recursive Virtual Machine (Lauer and Wyeth 1973). An address issued in the executing environment is translated to an address in each of the containing environments in turn. Protection checks are carried out at each translation from one environment to the containing environment.

Protection checking is not always associated with the transformation function. For instance, lock and key systems (e.g., storage keys on IBM System/360 computers (IBM 1968a)) are not intimately bound with the transformation function.

Chapter 3

THE COMPARISON OF PROTECTION SYSTEMS

To compare different approaches to protection, it is desirable to distinguish between a functional or logical scheme for protection and its mechanisation or implementation. The term protection system is used loosely to cover both the logical protection scheme and its implementation. For instance, the capability scheme, as presented by Dennis and Van Horn (1966), is a logical protection scheme. The Plessey PP250 processor (England 1972) is an actual implementation of capabilities used for protection.

Ideally, one would like to contrast protection systems in terms of their logical schemes, so avoiding in the first instance at least questions of mechanism, machine architecture, etc. Different implementations of a given scheme could then be compared to give a complete comparison of protection systems.

The descriptive language of a formal model of protection, if sufficiently general, would appear to be an attractive means of comparing the functional schemes of protection systems. The two most notable formal models which have been proposed to date are the access matrix model of Lampson (1971) and the environment model of Jones (1973).

3.1 Formal Models of Protection and Their Use in Comparisons

3.1.1 Lampson's Access Matrix Model

Lampson's model of protection (Lampson 1971) comprises three parts:

1. A set of objects X. An object is any entity to which access must be controlled. Each object has associated with it an identification number which is unique among other objects in the system for all time.

2. A set of domains D. Domains are the entities which permit access to objects. Since domains must be protected from each other, in the sense that execution of a process must pass from one domain to another in a controlled manner, domains are also considered to be objects.

3. An access matrix A which governs the accessing of objects by domains.

The access rights which domains have to objects are specified in the form of an access matrix. Domain names identify the rows and object names the columns. The entry A(d,x) contains a list of the access rights held by domain d to object x. A monitor checks for each access request (d,α,x) whether α is in A(d,x), and allows α access of d to x if $\alpha \in A(d,x)$ and not otherwise. When a process executing in domain d requests α access to x, the identification of the domain is provided by the 'system' and thus, in principle, cannot be forged. Hence, a process executing in a given domain can only access objects as permitted by the access matrix.

A set of four rules governs the establishment, deletion and modification of the elements A. These rules facilitate the transfer and deletion of access rights on command of a domain, providing the domain requesting the change has the appropriate authorisation. For this purpose, Lampson introduces the attributes of 'owner' and 'control' and the notion of the 'copy' flag. 'Control' access reflects a partial ordering between domains. Each object has one or more 'owners' who are automatically granted the complete set of access rights to the object. The copy flag is introduced so that a domain can prevent an untrustworthy subordinate domain from giving away access

permissions to objects. The monitor of the access matrix enforces these rules, ensuring that the access matrix is altered only in an authorised manner.

The question of whether the 'owner' of an object should be able to take away access permissions to the object from other domains is still open. Many systems, including Lampson's model, permit it, but Vanderbilt (1969) puts forward a contrary view.

The switch of a process from one domain to another is also controlled by entries in the matrix. A process can only enter those domains for which the current domain, in which the process is executing, has the 'call' attribute.

Above, we have talked of a single monitor validating all access requests. In any real system, it is likely that objects would be divided into classes such as files, segments and terminals. Associated with each class would be a monitor through which all accesses to objects of that class would pass to be validated. In the case of files the monitor would be the file system and for segments the segmentation hardware. It is a key point of the model that access rights are interpreted by object monitors at the time accesses are attempted.

Graham and Denning (1972) have extended Lampson's work in several directions. They allow more than one process to execute out of a domain and so introduce the term <u>subject</u> to denote the entity which requests access to objects. A subject is identified as a (process,domain) pair. This is a more faithful abstraction of many systems which do not implement a number of distinct domains for each process or even a domain per process. For instance, the domain characterised by supervisor state on IBM System/360 computers (IBM 1968b) is shared by all processes, but each process has its own problem state domain characterised by a particular storage key.

Graham and Denning have also investigated the problems of creating and deleting subjects and objects. Creation of a non-subject object, e.g., a segment, is straightforward and consists of adding a new column to the access matrix. The destruction of an object, permitted only to its owner, corresponds to deleting the column representing the object from the access matrix.

In the case of a subject, creation consists of adding a row and a column to the matrix. The destruction of a subject, permitted only to its owner, corresponds to deleting both the row and the column from the access matrix.

A number of minor extensions are also considered by Graham and Denning, such as the requirement that subjects be members of a hierarchy and the idea of indirect access, to facilitate the implementation of cooperation among mutually suspicious subsystems.

3.1.2 Jones's Environment Model

Jones's model (Jones 1973) attempts to formalise notions of protection within a set of processes sharing the use of objects. The objective of protection is seen to be the restriction of a process to the objects of immediate relevance to the task that the process is performing. Fundamental to the model is the concept of a _domain_ (referred to as an _environment_ by Jones). A domain is defined as a set of _rights_, each specifying an object and an access applicable to that object.

The model has three major aspects: the enforcement of protection, the transfer of rights into or out of domains, and the binding of process execution to domains.

The activity of a process is controlled by the _enforcement rule_ which states that a process can access an object only when the _right_

to do so is in the current domain of that process. The enforcement rule is fixed for all protection systems, but variations in the representation of domains and the manner in which protection checking is performed give rise to a variety of protection mechanisms.

To facilitate changes to a domain, a protection system must specify a policy for the movement of access rights into and out of a domain. To accommodate a variety of policies, the model provides a mechanism in the form of a set of primitives (viz.: COPY, DELETE, GRANT, CREATE) in terms of which a policy may be specified. The primitives are the only means by which domains may be manipulated.

The way suggested by Jones to control the use of primitives on individual rights, is to extend each right by a field which specifies how the right itself may be manipulated. An obvious example is the specification of whether or not the primitives COPY,GRANT,DELETE can be applied to a right by the process executing in the domain containing the right. To control the alteration of domains, domains are treated as objects so that they can be protected by the protection system.

The <u>domain binding rule</u> of the model requires specification of how crossings between domains can take place. Hydra (Wulf et al 1974b), for instance, links domain boundary crossing to nested procedure calls. In connection with domain boundary crossing, Jones discusses in detail the passage of rights between domains as parameters and defines a primitive AMPLIFY to cover the case where rights to a parameter are expanded during the act of crossing to the new execution domain. The purpose of amplification is to provide a controlled means to permit a called procedure domain to have greater access to an object named in a parameter right than the caller possesses.

The model allows the dynamic creation of types, that is programmer defined types of objects, based on the already existing set of types.

Objects of a programmer defined type can be created and accesses to such objects controlled by the protection system in the same manner as is done for other objects. Each object is assumed to have a unique global name so that an object can be identified uniquely among all objects in the system.

Jones illustrates how different abstract protection systems can be defined in terms of the model and then compared on an informal basis in terms of this notation.

Jones introduces the concept of <u>closure</u> which, together with the model, forms the basis of a technique for stating and proving properties that restrict execution. Informally, the closure of a domain is the set of domains which can be derived from the given domain by applying the rules governing the movement of rights into and out of domains.

The idea of obtaining proofs concerning the properties of a protection system is a very attractive one. It could lead the way to a formal treatment of protection with the obvious benefits of practical systems being supported by a well developed theory. In her thesis, Jones proves only one or two weak properties of an elementary system and it would be necessary to demonstrate that stronger and practically useful results could be obtained before this approach could be used as an aid to compare protection systems.

3.1.3 Criticisms of the Models

Formal models have a definite place in the comparison of protection systems, but their utility to date has been limited. Both models allow many protection systems to be expressed in a common abstract framework. For comparison purposes, this is a reasonable start, since protection systems which can be adequately described in terms of a model can be compared in the common language of the model. This leads to a qualitative

comparison but does not provide a yardstick for a quantitative comparison.

The description of a protection system in terms of the primitives of Jones's model would appear to be extremely complex, judging by the examples contained in her thesis (Jones 1973). This implies that it would be very difficult to extract important differences and note similarities between descriptions of two protection systems in terms of her model. However, if from the description of a protection system in terms of the model, theoretical results could be derived, along the lines indicated by the property of closure, then the formal model could prove a useful tool for the purposes of comparison.

Lampson's work (Lampson 1969a, 1971) was a first attempt to present the functional capabilities of several of the existing protection systems within a unified context. Different existing systems seem to correspond roughly to different ways of representing the matrix A and different choices of access modes and the rules, which permit changes to be made to the contents of the access matrix.

Not all protection systems can be readily described in terms of the access matrix. The form of addressing and protection typified by the Burroughs B6700 system (Hauck and Dent 1968), where a domain is structured into a display and sets of descriptors, is not well suited to a lucid description by the access matrix. (To be precise, we are referring to a modified form of the B6700, along the lines of the Algol W run-time system described in Chapter 5, in which all necessary protection checks are carried out.)

The drawbacks of Lampson's model include its inability to reflect directly any structure in a protection system, since all domains, objects and access rights are expressed in terms of a single matrix. The model assumes that an access permission applies uniformly to an object and its components. To enforce different domains having different access permissions to the components of an object (e.g., records of a file)

it is necessary to represent each component as an object in the access matrix as well as the object itself. Thus, a file is regarded as an object and some of its records may be objects too. With the access matrix model one has to be content with an unstructured, one-level view of the set of objects constituting a system. It is not possible to reflect the relationship between an object and its components, that is to represent objects formed by the combinations, regroupings and renaming of other objects. The creation and deletion of such objects thus poses difficult questions, largely ignored in the description of the access matrix model.

In Jones's model some of these criticisms are overcome. Objects have a more natural structural representation in terms of other objects and the type of each object is recognised.

An important aspect of both models is that objects must have unique identification numbers. Though necessary for realising the access matrix, many protection systems achieve protection without the need for names of global validity, for instance the Cambridge CAP machine (Walker 1973) and the Recursive Virtual Machine (Lauer and Wyeth 1973).

A key issue in both models is the checking of all accesses to objects, in particular, the access matrix model implies an active check of each access. In contrast, possession of a capability in the system proposed by Dennis and Van Horn (1966) is taken to be prima facie evidence of the right to access an object. No checking is thus needed at the time of access.

The achievement of a comparison method based on a formal model does not seem imminent. Thus, the remainder of the thesis will concentrate on more practical and less formalised means of comparing, and hence classifying, protection systems.

3.2 Suitability Measure of Jones

As a contrast to the formal model approach, which concentrates on a detailed description of protection systems, another line of attack is to gain a macroscopic view of protection systems and use that for comparison purposes.

An example of a quantitative macroscopic measure is the suitability measure of Jones (1973). Jones considers a demand, that is the specification of a situation to be satisfied by a protection system, to be defined in terms of:

a. constraints on the contents of domains, and

b. constraints on processes crossing from one domain to another.

So, a demand consists of a set of processes, a set of domains and their contents, and the possible domain crossings which each process may make.

Given a demand expressed in these terms, the aim is to compare how different protection systems satisfy or approximately satisfy such a demand, without adding processes or domains or otherwise altering the specifications. If a protection system requires domains to be nested one within another, then the implementation of a given demand may require that some domains contain more objects than necessary. Thus a protection system may prevent the construction of a domain which contains precisely the access rights needed to perform a computation.

To measure how accurately an execution domain suits its use, Jones introduces the accuracy measure.

Definition

The accuracy measure of a domain is the ratio of the number of access rights exercised in the domain compared to the total number of rights which could be exercised within the domain during the performance of one task.

If a procedure is invoked in a domain a number of times, the accuracy measure for that domain is defined to be the average of the accuracy measures over a representative sample of procedure invocations.

This measure has a maximum value of 1, attained when the domain is exactly tailored to its use, and a lower bound of 0 approached as the number of unused rights in the domain increases.

Using the accuracy measure to quantify how closely an implementation of a single domain meets the specification of that domain's contents, Jones proposes a measure, the suitability measure, which indicates how well a particular implementation satisfies a given demand.

Definition

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

A system cannot introduce new processes or domains to satisfy a demand, but might force processes or domains to be merged, and thus, or in some other way, cause the number of access rights that are provided in a domain to be 'unnecessarily' increased. For example, a demand can specify between which domains a process can cross. When an implementation cannot meet such a specification, Jones assumes that the two specified domains, between which a process is to cross, will be implemented as one domain, so that the crossing is rendered unnecessary.

A specific demand, which involves say k domains, has an implementation, in terms of a particular protection system, involving k or less domains. An accuracy measure can be computed for the implementation of each of the k domains. Merging domains in the implementation lowers the accuracy measure for those domains so that

an implementation's approximate solution to a domain crossing specification is reflected in the average of the accuracy measures.

The suitability measure ranges in value between 1, which indicates that the demand has been met exactly by the implementation and 0, a lower bound approached in cases where the implementation fails to reflect the demand situation.

Given a specific demand, the suitability measure will yield an ordering of protection systems along the suitability scale. Jones uses the suitability measure in an informal manner to compare the protection facilities of IBM's OS/360 MVT (IBM 1968b), Multics (Graham 1968), CAL-TSS (Lampson 1969b) and Hydra (Wulf et al 1974b), and obtains the following ordering:

$$0 < S(OS/MVT) < S(Multics) < S(CAL\text{-}TSS) < S(Hydra) < 1$$

where S(Z) indicates the value of the suitability measure for the system Z.

As pointed out by Jones, the accuracy measure for an execution domain is dependent on the execution which takes place in the domain. This in turn is dependent on the decomposition of the problem into processes and domains. The accuracy measure will indicate when a domain has more rights than are needed, but does not reflect 'under-stocking', which occurs if an unnecessarily fine decomposition generates multiple tasks each to be executed in almost identical domains. We return to this point in the next section.

3.3 The Modularisation Problem

The example contained in this section demonstrates that, although approximate, the suitability measure is useful for comparing protection systems, but also points to some of its deficiencies.

Consider a process P which uses access rights a,b,c,d,e,f, and can be viewed as two closely interacting routines P1 and P2, where P1 uses rights a,b,c,d and P2 uses rights a,b,e,f.

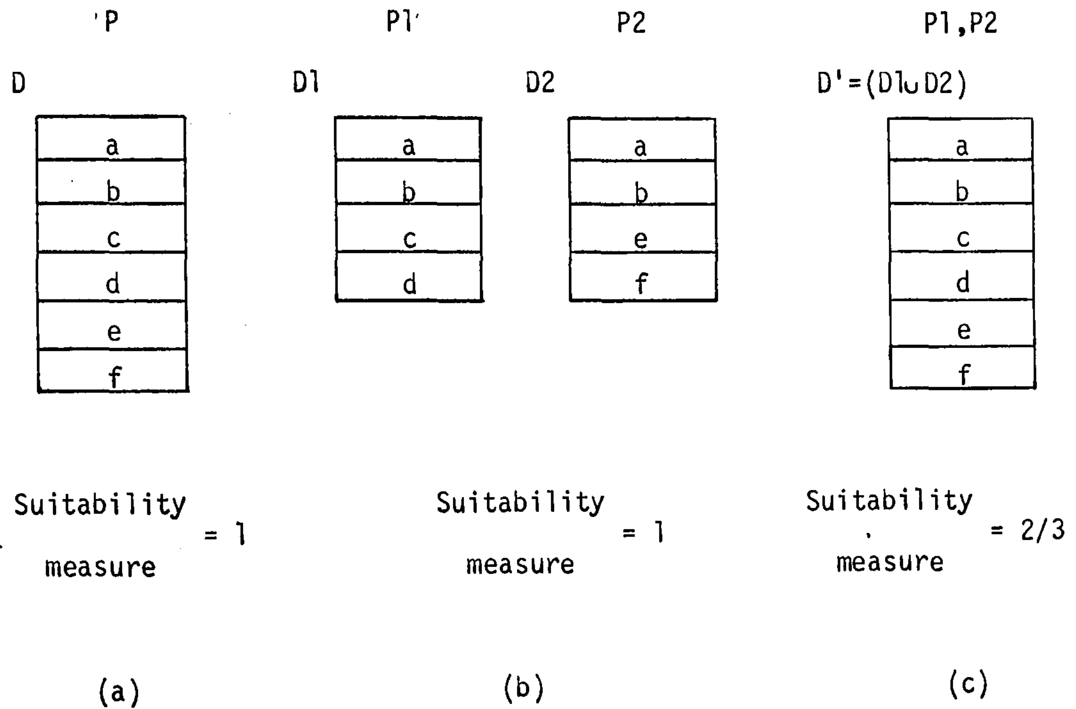


Figure 3.1

Possible execution domains of P and their suitability measures

In Figure 3.1, the three relevant implementations of the domains are shown. In (a), the process is regarded as a single entity P and executes out of the domain D which contains the six required access rights. The suitability measure in this case is 1.

Case (b) shows the process viewed as two separate routines P1 and P2, each executing in its own domain, D1 and D2 respectively. This implementation achieves greater protection than (a) in the sense that

the process is more restricted in what it can access at any time. In (a), the process can access any of six objects at any instant, but in (b) the process is always restricted to four objects. The suitability measure has the same value (namely 1) in both cases but does not reflect the extra protection achieved in (b). The reason for this is that the demand in (a) was specified as a process P and a domain D containing rights a,b,c,d,e,f, no indication being given that P splits into P1 and P2.

The third implementation (c) illustrates a protection system which although cognizant that P splits into P1 and P2 is unable to support separate domains D1 and D2. The suitability measure for (c) = 2/3. This reduced suitability measure, compared to that for (b), reflects the fact that the protection system has had to merge the two domains D1 and D2 into the domain D'.

This example illustrates that the suitability measure provides a means of comparing protection systems with respect to the original demand specification given in terms of processes and domains. The suitability measure is an attempt to quantify how closely a given protection system satisfies a given demand. What is not expected of the suitability measure is that it should indicate when a demand would be better phrased in terms of a different set of domains and domain crossings. The optimum manner in which any programming problem should be divided into processes and 'modules' and hence into domains and domain crossings is a very difficult question and is largely a matter of intuition.

This is the problem highlighted by cases (a) and (b) above, where (a) and (b) are two specifications for the same situation using different sets of domains. A measure which would indicate that (b) gives better protection than (a) is the average (with respect to time or number of

references) of the number of rights available to a process. 'Better' protection is interpreted in the sense that the process has, on average, fewer access rights which it can use, since this reduces the chance of an undetected error.

This measure would indicate that more protection (in the restrictive sense) could be achieved by introducing more domains. This is how the protection was increased in (b) over that in (a). This will often be the case, but there is likely to be some optimum number of domains for a given problem. If a problem is decomposed into the maximum number of domains possible, each domain would contain exactly one access right. One then has the extremely costly situation of switching domains each time the process wants to exercise a different right. A single domain prohibits the decomposition of a process into suitable routines, each executing in its own domain, so there is likely to be some intermediate point which offers a large degree of protection for a reasonable cost in terms of number of domains and domain crossings. It would be very difficult to determine by a measure the optimum number of domains for a given situation, since the decomposition of a problem into modules has many side effects other than protection, such as reliability and understandability.

The suitability measure provides a practical means of comparison but is limited in its scope. No account is taken of maintaining domains and other questions of cost, nor of excessive domain crossings. The suitability measure depends only on the concept of a domain and not on other aspects of Jones's model, such as the primitives by which access rights are created and deleted. However, this measure demonstrates the feasibility of comparing protection systems in terms of measures. The next chapter pursues this approach and develops measures which facilitate a cost-benefit comparison of protection systems.

Chapter 4

A MODEL OF PROTECTION AND A METHODOLOGY FOR COMPARISON

This chapter develops a more pragmatic approach to comparing protection systems than that of formal models, namely that of obtaining a macroscopic rather than a microscopic view. Another major distinction between this new model and those discussed in the last chapter, is that it is very closely based on the arguments given in Chapter 2 for considering protection and addressing together, rather than as separate or separable issues.

A simple model of protection is proposed and the application of the model to the comparison of protection systems is described. The model is somewhat arbitrary and is not important in itself, it is used primarily as a vehicle for obtaining precise definitions for cost and benefit measures. The comparison of protection systems is in terms of a cost-benefit analysis based on the measures.

4.1 A Simple Protection Model

For programs written in most, if not all, languages, it is possible at any point in the execution of a program to identify various sets of variables. Two such sets are the set of accessible variables and the set of variables which exist but are not currently accessible. The protection model formalises the notions of the various sets of interest.

Examples of these sets can be identified, for instance, in the execution of an Algol program. During the execution of a block or procedure in an Algol program, there is a specific subset of all variables declared in the program which are accessible, as defined by the rules of Algol. This subset includes the local variables declared

in the given block or procedure, the global variables declared in any containing block and any parameters passed to the enclosing procedures. If a standard implementation of Algol is considered (such as that described by Randell and Russell (1964)), the set of accessible variables can be identified with a collection of accessible regions of the run-time stack. There will be portions of the stack inaccessible to the block or procedure currently being executed, corresponding to variables declared within blocks or procedures not within the scope of the current block or procedure. As execution proceeds, blocks and procedures will be entered and left, and these sets of variables will change.

4.1.1 Program Model

Two types of program instruction are distinguished, variable accessing instructions and context changing instructions.

A _variable accessing instruction_ specifies an operation, such as add, read, etc., and one or more object names. The object names may be explicit names or they may be implicit in the operation, for example, 'add the two integers currently on top of the run-time stack'. The type of operation and the exact form and representation of objects will depend on the 'level' of machine in which one is interested. The aim of the protection model is to abstract from the physical representation of objects so the model is applicable to any desired 'level' within a computing system.

Before considering context changing instructions, we define the environment and the domain of a process.

Definition

The _environment_ of a process is the set of existing objects potentially accessible by the process.

This somewhat vague definition attempts to convey the intuitive notion of an environment. An environment delimits the totality of objects which a process may attempt to access. In terms of Chapter 2, an environment corresponds to the universe of discourse of a process.

The current domain of execution of a process will only allow the process to access a subset of the process's environment. The environment is the union of all objects to which all domains, which a process may enter, permit access. The fact that an object is in the environment of a process only makes that object potentially accessible by the process, because the process will have to enter a domain which specifically permits access to the object before being able to access that object.

An environment is a dynamic entity. Objects may be created and become accessible to the process or the process may be granted access to an already existing object, in either case the object is added to the environment of the process. If an object is destroyed it will be removed from the environment.

Example

> In the Cambridge Capability Computer (Walker 1973), the environment of a process is specified by the process capability segment which contains the complete set of capabilities which potentially may be referenced by a process.

At a specific point during execution, a process may be restricted to a subset of its environment. This subset is referred to as the current domain of the process.

Definition

The domain of a process at time t (measured in process time) is the set of access permissions which the process is able to exercise at time t.

Sometimes we will refer to the objects in a domain, meaning those objects in the process's environment to which the domain permits access.

The introduction of time here is only a device by which to distinguish different domains, actual time is unnecessary, something as simple as numbers of instructions executed will suffice. A process can only be in one domain at any instant, but at two different times $t_1, t_2, (t_1 \neq t_2)$ a process may be executing in two quite different domains d_1, d_2.

Example

In the Cambridge Capability Computer, a process is restricted, at any instant, to a subset of the process capability segment as defined by the current indirection tables (Walker 1973).

A change in the contents of a domain or a switch to a new domain is indicated by one or more context changing instructions. The function of such instructions is to provide the addressing/protection mechanism with information which determines the currently accessible set of objects.

A transformation function is required which will take any name issued by a process and use it to identify unambiguously an object within a given environment. In performing this transformation, use may be made of the information provided by the context changing instructions.

Example

In a typical Algol run-time system, names of variables take the form of address couples (l,d); l indicates the lexicographical level and d the displacement within that level. If a display is maintained in registers D(0),D(1),...,D(n), the transformation function is:

$$D(l) + d$$

(contents of the l-th display register plus the displacement d). This yields the identity (address) of the variable in the stack which is the environment for the Algol program. Typically, this identity will be converted to a virtual address by adding the address of the base of the stack and then to a physical address via segmentation hardware.

In those protection systems in which protection is embodied in the addressing mechanism, the domain of a process is specified by context information. Context information is a priori knowledge, saying that, for some probably significant time period, it is known that the state of the addressing mechanism will be constant. Rather than transmit such information with each instruction or address, it can be more efficient to extract it and provide it once at the beginning of the time period. Setting up the display on entry to a block in an Algol program, for example, can be viewed as the transmission of information defining the new domain.

4.1.2 Set Definitions

The protection model consists of the identification of five sets of objects pertaining to a process: the totality of objects in a computer system T, the environment of a process E, the theoretically accessible set A, the accessible set as defined by an implementation A', and the referenced set R.

The set of objects contained in the environment of a process (defined in the previous section) at time t is denoted by E.

In general, a process will only be permitted to access a subset of the objects contained in its environment at a given time. Normally, we can distinguish between two subsets of the environment, the theoretically accessible set A and the accessible set A' as defined by a particular implementation, though in some cases A and A' may have the same contents.

Definition

The theoretically accessible set A is a subset of the environment E. A contains those objects within E which the process may access at time t, as defined by some external specification.

The set A corresponds to the current domain of execution defined by information on the job-card (together with a definition of the JCL), for instance, or by certain statements (e.g., statements affecting scope rules) in the programming language used to write the program.

Example

In the execution of an Algol program, where domains are tied to blocks and procedures, the set E corresponds to all variables, arrays, etc., stored on the run-time stack. The set A consists of all variables, arrays, etc. accessible from the block currently being executed.

In practice, the realisation of the current domain in which a process executes by a particular implementation may contain objects other than those to which the external specification indicates access may be required. This may be because the implementation under consideration chooses to ignore certain of the information with which it is supplied regarding access requirements, perhaps because there is no obvious implementation, or because the cost of utilising all the information is too great. Another reason may be that certain checks are carried out at compile-time but are not reinforced by corresponding run-time checks. The accessible set as defined by an implementation is determined by the transformation function F (defined in the previous section).

Definition

The accessible set as defined by a particular implementation A' is the set of objects, at time t, to which the transformation function F will grant access if F is presented with a suitable set of object names.

Example

In the virtual machine design of Price (Parnas and Price 1973), (Price 1973), the environment of a process is its Virtual Space, a set of segment descriptors defining all the data which a process may access. The implemented accessible set of a process is defined by its Working Space since this contains segment descriptors describing all the data which a machine instruction may reference at a given instant.

The set A corresponds to what should be accessible to a process, whereas set A' corresponds to the set of objects actually accessible to a process at any given time. The protection mechanism is presumed to detect references to objects not contained in A'. An undetected protection violation will occur when an object $X \in A'$ is referenced and $X \notin A$.

While a process executes in a domain, with a corresponding theoretically accessible set A, the process will actually reference a set of objects R.

Definition

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

When R is referred to without mention of time, it will mean the set of objects referenced whilst the process is in a given domain. If a domain

D is temporarily left when the set R has the value R_1, then, when execution returns to domain D the set R starts with the value R_1. However, each time a domain is entered to perform a separate logical function, strictly each time a domain is created from the same specification, the set R has the initial value $\emptyset$ (the empty set). Temporary and permanent context changes are distinguished by the context-changing instructions which cause them.

The set R may include items outside A and A' because A and A' may change with time. R can be a useful indicator of the accessing patterns of a process, in particular, one is likely to be interested in R before a switch in domain and its relation to R built up after a switch.

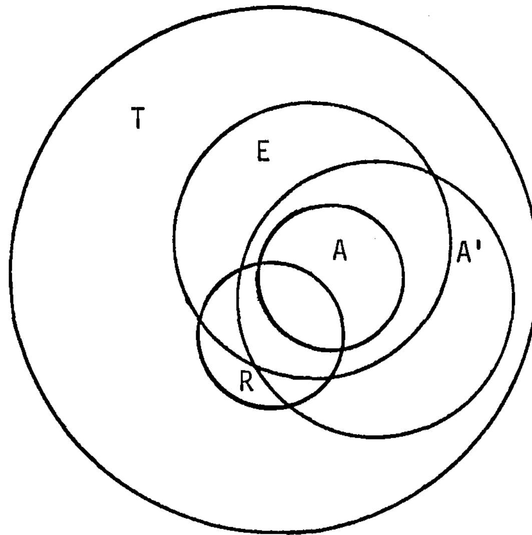
To complete the set definitions and to ease the task of defining set transitions, one further set is defined.

Definition

The set T is the totality of objects in a computer system.

The set T is composed of all existing and 'non-existing' objects. This apparent contradiction reflects our view that object creation is not truly an act of creation, but rather a combination, regrouping and renaming of other objects, similarly for object destruction.

The relationships between the sets are shown by means of a Venn diagram in Figure 4.1



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

Figure 4.1

Relationships between the sets

4.1.3 Transitions Between Sets

On a domain switch, execution switches to a new domain, thus the sets A and A' also switch to new sets corresponding to the new domain of execution. During execution within one domain, the actual contents of the domain may change resulting in corresponding changes to the sets A and A'.

To accommodate changes to these and other sets it is necessary to consider object transitions between sets. The following is a list of possible object transitions between sets, involving the theoretically accessible set A rather than the implemented accessible set A'. The notation $S_1 \rightarrow S_2$ is used to denote a transfer of an object $x \epsilon S_1$ from S_1 to S_2.

1. $E \rightarrow A$	an object is transferred from the environment to the accessible set.
2. $A \rightarrow E$	an object is transferred from the accessible set to the environment and so becomes, temporarily at least, inaccessible.
3. $T \rightarrow A$ ($T \rightarrow E$)	an object (or possibly a group of objects in T regarded as a single object in E) is transferred from T to the accessible set, and by implication to the environment.
4. $A \rightarrow T$ ($E \rightarrow T$)	an object is transferred from the accessible set (and by implication from the environment) to the set T; due to combination and regrouping the object may be represented by a number of objects in T.
5. $T \rightarrow E$	an object is transferred from the set T to the environment of the process, thus in some domain the process may be able to access the object.

6. E→ T an object is removed from the environment and transferred to T, again due to regrouping the object may be represented by a number of objects in T.

Similar further transitions must also be included if the referenced set is being considered.

A new domain of execution may be specified as a set of additions and deletions to the set of access permissions constituting the previous domain or might be specified by an explicit list of access permissions. Similarly, a set of object transitions can be specified for a domain switch to turn the old set A into the new set corresponding to the new domain. Alternatively, a complete specification of the new set A may be given independent of the previous domain.

The transition E→ T includes the cases of object deletion and revocation of access permission. If permission to access an object is revoked, then the object is effectively removed from the environment of the process. A certain object may not be currently accessible, but may be <u>potentially</u> accessible, i.e., the process could enter a domain in which the object would be accessible. If the process enters a domain from which it is no longer possible to enter a domain (by one or more domain crossings) in which an object x is accessible, then x may be deleted from the environment.

The dynamic creation of objects while executing in a domain, e.g., the use of records in Algol W, involves transitions T→ A and T→ E but does not affect the set R. The same transitions are involved in the case of a process gaining access to an object not previously in its environment, while executing in a specific domain, but again the set R is not affected.

4.1.4 Extensions to the Model

The model has been described in terms of restricting access to objects, the extension to different types of access presents no problem. To distinguish between different types of access permitted to an object, consider an object 0 with possible access types $a_1, a_2, \ldots, a_n$, as a set of pairs $(0,a_1),(0,a_2),\ldots,(0,a_n)$, and treat each as a separate object. If the current domain includes object 0, but only with permitted access types a_i, a_j, a_k, then objects $(0,a_i),(0,a_j)$ and $(0,a_k)$ are in the corresponding accessible set A, and objects $(0,a_h)$ $h \neq i,j,k$ are not included in A.

This level of detail may not always be appropriate and the simple model described above may be quite sufficient in many circumstances. For instance, in Algol-60, apart from value parameters, there is no notion of a variable being read-only or write-only, so there is little to be gained in trying to make the distinction. However, if procedures are included in the set of accessible objects and are represented by descriptors and referenced by address couples, as in the Burroughs B6700 (Hauck and Dent 1968), to 'read' a procedure could mean the act of passing a procedure as a parameter. To 'execute' a procedure would have the obvious meaning, but to 'write' a procedure would be meaningless. In such a situation, it may well be desirable to make the distinction between 'read,write,execute,...' types of access to objects.

The extension to different access types or operations could be of use in considering machines with a tagged architecture such as the Burroughs B6700 (Burroughs 1972) or the Rice Computer (Feustel 1972). Any operation σ would further restrict the set of accessible objects to objects of the form $(0_i,\sigma)$, i.e., those objects $0_i \epsilon A'$ for which access type σ is allowed.

The inclusion of domain switching operations presents little problem, particularly if a 'procedure' or 'subroutine' is associated with a domain. The domains to which one could transfer from the current domain would be

the 'procedure' objects in the current domain with the access type call.

Further extensions to the model to include user defined datatypes and operations, and the notion of a structured domain are discussed in Chapter 7.

4.2 Value of Protection

In terms of the above model, the accuracy measure of Jones, for a given domain, is the ratio $|R|:|A|$, (where the notation $|S|$ is used to denote the cardinality of the set S), of the size of the set of referenced objects R to the set of theoretically accessible objects A. (This measure ignores possible problems which may be caused by the fact that A may vary with time.) Averaged over a number of domains this measure becomes Jones's suitability measure.

Accuracy alone can give a very distorted picture of the true situation. A process may only use one object in a domain of two objects, an accuracy of 50%, but the domain could be restricting the process to two objects out of 1000, so the protection would be quite reasonable.

Jones's suitability measure deals with what a process actually accesses and attempts to evaluate how well a protection mechanism matches up to the dynamic behaviour of a process. Basing the evaluation of protection systems on the detailed dynamic behaviour of processes gives rise to a number of problems (c.f., the modularisation problem discussed in Chapter 3) because of the need to know the detailed accessing patterns of processes, which vary greatly from one process to another, and because of the possibility of measuring aspects of processes rather than protection mechanisms.

An alternative and more direct approach is to consider the use made by protection systems of available information, e.g., context information, which can be used to form domain specifications. In other

words, the implementation of a domain (i.e., the accessible set A' defined by the implementation) compared to the specification of the domain (the theoretically accessible set A), based on a priori knowledge. It is patently impossible to provide more protection than supplied information allows, but it is quite feasible to make only partial use of this knowledge.

Definition

The _value_ or _benefit_ of a protection mechanism is defined to be

$$|A| / |A'|$$

averaged over the domains of execution of a sample set of programs.

Thus, the value of a protection mechanism is defined as an empirical measure of how well implemented domains match the true domains; that is, the use made of context information or an external specification of what should be accessible. The measure is independent of the detailed behaviour of actual programs, since it is independent of the referenced set R. Sample programs are only needed for the specifications and implementations of a set of domains over which to average the benefit measure. The average can be a simple mean or a weighted average based on the 'time' spent in each domain or the importance, from the view of potential protection violations, of each domain.

Within a domain, weights could be applied to objects if some were considered much more important than others, e.g., shared objects, critical system objects, etc. If objects enter and leave the domains of a process during its execution, a time average based on the length of time each object remains in a domain can be taken for the benefit measure.

Above, we have used the term 'size of a set' without being explicit about its definition. The precise meaning of this term will depend very much on one's attitude towards objects. An array, for instance, can be regarded as a single conceptual object or as a collection of component objects. The 'sizes' of the sets A and A' are thus left to be defined for each particular comparison, so that the most appropriate measure of size can be used. In the next two chapters, the examples of the comparison of protection systems use storage requirements as a measurement of size and in Chapter 7 comparisons involving structured objects are considered.

4.3 Cost of Protection

A protection mechanism may be implemented in a combination of hardware and software, so the true cost of protection might be a measure of amount of hardware, speed of system (i.e., amount it is slowed down by protection checks), provision of software etc. In addition, a protection system or mechanisation may have an influence on the whole operating system and on the writing of user programs. It would be convenient if protection could be discussed and measured in isolation from the rest of the system. This is normally not possible, indeed in some systems it is impossible completely to separate the protection function from the addressing function since one mechanism has been devised to achieve both aims.

To avoid at least some of these problems, we follow the approach of Wortmann (1972) who devised an abstract but potentially useful cost measure for the comparison of different implementations of SPL (a dialect of PL/1).

A simple but illustrative cost measure is a linear measure based on the number of domain switches made by a process. This measure is

crude, but, being easy to calculate, can lead to rapid gross comparisons of protection systems.

Alternatively, a more complicated cost measure can be devised which emphasises the amount of information which has to be supplied or transmitted to set up domains and the bookkeeping involved in the maintenance of domains. We break down the total cost of applying protection to a process into the following six components:

1. Cost of creation of a domain
2. Cost of deletion of a domain
3. Cost of maintenance of a domain
4. Cost of domain crossings
5. Cost of dynamically adding or removing objects from a domain
6. Cost of protection enforcement, i.e., that a named object lies within the current domain.

The cost of creation of a domain is likely to depend on the size of the domain (i.e., the number of objects or access permissions to objects contained in the domain) and the amount of parameter checking etc. which has to be done. Domain deletion will usually be trivial and may have a fixed cost (possibly zero), since it will often involve little more than the deletion of information defining the domain. However, in some systems, deletion of a domain may be a much more complex task. In the Hydra system (Wulf et al 1974b), on the deletion of a domain it is necessary to decrement the reference count of every object for which there is a capability in the local name space defining the domain.

The cost of maintenance of a domain will usually reduce to a function of the space necessary to store a description of the domain, which in turn will depend on the size of the domain. The description of a domain may be kept in different places when execution is actually taking place in the domain and when the process is temporarily executing in another

domain. In the execution of an Algol program, for example, where the current domain is represented by the display registers, other domains, to which the process may return, are represented in terms of linkage information stored on the run-time stack.

A fixed cost could be assumed for each transfer to a new domain (domain switch), this would be appropriate if a domain switch merely involved the resetting of a fixed number of registers. A more realistic cost would be one related to the 'size' of the new domain, that is the amount of information required to specify the new domain. This would be applicable to situations in which a description of the domain has to be loaded into a specific area of memory or piece of hardware before the desired procedure can be executed in that domain.

The addition of access permissions to a domain or the removal of such permissions will involve a cost dependent upon the representation of the domain and the amount of information required to indicate the change.

The cost of protection enforcement may be independent of the current domain of execution (e.g., checking access bits in a segment descriptor). The total cost of enforcement is then a constant multiplied by the total 'time' (e.g., number of references) spent in execution by the process. Alternatively, the cost of checking the validity of an access may depend on the size of the current domain. One would expect the cost of enforcement to have a linear dependence on time, but access to 'well-protected' objects may cost more than to other types of objects.

Certain mechanisms may appear to involve no cost for protection enforcement, e.g., storage keys on the IBM System/360. Here though there is a hardware cost, albeit a once only cost, and presumably some small time delay to test the result of the protection check on each storage reference.

The general form of the six cost components has been discussed and it is envisaged that these components could be determined for any particular protection mechanism. Each component will generally be made up of a number of parts, such as storage space, time (in practice number of references) and possibly hardware costs. These parts could be treated separately or reduced to a common denominator, namely money.

Each cost component, for example the creation of a domain, will involve the invocation of a set of basic or primitive actions, the number of times each primitive action is invoked will depend on the domain in question. The actual form of the cost measure for a particular protection mechanism is obtained by determining the primitive actions $p_{i1}, p_{i2}, \ldots, p_{ik_i}$ involved in the i-th cost component $(i = 1,2,\ldots,6)$. The costs $c_{i1}, c_{i2}, \ldots, c_{ik_i}$ of each of the primitives are then determined. The total cost of the protection mechanism for a sample set of programs is then:

$$\sum_{i=1}^{6} \sum_{j=1}^{k_i} n_{ij} c_{ij}$$

where n_{ij} is the number of times primitive action p_{ij} is invoked during execution of the set of programs. We are assuming that the costs of the primitives c_{ij} have been reduced to a common denominator or time, space etc., are being considered separately, in which case there will be a sum of this form for each type of cost. Hardware costs, being once only costs, can be determined directly without resorting to such a summation.

This relatively complicated cost measure is in no sense an absolute measure of cost. Instead, this discussion is to be viewed as a set of guidelines for determining a more abstract cost measure for a protection mechanism, one which will facilitate at least a rudimentary comparison of costs of different protection mechanisms.

4.4 Comparison of Protection Systems

The model described above has been developed with a view to performing quantitative or qualitative comparisons of protection mechanisms. The model provides a measure to evaluate the benefit of a protection mechanism and guidelines for deriving a cost measure for the provision of the protection.

The cost measure is based on the run-time costs of performing the protection checks. Attempts have not been made to measure any increase or decrease in programming cost due to restrictions or requirements of the protection system, nor to evaluate any additional benefits which accrue from a particular mechanism.

Specific instructions for performing a comparison of protection mechanisms cannot be given due to the variety of possible criteria for evaluating which of several protection mechanisms is best suited to a particular application. However, we endeavour to convey a general approach by means of two examples contained in the next two chapters.

In the ideal situation, a need would arise to evaluate various protection systems in a well understood environment for which many 'typical' programs are in existence. A simple criterion for 'best' would be given, e.g., maximum amount of protection for a given cost. A straightforward quantitative analysis is then feasible. First the benefit and cost measures to be evaluated must be considered in terms of each of the candidate protection mechanisms. From this analysis, the statistics required to evaluate the measures can be determined. These statistics can then be gathered from the set of 'typical' programs by using a suitable method (e.g., alteration to a compiler, interpreter or emulator, or by simulation) without actually implementing the protection systems being considered. The measures can then be evaluated and the 'best' system determined.

If a new system and machine are being designed, it is unlikely that programs will exist for the new machine architecture. Thus, intelligent guesswork must be used or 'similar' programs measured. Account will also have to be taken of the effect of the protection mechanism on the remainder of the system and the way it fits into the overall design. In some cases, it may only be feasible to use the measures in a qualitative manner to discuss the differences of certain protection mechanisms or strategies.

The first example of the comparison of protection systems compares various implementations of Algol W run-time systems (Wirth and Hoare 1966) and evaluates the value of protection provided by each implementation and the associated cost. The second example illustrates the utility of the model in the system area, i.e., the problems of inter-user protection, by comparing the protection system used in the IBM System/370 DOS/VS operating system (IBM 1973) with an alternative system based on the same mechanism, namely storage keys.

The first example deals essentially with protection within a sequential process. The second tackles inter-process protection in a multiprogramming environment. If there was a system where the whole system and user programs were written in a single language (conceivably the Burroughs B6700 with Algol, though the B6700 in fact uses various differing versions of Algol, but one might find a common denominator), then all protection, both internal and external to a process, would be expressed in that language. The comparison method applied to such a language would have been sufficient to demonstrate the utility of the methodology, but such a system was not available to us.

Further examples, in Chapter 7, illustrate the application of the model to characterise protection in computer architectures and languages which exploit structure for protection purposes. Together, these examples will demonstrate that the model can be applied to almost any level of detail.

Chapter 5

A QUANTITATIVE STUDY OF PROTECTION IN IMPLEMENTATIONS OF ALGOL W

To illustrate the comparison of protection mechanisms, an experiment was conducted to examine the benefit and cost tradeoffs of protection in various feasible implementations of the Algol W language (Wirth and Hoare 1966), (Bauer et al 1968). The experiment was based on a sample of about 100 programs, written in Algol W, collected from a wide variety of people in the Computing Laboratory at Newcastle University.

Algol W is a development of the programming language Algol-60 and is used extensively in the Computing Laboratory for teaching purposes and research work. If the list processing facilities (records and references) are ignored, which they are for the purposes of this study, Algol W can be regarded as a variant of Algol-60, the differences in syntax and semantics being of minor significance. The important attribute of Algol W with regard to this experiment is that it is a block structured language.

The ready availability of a source of Algol W programs was a considerable influence on the selection of this experiment as a test bed of the ideas proposed in Chapter 4. However, a major advantage of this choice was the existence of an Algol W run-time system which has the capability of providing a trace of the execution of Algol W programs in source language terms. This system was developed by Satterthwaite (1972) primarily to assist the development and debugging of programs. It provided the essential basis for a means of gathering statistics on the execution of programs necessary to evaluate the cost and benefit measures.

5.1 Implementations of Algol W

Run-time implementations of block-structured languages generally make use of a stack on which to store program variables and control information, exploiting knowledge of the last-in-first-out creation and deletion strategy of program objects. Two very common styles of implementation perform storage allocation at the block level (Randell and Russell 1964) and the procedure level (Hawkins and Huxtable 1963), (Wichmann 1973) respectively. The aim of this experiment was to examine these two implementation styles from the protection aspect.

The Algol W run-time system provides a stack machine for the execution of Algol W programs. This stack machine is an abstract machine in the sense that it is simulated on a conventional von Neumann type computer. An activation record consisting of three parts (link information area, primary allocation and secondary allocation) is placed on the stack on entry to a block or procedure and is removed at block exit or procedure exit. For the purposes of implementation, a block is considered to be a procedure without parameters.

The link information typically includes pointers to the previous activation record on the stack (dynamic link) and the activation record of the statically enclosing block (static link). The primary allocation contains space set aside for the simple (i.e. not array) variables and descriptors of any arrays declared in the block. If the activation record corresponds to a procedure with parameters, space is also included in the primary allocation for the specification of the actual parameters passed to the procedure. The secondary allocation contains space for the elements of the arrays declared in the block.

Though the sizes of each primary allocation and each link information area can be calculated at compile-time, the size of each secondary allocation is not known until the array declarations within a block have been executed, since arrays may have dynamic bounds. As the exact position of an activation record on the stack cannot be predicted at compile-time, addresses cannot be calculated for program variables in terms of an absolute displacement from the base of the stack. The method of addressing program variables utilised in Algol W is the conventional use of a display and address couples (see (Randell and Russell 1964) and (Wichmann 1973)). A program variable declared in a block is uniquely defined by an address couple (l,d) where l = level of static nesting of the block in which the variable is declared (the lexicographical level) and d = displacement of the variable in the activation record of that block. Display registers point to the activation records of the block currently being executed and the statically enclosing blocks; the address of a variable within the stack is formed by adding the contents of the l-th display register and the displacement d.

This method of addressing permits access to the activation records of the blocks on the static chain (i.e., the statically enclosing blocks) of the block currently being executed. Other activation records on the stack cannot be accessed directly. Strictly speaking, a check should be performed on the displacement contained in an address couple to ensure that it lies within the appropriate activation record. This can easily be achieved if the display registers are of the base and limit form.

Actual parameters, which may lie outside the static chain of activation records, are accessed via descriptors placed in the appropriate accessible activation records. Array elements are

accessed via a descriptor in the primary allocation of the activation record. In Algol W, this descriptor takes the form of a dope vector, which used in conjunction with the subscripts, facilitates the calculation of the address of an array element.

A possible alternative to the above implementation of the Algol W run-time system would be to put an activation record on the stack only on entry to a procedure. This is feasible because, although procedures may be recursive, blocks within a procedure are not. The addressing scheme for storage allocation performed at the procedure level is very similar to the case where activation records correspond to blocks, except that the level of nesting, l, in an address couple is now the level of nesting of procedures. The outermost block of a program is treated as a procedure.

The advantage of this method is that since procedure entry is in general less frequent than block entry, fewer activation records are placed on the stack. The display will therefore be smaller, particularly important if it is desired to keep the display in a limited set of registers, and the cost of maintaining the display will normally be cheaper. However, to be balanced against this saving is the drawback that parts of the stack may be accessible which, according to the rules of the language, should not be accessible. Thus, if through an error in the compiler or a machine malfunction, an address couple describing a location in the theoretically inaccessible region of the stack is presented to the addressing mechanism, the request for access to the stack will in certain cases be granted.

The activation record for a procedure in such an implementation reserves space for all simple variables and descriptors for arrays declared in blocks in the procedure. The secondary allocation is

handled on a block basis, space for arrays is allocated on execution of an array declaration and deallocated on block exit. During execution in a procedure, the addressing mechanism will permit access to the whole of the procedure's activation record, which includes space set aside for variables declared in blocks contained within the procedure. The scope rules of Algol W, however, only permit access to such variables when execution is taking place within the block, in which the variable is declared, or a block enclosed by that block.

The alternative implementation based on procedures has two forms; the first which takes account of parallel blocks and the second which does not. Within a procedure, two blocks at the same level of nesting cannot have variables on the stack simultaneously (in the same activation of the procedure). Thus, such blocks can use the same space within an activation record for declared variables, reducing the size of the activation record. In Figure 5.1, if note is taken of parallel blocks the variable R2 can use the same storage location as I1 (assuming integers and reals both occupy one word) since these two variables can never exist simultaneously in one call of the procedure.

Another feasible implementation is a Fortran-like implementation in which space is reserved at program load time for all simple variables, array and parameter descriptors, and links for nested procedure calls and sequence control. The stack is used solely for the allocation of space for arrays and is necessary to permit arrays to have dynamic bounds. Execution of an array declaration allocates space on the stack for the array and fills in the addresses in the dope vector or descriptor. The space is deallocated on block exit.

```
PROCEDURE A;
 BEGIN
     REAL R1;                         Size of activation record of procedure
     .                                A in procedure implementations
     .
     .
     BEGIN
         INTEGER I1,I2,I3;            a. taking account of parallel blocks
         . . .                           = 4 words + link information
     END;
     .
     .
     .
     BEGIN
         REAL R2;                     b. ignoring parallel blocks
         .                               = 5 words + link information
         .
         .
     END;
     .
     .
     .
END;
```

Figure 5.1

An Algol W procedure and the size of its activation record

This implementation precludes recursion, but is interesting as a theoretical implementation since it provides a base line for the comparison of the three implementations of Algol W: storage allocation performed at block level, storage allocation performed at procedure level taking note of parallel blocks and storage allocation performed at procedure level ignoring parallel blocks. An implementation scheme following the Fortran-like approach, but with special provision for recursion, has been suggested by Sattley (1961).

In a Fortran-like implementation, all variables and arrays are always accessible since absolute addresses are used and no checks are

performed on addresses, except a simple check that they lie within the data area. No display is required, so the overhead of maintaining a display is removed.

To avoid obscuring the important issues in the above discussion of various implementations of Algol W, no mention has been made of the need for working storage necessary for the retention of temporary intermediate results in the evaluation of expressions. The compiler can calculate the amount of storage required to evaluate each expression in a program and hence deduce the maximum amount of storage required in each block or procedure. The appropriate amount of extra space can then be allocated in the primary allocation of each block or procedure. Alternatively, it is possible to arrange that the workspace for temporary information is taken on top of the stack. Wichmann (1973) reports that examination of post mortem dumps from the KDF9 compiler for Algol-60 showed that working storage variables were much less numerous than user declared variables which in turn were less than array storage.

An assumption made in discussing accessible areas of the stack is that array descriptors and actual parameter descriptors only allow access to the appropriate array elements, and actual parameters respectively. In the case of array descriptors, there is little problem since array elements can be addressed relative to a display register and thus any array element address is checked that it lies within the activation record to which the display register points. An actual parameter descriptor must contain an address relative to the base of the stack, as the parameter may not lie within the areas of the stack described by the display. To prevent access being gained to other supposedly inaccessible locations on the stack, it is necessary to ensure that a program cannot alter, even inadvertently,

the contents of the actual parameter descriptor. This point is taken up again in Section 5.5.

5.2 Benefit and Cost Measures

The purpose of this experiment is to compare the cost and benefit of protection in five implementations of Algol W. 'Protection', in this experiment, is defined to be accessibility to areas of the run-time stack. In other words, if a suitable address is presented to the addressing mechanism, will access be granted to a specific location? Access to files and devices, etc., and available instruction sets are not considered.

The implementations being analysed are:

- I1 an implementation based on a stack and performing storage allocation at the block level
- I2 an implementation based on a stack and performing storage allocation at the procedure level taking account of parallel blocks
- I3 as for I2, but no account taken of parallel blocks
- I4 a Fortran-like implementation with static allocation of space for all simple variables and array descriptors and using a stack for array allocation, no account is taken of parallel blocks within procedures
- I5 the actual Algol W implementation used at Newcastle, based on a stack and performing storage allocation at the block level, but not checking access attempts to link information or extra words on the stack used for boundary alignment.

Each implementation, except I5, is treated as an idealised abstract machine. Thus, certain details concerned with implementing these

machines on actual computers, which would otherwise detract from the essential aspects of this study, can be ignored.

Practical implementations do exist corresponding to I1,I2 and I3. I4 represents a worst case situation where everything is accessible, and thus provides a lower limit for reference purposes. The actual implementation of Algol W on the IBM System/360 computer (I5) is based on the 'ideal' implementation I1, which represents 'pure' Algol W, reflecting the rules of the language. Problems, such as the limited numbers of registers available for use as a display, the method of addressing employed on System/360 and boundary alignment in main memory, have forced a less than ideal practical realisation.

In all implementations, it is assumed that the display registers are of the base and limit form (even though this is not the case with the actual Algol W system), so only the allowed activation records can be accessed. The link information in implementations I1,I2,I3 and I4 is presumed to be inaccessible to all instructions except those which change the context. A device such as that suggested by Lauer and Snow (1972) would achieve this.

Temporary working storage is ignored and it is assumed that array and parameter descriptors only allow access to the appropriate part of the stack, and that some mechanism exists, such as tag bits, for preventing the contents of descriptors being changed by an executing program. We also ignore the possibility of preventing access to a variable by declaring a new variable with the same name. Further, the initial analysis does not distinguish between different types of access to the stack, namely read or write.

To perform the analysis, benefit and cost measures have to be derived which are then evaluated using statistics from a sample set of programs. Before the measures can be derived, however, it is

necessary to consider the nature of domains in the Algol W environment.

5.2.1 Domains in Algol W

In terms of the rules of the Algol W language (ignoring the facility which permits the dynamic creation and deletion of records), once a block has been entered and any array declarations contained within the block processed, the set of accessible objects is well defined and does not vary during the execution of the remainder of the block. The set of accessible objects will in general alter when a transfer of control is made between blocks. (For the purposes of this discussion, the term 'block' encompasses blocks, procedures and implicit subroutines used to evaluate actual name parameters.) The appropriate semantic unit to which to tie domains is therefore the block. If domains were keyed to procedures, inconsistencies would arise in certain cases. In particular, a block executing in the domain of its containing procedure would be allowed to access variables declared within other blocks within the procedure even if they were inaccessible according to the rules of the language.

During execution of an Algol W program, the theoretically accessible set corresponds to the domain described above. This is defined by the scope rules of Algol W which specify that the variables accessible at a given point in the execution of an Algol W program are those declared in the block currently being executed and any statically enclosing blocks, together with any actual parameters passed to enclosing procedures.

In this comparison of different implementations of Algol W, we are interested in the accessible set as defined by each of the implementations I1,I2,I3,I4 and I5 at any point in the activation

of an Algol W program and how these accessible sets compare with the theoretically accessible set.

The run-time system I1, which directly implements the rules of the Algol W language, places an activation record on the stack for each activation of a block. The accessible set at any time is defined by the display, being composed of the accessible regions of the stack pointed to by the display plus any other areas of the stack indicated as accessible by descriptors contained within the directly accessible activation records. In the case of I1, the accessible set and the theoretically accessible set are identical.

Implementations I2 and I3 put an activation record on the stack on entry to a procedure or implicit subroutine. Entry to a block may cause an adjustment to be made to an activation record if an array is declared within the block, but does not cause the creation of a new activation record. In implementations I2 and I3, the accessible set at any time during the execution of an Algol W program is defined by the primary and secondary allocations of activation records pointed to by the display, plus any actual parameters passed to procedures on the static chain. Each accessible set in implementations I2 and I3 will generally correspond to a number of domains as defined by Algol W.

The execution of a program in the Fortran-like implementation I4 occurs in what it is most convenient to regard as a single accessible set. The accessible set is composed of a static part, including all simple variables and array descriptors, and a dynamic part consisting of a stack on which arrays are implemented. It is assumed that an array descriptor is set to null, thereby denying access to the stack, except during the period of existence of the array.

The actual implementation of Algol W, I5, follows the essential ideas of I1, but is limited by the architecture of the 360 computers. The display is maintained in a subset of the machine's general registers which have no limit extension. In practice, therefore, the majority of the stack, if not all of it, is usually accessible. However, presuming that the display is maintained in base and limit registers allows a crude comparison to be made between the actual Algol W implementation and the ideal I1. The main effects being studied are allowing the link information to be included in the accessible set and the effect of having to align arrays and activation records on double word boundaries.

The following simplifying assumptions have been made in the above discussion.

1. Space required for the temporary storage of intermediate results of expression evaluation has been ignored. This is partly due to the problem of calculating the amount of temporary storage required to evaluate an expression, since it will depend on the architecture of the machine being used.

2. Standard functions (e.g., square-root, sine) have been treated as indivisible machine instructions causing no switch in domain. This is justified on the grounds that standard functions make no reference to the current domain except to the parameter(s) passed to the function.

3. During a domain switch, it has been assumed that no access is made to source language variables. In other words, the fact that execution of array declarations can involve expression evaluation is ignored. Similarly, the initialisation of VALUE parameters and the setting of

RESULT parameters are also ignored. The number of references to source language variables involved in these actions is generally very small compared to the number of references made within a block and hence this simplifying assumption is reasonable.

4. Algol W specifies that a control variable in a FOR statement is only accessible within the statement following the DO and can only be read, not written. This is checked by the compiler but not by the run-time system. Therefore, FOR loop control variables have been treated as being declared in the block enclosing the FOR statement and accessible throughout the block.

5. All identifiers in a program are assumed to be unique, so that there is no possibility of redeclaration of an identifier already used within a program, which would cause a hole in the scope of the original variable with that name.

6. In Algol W, an actual name parameter which is a simple variable is passed by reference, thus avoiding the activation of an implicit subroutine on each reference to the formal parameter. This optimisation is assumed to occur in all implementations.

5.2.2 Benefit Measures

To evaluate benefit measures of the form suggested in Chapter 4, it is necessary to define the 'size' of an accessible set in the context of Algol W. The simple-minded approach of indicating the size of an accessible set by the number of simple variables, arrays and parameters contained in the set does not fairly represent an

accessible set, since an array is usually regarded not as a single object but as a collection of array elements. Individual array elements could be counted as objects, but problems would still arise in assessing the size of an accessible set in implementation I2, where space on the stack, which is not currently being used to store objects, is accessible. A more realistic approach, and one which avoids this last problem, is to consider an accessible set to be the total accessible area of the stack, i.e., the size of the current accessible set is expressed as the number of bytes on the stack which could be accessed if suitable addresses were presented to the addressing mechanism.

Since accessible sets are being measured on an IBM System/360 computer, some machine dependence will affect the measurements, e.g., descriptor sizes and space allocated to variables of different types. This effect is unlikely to be marked since it is only with logical variables, where for convenient manipulation a byte is allocated to each logical variable rather than one bit, that the 360 architecture has had a strong influence, otherwise space allocated to variables of different types is fairly conventional.

The calculation of the size of an accessible set in each of the implementations I1, I2, I3, I4, and I5 is indicated in Table 5.1. The size of the theoretically accessible set, as defined by the Algol W language (subject to the simplifications noted earlier), is achieved by implementation I1.

Let a program written in Algol W and run on implementation I1 execute in domains $d_1, d_2, \ldots, d_n$ $(n \geq 1)$ and the sizes of the corresponding accessible sets be $s_1, s_2, \ldots, s_n$. The total number of distinct domains in which the program executes (= total number of activation records placed on the stack) is $n \geq 1$. Execution in a domain may take the form

Table 5.1

Definition of the size of the current accessible set during execution of an Algol W program in each implementation of Algol W.

Implementation	Size of Accessible Set
I1	The sum of the allocation of each block, procedure or implicit subroutine on the static chain of the activation record on top of the stack, where the allocation of an activation record is composed of:
I2	The sum of the allocation of each activation record on the static chain of the activation record on top of the stack, where the allocation of an activation record is composed of:

I1

The sum of the allocation of each block, procedure or implicit subroutine on the static chain of the activation record on top of the stack, where the allocation of an activation record is composed of:

primary allocation
- (space for declared variables of simple type
- (space for descriptors of declared arrays
- (space for value/result parameters
- (space for descriptors for formal name parameters
- (space for descriptors for array parameters

secondary allocation
- (space for elements of declared arrays

space for actual name parameters not otherwise accessible

space for elements of (sub)arrays passed as parameters and not otherwise accessible

I2

The sum of the allocation of each activation record on the static chain of the activation record on top of the stack, where the allocation of an activation record is composed of:

space for the primary allocation - the maximum required by the procedure calculated by summing the primary allocation of each block (as defined in I1 above) in the procedure, taking account of parallel blocks.

space for elements of declared arrays, i.e., <u>all</u> arrays currently declared within <u>the</u> procedure

space for actual name parameters not otherwise accessible

space for elements of (sub) arrays passed as parameters and not otherwise accessible

Table 5.1 (continued)

Implementation	Size of Accessible Set
I3	As for I2, except that the primary allocation is calculated for each procedure on the basis that no account is taken of parallel blocks within the procedure
I4	The sum of the size of the primary allocation of each procedure declared in the program, plus the space taken by the array elements of all arrays currently existing on the stack.
I5	The sum of the allocation of each activation record on the static chain of the activation record on top of the stack, where the allocation of an activation record is composed of: space for the activation record space for actual name parameters not otherwise accessible space for elements of (sub) arrays passed as parameters and not otherwise accessible

of periods of execution separated by periods when the program is executing in other domains.

When the same Algol W program is executed by implementation I_j (j = 1,2,3,4,5), let the sizes of the accessible sets corresponding to the domains $d_1, d_2, \ldots, d_n$ be $t_{j1}, t_{j2}, \ldots, t_{jn}$. Whereas domains $d_1, d_2, \ldots, d_n$ are all distinct in I1, the corresponding accessible sets in the other implementations may not be distinct.

A first benefit measure for the implementations of Algol W based on the execution of an Algol W program is:

$$D_j = \frac{\sum_{i=1}^{n} \frac{s_i}{t_{ji}}}{n} \qquad j=1,2,3,4,5$$

In this measure, no weights have been applied to the domains. D_j can be evaluated for each implementation for a given program, though for I1, D1 = 1.

A further benefit measure can be developed by weighting each domain by some characterisation of the time spent in the domain. The statistics necessary to evaluate the benefit and cost measures are gathered from executing Algol W programs by an interpretive technique. This precludes accurate measurement of time spent in a domain since it is impossible to exclude the time required for interpretation. Instead, the number of references made in each domain to source language variables and array elements is used. If the number of references made while executing in domain d_i is r_i, the weighted benefit measure is:

$$B_j = \frac{\sum_{i=1}^{n} \frac{r_i \, s_i}{t_{ji}}}{\sum_{i=1}^{n} r_i} \qquad j=1,2,3,4,5.$$

The factor r_i is used to weight domains because it gives some indication of the 'time' spent in domain d_i from the Algol W or

source language point of view.

In counting references to source language variables and array elements while in a domain, the following are ignored:

references to constants, since constants are stored with the compiled code or in immediate instructions

references to array descriptors

references to parameter descriptors

references to anonymous variables (temporary results)

references to predeclared variables of the language

It is possible that certain domains may be entered by the executing program but have a reference count of zero. An implicit subroutine, for example, may calculate the value of an expression involving predeclared variables and constants. Such domains contribute to the measure D_j, but are discounted in B_j. Normally, such domains exist for performing specialist functions and are relatively infrequent.

These measures are applicable to implementation schemes where arrays are implemented off the stack, as in the case of Algol on the Burroughs B6700 (Organick 1973), as well as the type of implementation being considered in this chapter, where all arrays are kept on the stack.

5.2.3 Cost Measures

The mechanisms which contribute to protection in the five implementations of Algol W being considered also provide other advantages. For instance, addressing variables by lexicographical level and displacement, in conjunction with a display, facilitates recursion. So, it is difficult to distinguish and assess precisely the costs attributable to protection. Nevertheless, it is possible

to achieve a crude quantification of the relative costs of each implementation with regard to protection.

Costs are incurred both at compile-time and run-time. Compile-time costs will depend somewhat on what is provided by the run-time system. For example, the provision of special hardware or an interpretive run-time system with domain switching instructions might ease the task of the compiler. A particular difference between implementations I2 and I3 is that the compiler for I2 must take account of parallel blocks within each procedure in calculating the size of a procedure's activation record, whereas the compiler for I3 can ignore parallel blocks. It is thought that any differences in compilation costs between the five implementations would be minor and completely masked by the general overhead of compilation. Compilation costs are therefore ignored entirely and we concentrate on run-time costs.

Run-time costs fall into three categories: extra processing time, extra storage space and the provision of special hardware. There is likely to be a tradeoff between these three costs; for instance, the provision of extra hardware is likely to reduce the increased processing time and storage requirements of the protection mechanism.

Since implementations I1, I2, I3 and I4 are being regarded as abstract machines, no attention is paid to the precise nature of how the implementations could be achieved on an extant or proposed computer. The tradeoffs amongst the costs are therefore not examined and the question of the provision of special hardware is ignored. Any extra storage space required will depend to a great extent on the machine used but it is assumed to be minimal. Costs are therefore calculated in terms of extra processing time.

First, the costs of protection in implementation I1 are examined under the six headings proposed in Chapter 4. The differences occurring in implementations I2, I3, I4, and I5 are then described and two cost measures proposed.

Domain creation.

On entry to a new domain, it is necessary to place a new activation record on the stack and set up the appropriate linkage (static and dynamic) to other activation records on the stack. Descriptors located in the primary allocation and representing parameters and arrays will be set to describe the corresponding objects. The secondary allocation will be obtained as array declarations are executed. The display will be adjusted as part of the domain switch to this domain.

Domain deletion.

A domain is deleted when the program leaves the domain and either terminates execution or enters a domain from which it is impossible to return to the domain just left. The space occupied by the activation record corresponding to the domain is freed for future use.

Domain maintenance.

A domain remains in existence as long as the corresponding activation record remains on the stack, i.e., has not been deleted. Apart from the activation record remaining on the stack, the only cost involved in maintaining a domain is to ensure that when a domain is left by an executing program on a temporary basis, means are provided for returning to execute in that domain. In the

majority of implementations, this is achieved by means of the pointers in the linkage area of each activation record. The cost of domain maintenance can therefore be ignored.

Domain switch.

A domain switch occurs on:

a. Block entry or exit

b. Procedure entry or exit

c. Implicit subroutine entry or exit

d. Non-local GOTO

A domain switch implies an adjustment to the display to alter the set of accessible activation records. Entry to or exit from a procedure or implicit subroutine will generally cause a more substantial adjustment to be made to the display than block entry or exit. A non-local GOTO will always be directed to a visible label and hence will only result in display entries being discarded. The display can be reset on a domain switch from information in the link area of the activation record corresponding to the new domain of execution (c.f., (Hauck and Dent 1968)).

Domain changes.

In implementation I1, domains correspond to blocks. The acquisition of space for all objects declared in a block is treated as being part of domain creation. There is no possibility of dynamic variable creation or deletion within a block once an activation record has been established, thus there are no domain changes.

Protection checking.

Protection is enforced by the use of address couples which allow little possibility for protection violation. For an address couple (l,d), l must be less than or equal to the index of the current highest numbered active display register and d must be less than or equal to the limit portion of the l-th display register. The amount of checking which has to be performed is therefore minimal.

Implementations I2 and I3 can be considered together since they will have the same run-time costs. A new activation record is created only on entry to a procedure or implicit subroutine and activation records are deleted in the same manner as in I1. Domain switches cause similar changes to be made to the display, though they will normally occur less frequently than in I1 and involve fewer display registers. Protection checking is performed in precisely the same way as in I1. The real difference between I2, I3 and I1 occurs with array declarations in a block which cause an increase to be made in the size of the secondary allocation in the activation record on top of the stack. To enable the stack to be retracted by the correct amount on block exit, it is usual to maintain a vector within the link area of the activation record of each procedure, in which is recorded the limit of secondary storage for each block level within the procedure. On block exit it is then a simple matter to reset the size of the secondary allocation.

The Fortran-like implementation, I4, is very different from I1, I2, or I3. A single activation record exists in which the Algol W program carries out the whole of its execution. Apart from establishing this activation record prior to program execution, there

is no creation of activation records, similarly with deletion. There is no protection checking of the form described above since absolute addresses are used, but we assume a check to ensure that addresses lie within the program data area. All array declarations cause a change to the activation record in a manner similar to that for I2 and I3.

Implementation I5 is assumed to have the same costs as I1.

Activation records are a convenient means of allocating storage space and this is their prime function; the cost of actually putting activation records on the stack is therefore ignored. Array allocation occurs in all implementations and involves similar actions in all cases; since it is part of storage allocation this cost is also ignored. Protection checking occurs in all implementations and we assume this checking is done by hardware in parallel with memory referencing, so it contributes nothing to extra processing time. Protection cost is therefore assessed as the cost of maintaining the display. Since each Algol W program will incur different costs, comparing absolute costs is meaningless, therefore, costs are compared relative to that of I1 for each program.

In the simpler approach, we assume that the cost of a domain switch predominates and define the cost K_j of implementation I_j as being the number of domain switches occurring during execution of an Algol W program on implementation I_j divided by the number of domain switches which would occur if the program were executed on implementation I1. Each domain switch is assumed to incur unit cost in processing time. This provides a crude basis for comparing the costs of each implementation.

The other approach is to make a detailed analysis of the cost, breaking the maintenance of the display down into a number, say q,

of primitive actions each with associated cost c_i $(i = 1,2,\ldots,q)$ and to measure the number of times $(n_{ji}, i=1,2,\ldots,q)$ each primitive is invoked by implementation I_j $(j=1,2,3,4,5)$ during the execution of an Algol W program. The cost L_j of implementation I_j relative to that of I1 for the execution of a given program is then:

$$L_j = \frac{\sum_{i=1}^{q} c_i n_{ji}}{\sum_{i=1}^{q} c_i n_{1i}} \qquad j=1,2,3,4,5$$

Four primitive actions can be identified (assuming the existence of a top-of-stack pointer) and are as follows:

1. Setting up the linkage information and top-of-stack pointer on the creation of a new activation record.
2. Resetting the top-of-stack pointer on the deletion of a domain.
3. Altering one display register on a domain switch. (A domain switch will generally involve more than one display register.)
4. Setting or resetting the limit extension in the current top display register and the top-of-stack pointer on a domain change.

The cost of each primitive action is c_1, c_2, c_3, c_4 respectively and is assumed constant and to have the same value for each implementation. This second cost measure is still somewhat crude, but enables a better assessment of relative costs to be made than the first measure.

5.3 Method of Gathering Statistics and Evaluating Measures

The statistics required for the evaluation of the cost and benefit measures concern the dynamic execution of Algol W programs rather than the static program text. Therefore, it is necessary to gather the statistics during the execution of each Algol W program. There are three obvious techniques by which this may be done: hardware modification or monitoring, interpretive execution or emulation, or compiler modification; all of which have attendant problems.

For this study, hardware modification was not feasible and there was no hardware monitor available.

If the Algol W system was coded as an emulator which ran on a microcoded computer, it would, in theory, be possible to modify the emulator to record the appropriate statistics during execution. This method has two advantages; (1) emulators are generally relatively easy to alter and (2) the overhead of gathering the statistics would be low. Thus, the total execution time of an Algol W program would only be increased by a small proportion. Compilers which use interpreters to execute the compiled program also make the collection of such information a straightforward inexpensive operation, since the time required to gather the statistics is swamped by the time taken to interpret. Wichmann (1970),(1973) utilised the interpreter of Randell and Russell (1964) in this way to obtain dynamic statistics from Algol-60 programs. Wortman (1972) used an interpreter which ran on an IBM System/360 computer to study the design of a proposed machine for executing SPL programs (a dialect of PL/1).

The third technique involves modifying the compiler so that code is inserted into the object program at suitable points to record the activities of the executing program - this method has been used by Brundage and Batson (1974) to obtain symbolic address traces of Algol-60 programs.

In developing a tracing and debugging system for Algol W, Satterthwaite (1972) wrote what amounts to a simulator of an IBM System/360 computer. When tracing of a program is forced by the programmer, execution of the program is simulated and a trace produced, in terms of the original source program, which includes details of the statements executed and any variables which are read or altered. Since this simulator already existed, it was decided to use a variation of the interpretive technique to gather statistics rather than attempt to make alterations to the compiler which would have been a major undertaking. When programs are traced by the Algol W run-time system, the execution time of programs increases by a multiplicative factor of 50 to 150. This huge overhead of tracing meant that the addition of code to gather statistics caused little further degradation, but had the disadvantage that some interesting Algol W programs could not be analysed because their execution time under simulation was prohibitive.

To achieve flexibility, since it was envisaged that as the experiment progressed further statistics might be required, the general types of statistics to be gathered were identified and the most suitable places in the simulator for collection pinpointed. The statistics fell into four broad classes according to the time at which each statistic was gathered. These times were: initial program entry, reference to a source language variable, domain switch and final program exit. Having identified the places in the simulator where these statistics could be gathered, a skeletal measurement system was written to interface with the Algol W run-time system at these points. To obtain further measurements or to alter the statistics being obtained, it was only necessary to add code to or alter code in the procedure bodies of the skeletal measurement system.

This approach, as opposed to inserting odd pieces of code in the run-time system itself, was found to be effective. Once the interface to the run-time system had been established, little further change was made to the code of the run-time system.

The Algol W system at Newcastle appends to the end of each code segment produced by the compiler a name table containing details of all variables, arrays and parameters declared within the corresponding block or procedure. This information is necessary for diagnostic and tracing purposes and allowed the size of each accessible set to be calculated. When a program has been loaded into memory, but prior to execution, the measurement system performs an analysis of the static structure of the program. To increase efficiency, the execution of the program is mirrored on another stack on which information, useful in the evaluation of the measures, is kept to save it having to be recalculated each time it is needed.

Evaluation of the cost and benefit measures could either be done on a separate run using the statistics from the execution of the program or during the actual execution of the program. The latter method was adopted since it added little further overhead to the time of execution of programs and avoided problems of storing the statistics and later evaluating the measures.

It is perhaps interesting to note that what is being obtained by the measurement system, though not permanently recorded, is a source language trace of Algol W programs. In the study of paging algorithms, it has been common practice to analyse detailed address traces of programs, but as far as the author is aware, very little work has been done with source language traces of programs. A recent example of the use of a source language trace is the work of Brundage and Batson (1974) in examining the advantages of using

associative registers on a computer like the Burroughs B5500. A source language trace of the execution of a program reflects the structure contained in the original program which it is usually impossible to regain from an address trace. It seems plausible to expect that the behaviour of programs written in Algol W or other Algol-like languages is much better discussed in terms of Algol level trace data, and equally that such data can be usefully employed for the design of high-level language machines.

5.4 Evaluation of the Measures

To evaluate the measures, some 100 Algol W programs were gathered from members of the computing community at Newcastle University. The majority of the programs were written by undergraduates in all three years of the degree course in computing science, and M.Sc. and Ph.D. students in computing science. This set of programs is representative of the programs submitted to the Algol W compiler at Newcastle.

Included in the sample were solutions to exercises set by lecturers, projects done by undergraduates in their third year and programs written as part of research work being carried out by post-graduate students and members of staff. The applications themselves were in general of a scientific nature including numerical analysis, simulation, combinatorial problems and sorting. The programs could be classified as small to medium in size and complexity, containing from 1 to 55 blocks and procedures. The number of statements contained in the programs ranged from approximately 20 to 500 and the number of references to source language variables generated during execution varied between 381 and 1,134,434.

The programs themselves were gathered on an ad hoc basis rather than altering the compiler to save systematically all syntactically

correct programs on magnetic tape or disk, a method used by Wichmann (1970) and Wortman (1972). The majority of programs submitted to the Algol W compiler arise from introductory programming courses given within the University. These programs tend to be short, typically 10 - 15 lines, and contain only one block and no procedures. Such programs do not warrant the analysis being performed here, so it was decided to gather programs by personal contact to obtain a wider variety.

Statistics were gathered from the execution of these programs and used to evaluate the cost and benefit measures for each implementation. All the measures could be evaluated using the single set of statistics. When interpreting the results presented in this section, it is necessary to bear in mind the environment, namely one of teaching and research.

The mean and standard deviation of the benefit measures obtained from the analysis of the execution of the programs are shown in Table 5.2.

The values of the benefit measures are shown to 4 decimal places to illustrate the small difference between certain of the measures (particularly D_2 and D_3, B_2 and B_3). The benefit measures of implementation I1, namely D_1 and B_1, both have the value 1, since I1 implements the rules of the Algol W language. In general, $D_2 \leq 1$, $B_2 \leq 1$ and for most programs $D_2 < 1, B_2 < 1$. It can be seen from Table 5.2 that the difference between B_2 and B_1 for the sample of programs is relatively small. In implementation I2, the extra accessible locations in the implemented accessible set occur in the primary allocation (i.e., equivalent to simple variables) and only in certain special cases in the secondary allocation (i.e., array elements). For the majority of domains, it is to be expected that the secondary allocation in the accessible sets of I1 and I2 will be the same, and since

	D_1	D_2	D_3	D_4	D_5
mean	1.0000	0.8909	0.8905	0.8138	0.7182
standard deviation	0	0.2309	0.2308	0.2510	0.1891

	B_1	B_2	B_3	B_4	B_5
mean	1.0000	0.9686	0.9682	0.8938	0.7927
standard deviation	0	0.1007	0.1008	0.1545	0.1595

Table 5.2

Benefit measures obtained from analysis of the sample Algol W programs

storage occupied by array elements is usually much greater than that occupied by simple variables (Wichmann 1973) it is not surprising that the values of B_1 and B_2 are very close.

The difference between B_2 and D_2 can be explained by the effect of the weights used with domains in the B-measure. From studying the program texts, it was noted that a large proportion of the programs in the sample used dynamic arrays typically in the form:

```
A:  BEGIN
        INTEGER N;
        READ(N);
B:      BEGIN
            INTEGER ARRAY MAT(1::N,1::N);
            :
        END;
    END.
```

In implementation I2, the two blocks A and B would be represented by one activation record on the stack. When executing in block B, the accessible sets defined by I1 and I2 would be the same but when executing in block A the accessible set defined by I2 would include not only the variable N (which is the sole contents of the accessible set defined by I1) but also the array descriptor for the array MAT and any other variables declared in block B. So the accessible set for the block A in implementation I2 would be much larger than the theoretically accessible set. In benefit measure D_2, blocks A and B contribute equally to the measure. However, in B_2 each domain is weighted by the number of references to source language variables occurring within the domain, and since the number of references occurring in block B will usually be very much greater than in block A, the effect of the implemented accessible set for block A will have a much more marked effect on D_2 than on B_2.

In implementation I3, since no account is taken of parallel blocks within procedures, activation records will, in general, be larger than those of I2 and hence $D_3 \leq D_2$ and $B_3 \leq B_2$. Apart from the most trivial programs, $D_3 < D_2$ and $B_3 < B_2$, but one would expect the differences to be small. This is borne out by the measurements made of the sample set of programs, since the difference between D_2 and D_3 and that between B_2 and B_3 is only just discernible. To some extent, this is probably a reflection on the fact that most programs in the sample were small to medium in size and not complex in nature.

The Fortran-like implementation will have a benefit measure of 1 only for programs with one block and no procedures. Since all space for the primary allocation of all blocks is allocated at program entry $D_4 \leq D_3$ and $B_4 \leq B_3$. Apart from the trivial programs mentioned above, the inequality is strict in both cases. The values of D_4 and B_4

are considerably better than might have been expected, indeed I4 outperforms I5 very noticeably (the benefit is somewhat greater and as will be seen the cost is far lower). The probable reason for this is that the total primary allocation for all blocks declared in a program is often small compared to the total size of arrays accessible at any instant during execution of a program.

The measures D_5 and B_5 illustrate the effect of the link information area in each activation record being accessible and the necessity for aligning activation records and arrays on double word boundaries. As in the case of I4, these effects are tempered by the large arrays declared in most of the programs contained in the sample.

The values of the standard deviations of the benefit measures for the sample of programs indicate a fair variation in the values of the measures for individual programs. With a sample of some 100 programs collected from about 70 different authors, and bearing in mind that these are empirical measures, such a wide variation is almost inevitable. However, the measures do indicate the essential aspects of the different implementation strategies. The larger standard deviations in the case of the D-measures can probably be explained along lines similar to that used to explain the difference between B_2 and D_2, namely the effect of domains with accessible sets that differ greatly in I1 and I2.

The first cost measure K, the number of domain switches for each implementation relative to the number of domain switches in implementation I1, are shown in Table 5.3. Costs K_2 and K_3 are the same, as precisely the same actions are involved in implementations I2 and I3 at run-time. The cost of I5 was assumed to be the same as that of I1. K_4 is very small because it was assumed that there were only two domain switches in the case of implementation I4, namely program entry and program exit.

	K_1	K_2	K_3	K_4	K_5
mean	1.0000	0.7968	0.7968	0.0411	1.0000
standard deviation	0	0.3112	0.3112	0.1094	0

	L_1	L_2	L_3	L_4	L_5
mean	1.0000	0.8524	0.8524	0.3389	1.0000
standard deviation	0	0.1742	0.1742	0.1473	0

Table 5.3

The mean and standard deviation of cost measures K and L.

K_2 gives an indication of the cost saving of implementation I2 over I1, though there was a wide variation between individual programs.

In the evaluation of cost measure L (see Table 5.3) the costs of the individual primitives were each taken to be 1 (i.e., $c_1=c_2=c_3=c_4=1$) since each primitive essentially involved setting the contents of a register. The individual c_i's are implementation dependent, but in this chapter we are concerned with implementation schemas rather than the precise details of how each implementation is mechanised, and so we take a simple approach here. L_4 has a significantly higher value than K_4 because the execution of an array declaration involves a cost to the Fortran-like implementation in the L-measure. Similarly, the cost of accommodating array declarations in blocks, as opposed to procedures, tends to make the cost L_2 higher than K_2.

In Figure 5.2 two cost-benefit spaces are shown; cost K against

benefit D and cost L against benefit B. In both spaces, it is almost impossible to distinguish between implementations I2 and I3. I4 appears to give a fair amount of protection at low cost, though this is largely due to the effect of arrays as discussed earlier. In terms of run-time costs, it would appear well worthwhile using implementation I1 to achieve the greatest possible amount of protection.

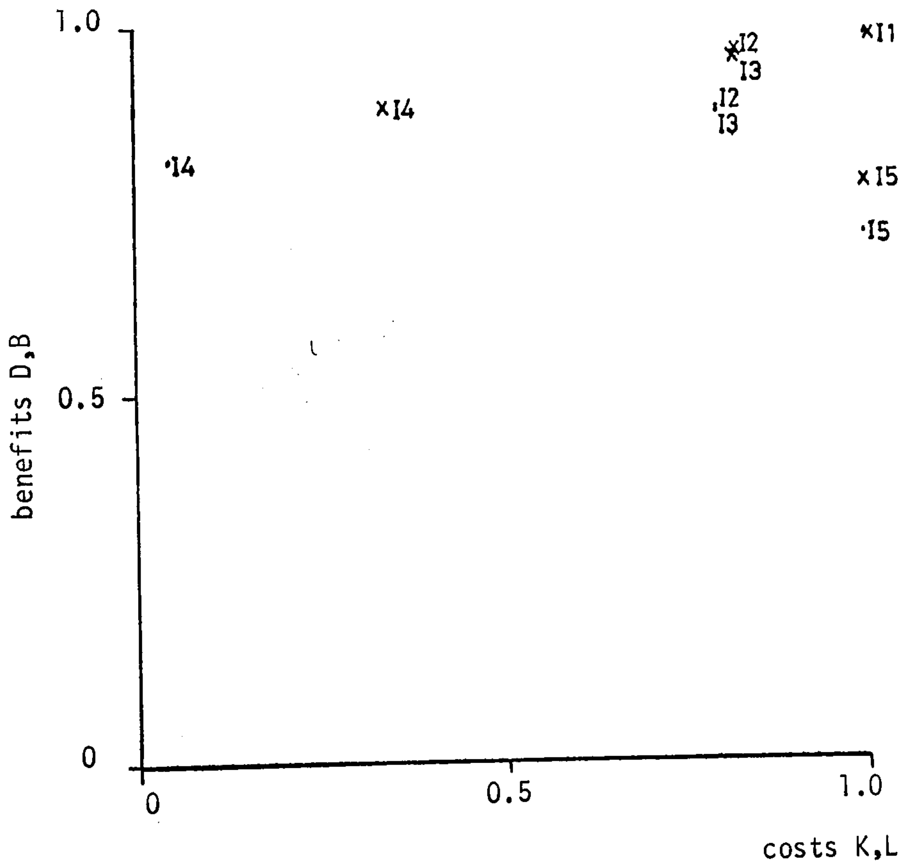


Figure 5.2

Cost-benefit spaces: K against D indicated by . and L against B indicated by x

The significant improvement possible in the protection afforded by the Algol W run-time system in use at Newcastle, I5, is also illustrated in Figure 5.2. The changes necessary to obtain this improvement are making the registers forming the display into base and limit type registers, removing the necessity for aligning arrays and activation records on double word boundaries (a restriction imposed for technical reasons on the IBM System/360 computers but relaxed on the 370 series) and making the link information inaccessible to all but the context-changing instructions.

5.5. Extensions to the Study

A further point in cost-benefit space could be obtained by examining the implementation of Algol W which allocates storage on the basis of a program-level rather than a block or procedure. A program-level is a unit of the program which does not recurse within itself. This implementation is a hybrid of I2 and I4, it minimises the number of activation records which have to be placed on the stack but still allows recursion. To evaluate the cost and benefit measures for this implementation, it is necessary to detect all recursive procedures within a program. The only practical way to do this is by altering the compiler and since all other measurements were carried out by the run-time system, it was decided to omit this implementation from the study.

An extension to the anlysis would be to recognise different types of operation (e.g., read and write access) and the accessible set available to an operation of a given type. It would be an advantage, for instance, to flag all descriptors contained on the stack as being read-only, to prevent the possibility of a program gaining access to 'inaccessible' areas of the run-time stack by

altering a descriptor. In Algol W, the control variable in a FOR statement is defined to be read-only within the body of the FOR statement and this could be catered for in a similar manner.

If a machine with a tagged architecture (e.g., Burroughs B6700) was being considered, operations could be classified according to whether they operated on integers, reals, strings etc. The type of operation would restrict the accessible set to be the subset of the implemented accessible set containing objects of the appropriate type. This scheme would also enable the set of allowed domain crossings from a given domain (i.e., possible procedure calls and block entries) to be included in the analysis if such domain crossings were represented by descriptors.

A different direction in which an analysis such as the one contained in this chapter could be taken is the evaluation of alternative scope rules for Algol-like languages. This would be more in the way of a study of how extra information could be supplied so that the domain as defined by the language matches more closely the objects actually used by a program. Proposals such as partitions in LIS (Ichbiah et al 1973), and ways of avoiding the automatic accessibility of global variables by an inner block (Wulf and Shaw 1973) could be assessed in specially written programs.

The inclusion of the referenced set R in the protection model presented in Chapter 4 and the collection of symbolic trace information (i.e., trace information at the source program level), as is required for the evaluation of the cost and benefit measures, opens up the possibility of studying program locality. If domains are a close match to a program's requirements during execution, they can be used to predict the information which a program will need to access during the time the program executes in that domain. Madison and Batson (1975)

have studied program locality in the case of Algol-60 programs on a Burroughs B5500 and the implications for modelling program behaviour and memory management in virtual memory systems.

5.6 Evaluation of the Experiment

Having performed the empirical analysis of different Algol W run-time implementations, it is necessary to question the utility of the model and methodology as put forward in Chapter 4, and the usefulness of the results of the experiment.

The applicability of the model and methodology to some, if not many, situations has been demonstrated by this experiment. The employment of the methodology has provided a quantitative comparison of the benefits and costs of protection in alternative implementations of Algol W. The results could be used, for instance, as a quantitative basis for part of an assessment of run-time strategies in the implementation of a language similar to Algol W. It must be noted, however, that the results are empirical and reflect the environment in which Algol W is used at Newcastle. In another situation, e.g., a commercial environment, programs may exhibit a significant difference in the way they are written and use of a sample of programs from such an environment might lead to some variation in the results. Even so, the sample of programs used in this experiment was taken from a wide selection of people in the computing community at Newcastle, which compares favourably with the study conducted by Wortman (1972) into the design of an SPL machine. Wortman used programs from an introductory programming course, so his sample was biased by repetition of the same or very similar programs since the programs represented solutions to a collection of about 10 relatively easy problems.

The results of this experiment are largely unsurprising but argue in favour of a run-time system based on storage allocation at the block level. This study has really been concerned with accidental errors such as hardware induced errors. A weighting of important domains or significant objects within each domain may change these results and their applicability. If one could identify potential 'small' violations which if achieved made it possible to access all objects in the system, and gave them a heavy weighting, evaluation of the measures would be of use in penetration studies, like that carried out by Belady and Weissman (1974).

Apart from the main goal of achieving a quantitative assessment of protection in different implementations of Algol W, the methodology has lead to a number of beneficial side effects. An obvious one is that the analysis has drawn attention to possible improvements to the Algol W run-time system currently in use. The analysis could be extended to yield a cost and benefit assessment of the effect of each proposed improvement.

The results presented here are essentially global measures of certain program behavioural characteristics, and we have shown how they can be used to evaluate different implementations of Algol W from the protection point of view. One important question concerns the extent to which the sample of programs used is representative of the mix of real world computing problems. Clearly an increase in the size of the sample would be an improvement and an obvious bias is the lack of any really large programs.

There is a drawback of this general approach to the comparison of protection systems which should be mentioned here, though the topic is covered in greater detail in Chapter 7. A domain, in each implementation of Algol W, has been considered as a set of accessible areas of the

stack. Domains have been compared on the basis of the total amount of the stack (in terms of storage area) to which they permit access. This is a simplified view of the true nature of a domain but suffices for performing the gross comparisons which the methodology is aimed at.

Treating a domain as the sum of accessible areas of the stack totally ignores any structure which may exist within the domain and which may itself provide additional protection (c.f., Lampson (1974)). If array descriptors, for instance, are formed according to the specification given by Iliffe and Jodeit (1962) using codewords, then it is a simple matter for the hardware to perform checks that each subscript of an array element lies within the defined range of values for that subscript. The concept of a structured domain is discussed in Chapter 7.

Chapter 6

INTER-PROCESS PROTECTION IN A NAIVE OPERATING SYSTEM: A STUDY OF THE IBM SYSTEM/370 DOS/VS OPERATING SYSTEM

To demonstrate the utility of the proposed methodology in the comparison of simple protection systems providing inter-process protection, implemented as a policy of complete isolation between processes, an analysis of the IBM System/370 DOS/VS operating system (IBM 1973),(Birch 1973) was performed. This operating system was selected because it enabled the methodology to be applied to an environment in which inter-process protection is important and the statistics required for the analysis were readily available.*

In this chapter, the protection system used in DOS/VS is compared, by means of a cost-benefit analysis, with an alternative protection system which offers greater memory protection but at an increased cost. DOS/VS uses a fixed sized partition approach whereas protection in the alternative system is based on variable sized partitions, which can be fitted to the individual memory requirements of each process. Only memory protection is considered in detail, though later in the chapter there is some discussion of I/O devices and the possibility of extending the comparison to include the file system.

A rider must be added to the results obtained in this chapter concerning the validity of the data on which the analysis is based. It should be emphasised that the clock used to time the programs was relatively coarse (timings were available to the nearest 1/300th of a second), though this was compensated for in part by the relatively

---

*The generous assistance given by Procter & Gamble Ltd. in obtaining the required statistics from their computing system is gratefully acknowledged.

slow speed of the processor (IBM 370/135). More significant, only the total execution time of processes was available, so the amount of time spent in supervisor state by a process had to be estimated from the input/output counts for the process. Definitive conclusions concerning DOS/VS should not be drawn from the results of the analysis, rather the chapter should be viewed as a further example of the application of the methodology.

6.1 IBM System/370 DOS/VS Operating System

The disk operating system (DOS/VS) for the IBM System/370 computers is a medium scale general purpose multiprogramming operating system designed to provide background batch processing facilities in conjunction with up to four foreground processes. It is a naive operating system from the protection point of view. Programs are executed in fixed-sized memory partitions and inter-process protection is achieved by endeavouring to enforce complete isolation between user processes. The storage key protection feature prevents a program in one of the five partitions from writing into, reading from or directing an input/output operation into any of the other partitions or the supervisor area.

The unit of work a user submits to the system for processing is called a _job_. A job may consist of a number of _job-steps_, each job-step consists of one program which executes after the preceding job-step is completed. All the job-steps of a job are executed sequentially in the same partition. The execution of a job-step constitutes a process and forms the basic unit of protection.

A single virtual memory is maintained as an extension to real memory and its organisation is shown in Figure 6.1.

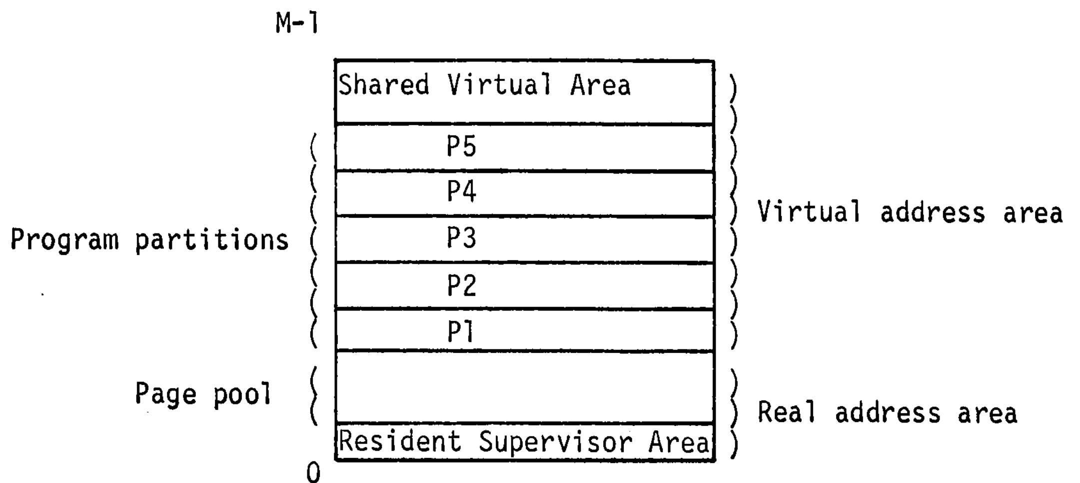


Figure 6.1

Memory organisation under DOS/VS

The combined real and virtual memory can be any size up to 2^{24} bytes (providing it is a multiple of 2K bytes) and is divided into 2K byte pages. Pages in virtual memory are brought into the page pool in real memory on demand by conventional demand paging techniques.

The real memory area contains a resident supervisor area and the page pool. Virtual memory contains a shared virtual area and between 1 and 5 partitions, the number being specified at the time of system generation. The size of each partition (a multiple of 2K bytes) is fixed at system generation time but can be modified by the operator. The number of partitions cannot be altered, but the amount of storage allocated to a partition can be set to zero, which in effect reduces the number of partitions. An assumption underlying the design of the operating system is that such changes to partitions

are relatively infrequent. Resident programs that can be shared between jobs and that are used frequently can be stored in the shared virtual area of the virtual memory.

The use of the various partitions for foreground or background jobs is immaterial here. A partition may be allocated space in the real address area. A program being executed in such a partition can elect to run in real mode and to have all its pages kept in main memory throughout its execution. Running in real mode means that the dynamic address translation facility is bypassed resulting in a significant improvement in efficiency. The use of certain peripherals with timing constraints, e.g., an optical character reader, or if a program is doing a substantial amount of input/output are also reasons for executing a program in real mode. In particular, the spooling program (POWER) supplied with the operating system runs in real mode. The usual mode of execution is virtual mode, in which case the real area allocated to the partition is viewed as part of the page pool and all addresses are converted by the dynamic address translation facility to real addresses.

When a program is to be executed, it is allocated to one of the five partitions, according to specifications contained on the job card. The whole partition is allocated to the program regardless of how much memory the program actually requires. In virtual mode the partition in virtual memory is allocated and in real mode the area of real memory assigned to the partition is allocated to the program. There is no restriction on a program accessing all of its allocated partition; the only constraint on a program is that all memory references generated by the executing program must remain within the assigned partition. The size of a partition cannot be changed while it contains an executing program.

The dynamic address translation mechanism ensures that all addresses used to access virtual memory fall within the size of the combined virtual and real memory (M, say) by means of a simple comparison of the page bits contained in the address with the size of the page table. In real mode, the hardware traps any address that is outside the range of the physical memory. Inter-process protection is achieved by means of hardware storage keys. Physical memory is divided into 2K byte blocks and associated with each block is a 4-bit storage key which can be set to a number between 0 and 15. These storage keys may be set and examined by appropriate privileged instructions. Each partition is assigned a different key between 1 and 15. When a page, from the virtual memory area of a partition, is transferred to a block in main memory the storage key of that block is set to the key assigned to the partition.

When a program is being executed by the processor, the program status word (PSW) of the executing program contains the same 4-bit key as that assigned to the partition in which the program is executing. The supervisor and programs in the shared virtual area execute with key zero. Before access is allowed to storage, a comparison is made by hardware between the key associated with the memory block, containing the real address issued by the program in the case of execution in real mode or the real address corresponding to the virtual address issued by the program in virtual mode, and the key in the PSW. The access is only allowed if the two keys are equal or if the PSW contains the 'master key' (key zero).

There are two versions of the storage key protection mechanism: fetch/store protection, which checks both fetch and store accesses to memory, and store protection which only checks stores. If a protection

violation does occur, a protection interrupt causes control to be regained by the supervisor.

The memory protection feature functions independently of the supervisor/problem state status of the machine and is always active. However, DOS/VS for simplicity identifies states of the machine with protection states. When the processor is in problem state executing a user program, the protection key in the PSW is that of the partition to which the program has been allocated. If the supervisor or code from the shared virtual area is being executed the processor is in supervisor state and executes with key zero. Key zero allows access to all memory, thus the supervisor and programs in the shared virtual area are 'trusted' to be well-behaved. In problem state, privileged instructions cannot be executed, so user programs are unable to change the storage keys or the protection key in the PSW.

References to memory can also be made by channels when transferring information to or from peripheral devices. Such transfers are also subject to the storage key protection mechanism. Channel programs which contain virtual addresses are first transformed by the supervisor into an equivalent channel program containing real addresses. A channel program contains one or more <u>channel address words</u>, each specifying an area of memory to or from which information is to be transferred. Each channel address word also contains a protection key and the hardware checks that this key matches the key(s) of the corresponding physical memory area before the transfer of information commences. In the case of unequal keys (key zero matches all storage keys) a protection interrupt occurs.

6.2 An Alternative Protection System

With DOS/VS, the sizes of the individual partitions can be arranged so that few, if any, programs need to use overlays, the method employed to overcome the constraints of small partitions in 360 DOS (IBM 1971). With larger partitions, it is to be expected that the majority of programs will use less than the whole of the allocated partition. The alternative protection system aims to limit a program during execution more closely to its known actual memory requirements than is achieved by DOS/VS. The same virtual storage organisation as used by DOS/VS is assumed.

To suit a program's memory needs, it is necessary to have information on its memory requirements during execution. There are two obvious sources for this information:

a. an estimate provided by the user

b. information from the loader and from requests for more space.

An estimate provided by the user can be expected to be greater than that actually required by a program and ignores the fact that programs, in general, have dynamic memory requirements. Recognising this behavioural aspect of programs can result in further improvement to the protection afforded to a program, though almost certainly at an increased cost. The alternative protection system is based, therefore, on information supplied by the loader and on requests from the executing program for more memory space.

Before execution, every user program is assumed to be loaded into the appropriate partition by the loader and thus the immediate memory requirements of the program can be supplied to the supervisor. If the program is to run in virtual mode, the supervisor can then indicate that any unused pages in the virtual area of the partition are not to be referenced, by means of an extra bit (the 'used' bit) on each page

entry in the page table. A reference to such a page will cause a protection violation interrupt. If a program requests more memory space (such requests can only be for a contiguous extension of a multiple of 2K bytes to the memory already allocated to the program), the appropriate 'unused' pages can be made available by resetting the relevant bits in the page table.

A program which is to run in real mode will be allocated space in the real memory allocation of the partition. The blocks in this allocation will have their storage keys set to that of the partition. Any unused blocks remaining in the real memory allocation of the partition will have their storage keys set to a key (e.g., 15) which indicates 'unused'. Additional memory can be allocated to the program by changing the storage key on the appropriate block(s) from 'unused' to that of the partition.

Execution then proceeds as under DOS/VS except that attempted memory accesses by the program in problem state to the unused space are treated as protection violations. A new state, the SVC state, is introduced to allow the supervisor to perform functions on behalf of a user program, but be restricted by the protection system to the supervisor area and the program area, in the case of real mode, and the supervisor area, the shared virtual area and the allocated program area in virtual mode. The third state, supervisor state, is reserved for the supervisor when it performs functions on behalf of the system, e.g., scheduling programs to be executed. In this state, the whole machine is accessible and the storage keys can be altered.

A possible mechanism to implement the alternative protection system is as follows. In problem state, the protection key of the partition, in which the currently executing job-step resides, is stored in the active program status word (PSW) and access is only permitted to blocks

of memory having the same key. On a switch to SVC-state, the protection key of the partition is still kept in the PSW, but access is now allowed to blocks with the same key or with key zero. By setting the storage keys of blocks containing supervisor code or shared virtual code to zero, access is restricted to the relevant portions of memory. In supervisor state, the master key (key 0) is stored in the PSW and access is allowed to the whole of memory.

6.3 A Cost-Benefit Analysis of DOS/VS and the Alternative Protection System

The first stage in performing a cost-benefit analysis is to identify the domains in which each process may execute and derive benefit measures for each of the protection systems. Cost measures must also be deduced for each protection system being considered. Then, given, in this case, a set of statistics of programs which have been run under DOS/VS, the measures can be evaluated and a comparison of the protection systems carried out.

6.3.1 Identification of Domains

In DOS/VS, two domains can be identified for each process: the problem state domain and the supervisor state domain. The problem state domain essentially consists of the partition allocated to the job-step and the machine instructions apart from the privileged instructions. In the supervisor state domain, the whole memory is accessible (either real or real + virtual depending on the mode) and all instructions can be executed.

A user process will normally execute in the problem state domain. From time to time, the process will need the services of the supervisor; usually requests for the initiation of input/output. Such requests

(SVC's) are accompanied by a switch from the problem state domain to the supervisor state domain. Programs residing in the shared virtual area and the supervisor process, which performs scheduling functions, resource allocation, etc., always execute in supervisor state.

Under the alternative protection system, the domains in which user processes execute are very similar to those implemented by DOS/VS. The problem state domain corresponds to the problem state domain in DOS/VS, except that a process is restricted to the portion of the partition which it actually requires. Supervisor state domain in DOS/VS corresponds to the SVC state domain in the alternative protection system. Under the alternative protection system, only the supervisor process executes in the supervisor state domain.

The cost-benefit analysis of the two protection systems is performed on the basis of the amount of accessible memory in each domain, so other aspects such as available instruction sets, domain switches, etc., in each domain are ignored.

6.3.2 Benefit and Cost Measures

In the development of these measures we assume that fetch and store protection is in operation, i.e., all accesses to memory are checked by the protection system, and concentrate solely on memory protection. Access to memory by input/output devices via channels is ignored for the moment and it is assumed that there is no inter-partition communication.

Figure 6.1 showed the organisation of memory under both protection systems. Let the following quantities be represented as follows:

size of the resident supervisor area	s bytes
size of the shared virtual area	h bytes
size of physical memory	R bytes
size of real + virtual memory	M bytes

M,R,s and h will all be integral multiples of 2K bytes (the page size), so the supervisor and the shared virtual area may be allocated memory areas slightly larger than they strictly require. The possibility of making a correction for this is discussed later.

Let all job-steps executed on the processor over a given test period be indexed by the positive integers, and the total number of job-steps be N. Job-step $j (1 \leq j \leq N)$ during its execution will occupy a certain amount of the partition (either in real or virtual memory depending on the mode) to which it is allocated, say q_j bytes. If the job-step has dynamic storage requirements, q_j is the maximum memory requirement of the job-step within the partition. In general, q_j will only be known after the job-step has been executed.

Since storage is allocated in 2K byte blocks, the minimum storage that can be allocated to job-step j

$$= \left\lceil \frac{q_j}{2048} \right\rceil \cdot 2048 = \underline{r_j p_j} \qquad 0 < r_j \leq 1$$

where p_j = size of the real or virtual allocation of the partition in which job-step j executes. Hence, r_j , the portion of the partition used by job-step j, can be determined. This is illustrated in Figure 6.2.

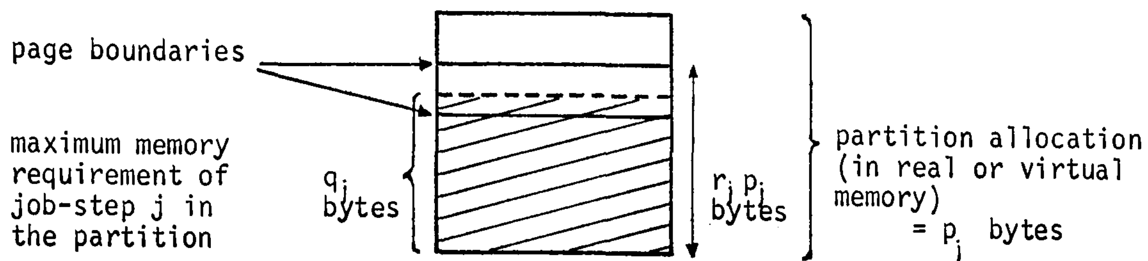


Figure 6.2

Job-step j executing in a partition

In the problem state domain of job-step j:

	real mode	virtual mode
amount of memory accessible under the alternative protection system	$r_j p_j$	$r_j p_j$
amount of memory accessible under DOS/VS	p_j	p_j

In the SVC-state domain or supervisor state domain:

	real mode	virtual mode
amount of memory accessible under the alternative protection system	$s+r_j p_j$	$s+h+r_j p_j$
amount of memory accessible under DOS/VS	R	M

The amount of accessible memory under the alternative protection system in the problem state domain and SVC-state domain represents that amount of memory to which access is required, taking account of all known information about the job-steps. In a sense, this is a tighter upper bound on the memory requirements of job-steps than that used by DOS/VS, which could only be improved upon if further information about a process's intentions had been provided. From the

point of view of the model developed in Chapter 4, the alternative protection system represents the 'ideal' system, making use of all available knowledge. The amounts of accessible memory for the alternative protection system represent the sets to which access is required, i.e., the theoretically accessible set, in the model's terms.

If we ignore the supervisor process, the benefit measure under the above assumptions for DOS/VS is:

$$\frac{1}{N}\sum_{j=1}^{N}(\text{benefit measure for job-step } j).$$

We choose to weight each domain by the time spent by a process in that domain, using time as a measure of the number of references made within a domain. So the benefit measure for job-step j

$$= \frac{1}{\text{total execution time of job-step } j} \cdot \sum_{\text{domains } d}\left(\frac{\text{amount of memory to which access is required in domain } d}{\text{amount of accessible memory in domain } d} \cdot \text{time spent in domain } d\right)$$

Hence for N job-steps the benefit measure for DOS/VS

$$= \left(\sum_{j=1}^{N}\underline{(r_j\, t_{jp}+\alpha_j\, t_{js}) \cdot \frac{1}{(t_{jp}+t_{js})}}\right)\cdot\frac{1}{N} \qquad \text{(B1)}$$

where $\alpha_j = \begin{cases} \dfrac{s+r_j\, p_j}{R} & \text{for job-steps running in real mode} \\[2ex] \dfrac{s+h+r_j\, p_j}{M} & \text{for job-steps running in virtual mode} \end{cases}$

t_{jp} = problem state time for job-step j

t_{js} = supervisor state time for job-step j

In this measure, we are assuming that the time spent in a domain is a good indicator of the number of memory references made in a domain. This is a reasonable assumption for the sample set of job-steps used in this analysis since all the job-steps used a wide mix of the machine instructions and the execution time is of the order of seconds for almost all the job-steps.

When a 370 computer runs in virtual mode, the effective instruction rate is lower than that obtained in real mode due to the overhead of dynamic address translation. In the benefit measure, only the proportions of times spent in problem state and supervisor state are significant and so it is unnecessary to make a correction for job-steps running in virtual mode.

The corresponding benefit measure for the alternative protection system is 1, since in problem state domain and in SVC-state domain the amount of accessible memory is precisely that which is required to be accessible.

The above derivation of the benefit measure B1 was based on the maximum memory requirement of a job-step within a partition, namely q_j. The alternative protection system in fact allows job-steps to have dynamic memory requirements, more memory can be requested though not later released. Therefore, the memory requirements of a job-step can be represented by a monotonic increasing function of time, $q_j(t)$, though its value only changes a finite number of times and by a discrete amount each time.

If changes to the memory requirements of job-step j occur at times $t_{j0}, t_{j1}, \ldots, t_{je+1}$ (where any convenient measure is used for 'time',

e.g., processor time), and t_{j0} represents the start of program execution and t_{je+1} the end, and the corresponding memory requirements are $q_{j0}, q_{j1}, \ldots, q_{je}$ (measured after the memory change has taken place), then the benefit measure is

$$\left(\sum_{j=1}^{N} \left(\left(\sum_{i=0}^{e} r_{ji} t_{jpi} \right) + \left(\sum_{i=0}^{e} \alpha_{ji} t_{jsi} \right) \right) \cdot \frac{1}{\sum_{j=0}^{e} \left(t_{jpi} + t_{jsi} \right)} \right) \cdot \frac{1}{N} \qquad \text{(B1')}$$

where $$r_{ji} = \left\lceil \frac{q_{ji}}{2048} \right\rceil \cdot 2048$$

$$\alpha_{ji} = \begin{cases} \dfrac{s + r_{ji} p_j}{R} & \text{for job-steps running in real mode} \\[2ex] \dfrac{s + h + r_{ji} p_j}{M} & \text{for job-steps running in virtual mode} \end{cases}$$

and t_{jpi} and t_{jsi} are respectively the problem state and supervisor state times of execution between t_{ji} and t_{ji+1} .

If the benefit measures B1 and B1' are compared:

$$\sum_{i=0}^{e} t_{jpi} = t_{jp} \qquad \sum_{i=0}^{e} t_{jsi} = t_{js}$$

$$q_{ji} \leq q_j \quad \Rightarrow \quad r_{ji} \leq r_j \qquad i=0,1,\ldots,e$$

hence B1' $\leq$ B1. Thus B1 is an upper bound for the protection afforded by DOS/VS.

Statistics at the level of detail necessary to calculate B1' were not available, so this measure has not been evaluated. Instead, the measure B1 has been used since it represents an upper bound for B1'. In the case of the 370/135 system being considered, few of the job-steps

had dynamic memory requirements and thus B1 provides a good estimate for the measure B1'.

To complete the comparison, it is also necessary to compare the relative costs of the two protection systems. We consider first the DOS/VS scheme for utilising storage keys under the headings suggested in Chapter 4.

There are three types of cost: hardware, storage space and processing time. The provision of the hardware to maintain the storage keys associated with each block and to check the protection key in the PSW (or channel address word in the case of I/O transfers) against the appropriate storage key on every memory reference will result in a significant once only cost. Since the protection checking is performed by hardware, software is only required to maintain the storage keys on the blocks of main memory. The space taken up by this extra code and any associated tables will be relatively small and is ignored in this analysis. Of more significance is likely to be the processing time required to maintain the protection system, but in the case of DOS/VS this is largely dwarfed by the paging overhead.

Domain creation

The partitions, the shared virtual area and the supervisor area are established at system initialisation time. (In the case of the actual system being considered this was performed twice a day every day; once in the morning to establish the configuration for day-time use and once in the evening when the remote job entry lines were disconnected.) The significant actions are setting up the areas of real memory and creating the page table. This cost is relatively insignificant compared to the total cost of initialisation. When a

program is loaded into a partition prior to execution, either the appropriate part of the page table will have to be set up for execution in virtual mode or storage keys set on blocks in main memory for execution in real mode. Such costs are small compared to the loading overhead.

Domain deletion

When a process terminates execution, it is only necessary to reset the relevant entries in the page table and to change the storage keys on any pages which belonged to the process and were mapped into real memory, thus preventing another process executing in the same partition from inadvertently accessing those pages.

Domain maintenance

In real mode, the memory area allocated to a user process resides in the physical memory and no maintenance is necessary. In virtual mode, when a page is brought into real memory it is necessary to set the storage key on the block into which the page is loaded. The few instructions necessary to do this make little impact on the normal paging overhead.

Domain switches

User processes can switch between problem state domain and supervisor state domain by issuing a supervisor call (SVC). The appropriate changes to the protection key in the PSW take place automatically as part of the SVC instruction, so the cost of such a domain switch is minimal. The same is true of a domain switch from supervisor state domain to the problem state domain.

Domain changes

Changes to partition sizes do not occur during execution of a job-step, so under DOS/VS no domain changes take place. In the actual system under consideration, the partition sizes were only changed when the system was reinitialised.

Protection checking

The actual protection checking on references to memory by the processor and channels is carried out by hardware and thus incurs the one-time hardware cost.

It would be exceedingly difficult to deduce the actual total absolute cost of the DOS/VS protection system. However, since two protection systems are being compared which use essentially the same hardware mechanism, it is more convenient, for the purposes of comparison, to consider the extra cost of the alternative protection system over that of DOS/VS rather than the absolute cost of each system. It is only necessary, therefore, to identify the differences between the two systems and quantify the increased cost.

As with DOS/VS, there will be a once only hardware cost for the provision of the storage key mechanism for the alternative protection system. The mechanism is largely the same in both cases, though the alternative scheme requires hardware support for the SVC state. It is likely that the increased cost would be small.

Domain creation

The initial setting up of the system will be little different from the initialisation of DOS/VS except that the keys on the super-

visor area memory blocks are set to zero. On loading a program to be executed, the portion of the partition not required by the job-step is flagged as 'unused'. In virtual mode, use is made of an extra bit on every entry in the page table to indicate whether each page in a partition is 'used' or 'unused'.

Domain deletion

As with DOS/VS, when a process terminates execution the storage keys on any blocks containing pages which belonged to the process should be set to 'unused'. If the process executed in virtual mode, the relevant page entries in the page table should be reset.

Domain maintenance

The maintenance of domains under the alternative protection system is very similar to that performed by DOS/VS except that when a page from the shared virtual area is loaded into real memory, the storage key on the memory block into which the page is loaded must be set to zero.

Domain switches

The number of switches between domains is the same as occurs under DOS/VS except for the switches caused by requests for more memory space.

Domain changes

Under the alternative protection system the portion of a partition allocated to a process can be increased if a request is made by the process. The servicing of such requests will involve a switch into the supervisor state domain to enable the allocation of more space

in either real memory, by changing 'unused' storage keys to 'used', or virtual memory, by setting the appropriate bits in the page table. It is assumed that requests for more memory occur very infrequently relative to the number of instructions executed between such requests.

Protection checking

From the point of view of protection checking, problem state and supervisor state are the same as in DOS/VS. The introduction of the SVC state will involve some modification to the checking algorithm. The provision of a state indicator in the processor to represent SVC state and the necessary alterations to the checking algorithm are likely to add a small incremental amount to the once only hardware cost.

To summarise the increased cost of the alternative system, there is a small additional hardware cost, a small extra storage cost, resulting from the new bit on every page table entry and the code to support the new features of the alternative protection system, and increased processing time. The extra hardware and storage space costs are ignored, hence cost in this analysis is equated to extra processor time required by the alternative protection system.

The extra processor time is made up of repetitions of the following primitive actions:

1. domain creation in real mode -

   setting the storage key on a used memory block

2. domain deletion in real mode -

   setting the storage key on a previously used memory block to 'unused'

3. domain creation in virtual mode -

   setting the 'used' bit in the page table entry of a used virtual page

4. domain deletion in virtual mode -

   resetting the 'used' bit in the page table entry of a previously used virtual page

5. protection checking in virtual mode -

   on a page fault, checking that the 'used' bit in the page table entry for the required page is set

6. domain change in real mode -

   setting the storage key on a memory block to 'used'

7. domain change in virtual mode -

   setting the 'used' bit in the page table entry of the allocated page

8. servicing of request for more memory.

If the processing time for each of these primitive actions is $c_1, c_2, \ldots, c_8$ respectively, then an estimate for the extra time necessary to complete the sample of job-steps is

$$\sum_{i=1}^{8} c_i n_i \qquad \text{(C1)}$$

where $n_1, n_2, \ldots, n_8$ are the total number of times each respective primitive action is invoked during execution of the job-steps. This measure for the extra cost is only an estimate because in many cases storage keys will not need to be changed and thus there are opportunities for optimising the actions of the alternative protection system.

6.3.3 Evaluation of the Measures

The system from which statistics were obtained to compare DOS/VS and the alternative protection system was an IBM 370/135 with 192K bytes of main memory and associated peripherals. Figure 6.3 shows

the essential details of the configuration.

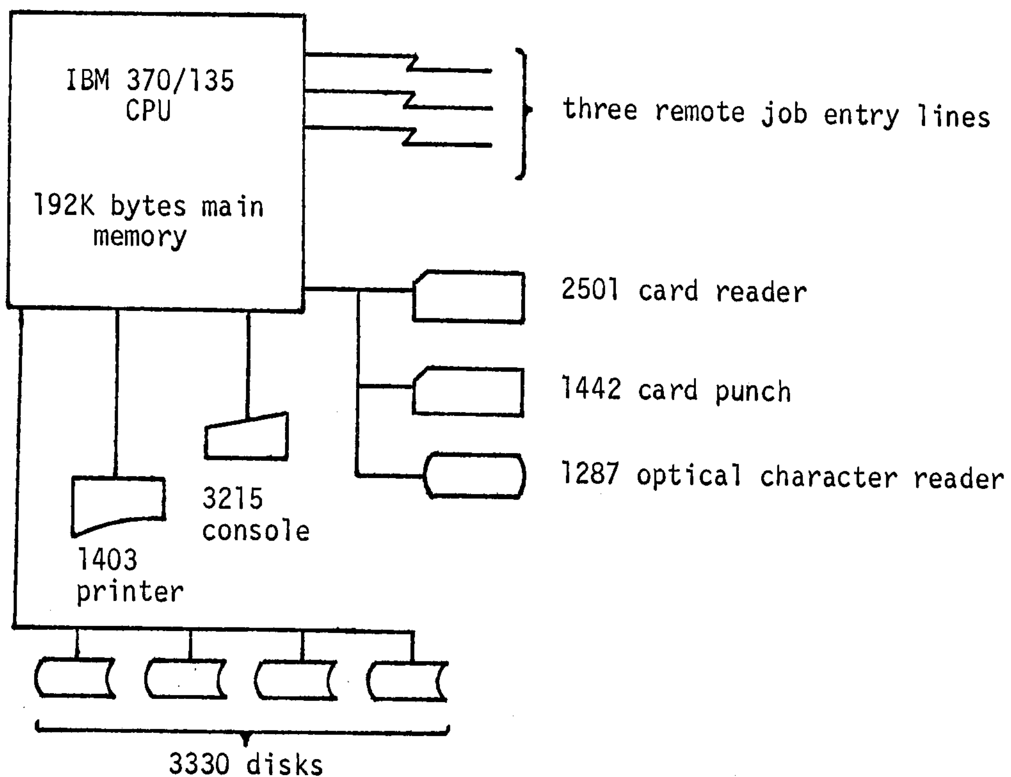


Figure 6.3

IBM 370/135 configuration

The system is run for two 8 hour shifts on each weekday, using DOS/VS release 30 as the operating system. Three partitions are maintained: one background (BG) and two foreground partitions (F1 and F2). The POWER spooling system (supplied by IBM with DOS/VS) always occupies the F1 partition. The background partition is used for batch production work and the remaining foreground partition, F2, for testing and system development during the day shift and for batch work during the night shift. During the night shift, the remote job entry lines are not supported.

The size of the F1 partition is changed at the start of each shift by re-initialising the system, though the number of partitions remains fixed. The organisation of real and virtual memory during both shifts is shown in Figure 6.4.

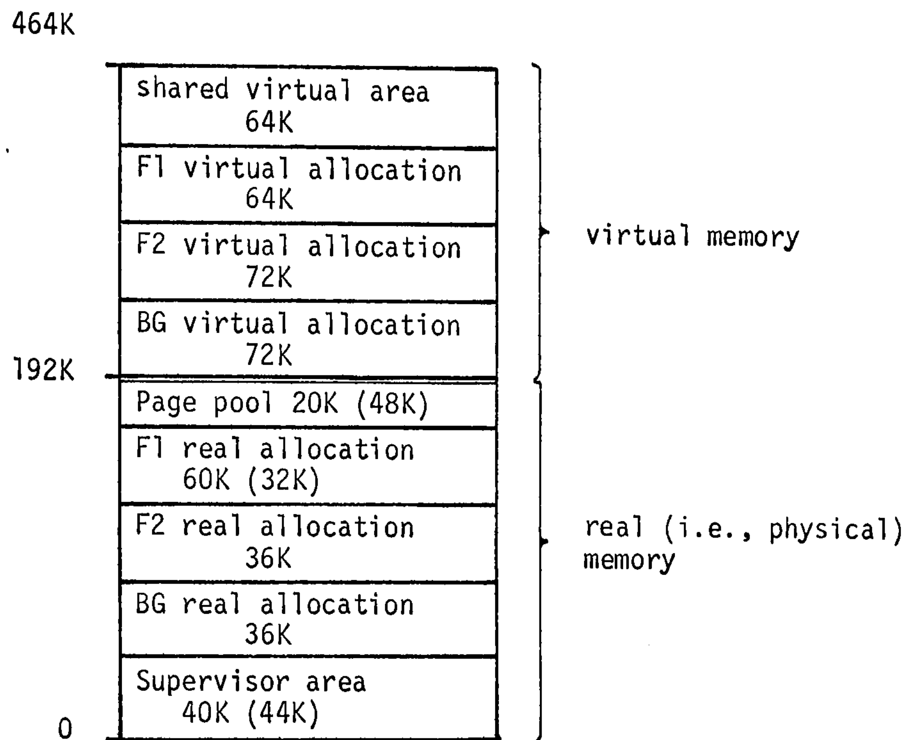


Figure 6.4

Organisation of real and virtual memory during the day-time shift

(Where the size of an area changes for the night-time shift, the night-time value is shown in brackets.)

Space in real memory is allocated to each of the three partitions. In the case of the F1 partition this is always occupied by the POWER program since it runs in real mode.

DOS/VS requires that every partition be allocated a minimum 64K bytes of virtual memory even though it may never be used. So the F1 partition has a 64K virtual memory allocation even though it remains

unused. For similar reasons, the shared virtual area occupies 64K of virtual memory, when in fact only about 10K is actually used by the system under consideration.

Statistics were gathered on every job-step executed on the system in the space of one week. DOS/VS provides a facility whereby job-step statistics recorded by the operating system during the execution of the job-step are available to a user-supplied accounting routine at the termination of each job-step. These statistics were simply written to a disk-file during the two shifts each day, and at the start of the next day the file was dumped to cards for later analysis.

The numbers of job-steps run each day (i.e., the cumulative totals for both shifts) in each partition, broken down by mode of execution, are detailed in Table 6.1.

The majority of job-steps (92 - 94%) ran in virtual mode in partitions BG and F2. In assessing the results of evaluating the measures, therefore, greater emphasis will be placed on the BG and F2 virtual mode job-steps, rather than the job-steps run in real mode.

	Monday	Tuesday	Wednesday	Thursday	Friday
BG - real	18	8	7	6	5
BG - virtual	220	257	215	270	256
F1 - real	2	2	2	2	2
F1 - virtual	0	0	0	0	0
F2 - real	22	38	35	29	37
F2 - virtual	342	431	348	311	321
Total no. of job-steps	604	736	607	618	621

Table 6.1

Numbers of job-steps run each day in each partition, broken down by mode of execution

The relevant statistics gathered on each job-step included:

- identification of job-step
- date and time of start of execution
- partition in which job-step was executed
- mode of execution (real or virtual)
- execution time of job-step
- address of block or page containing the largest address referenced by the job-step in problem state
- input/output counts for each peripheral device.

Only the total execution time of each job-step was recorded by DOS/VS, separate values for the times spent in supervisor state and problem state, as required for the evaluation of the benefit measure B1 were not available. In theory, it would be possible to obtain the separate problem state time and supervisor state time for each job-step, either by making alterations to the operating system or by use of a hardware monitor. In practice, there was no possibility of being able to make (the quite significant) changes to the operating system or to use a hardware monitor, since the system operated in a commercial environment. However, by making the assumption that almost all supervisor state time of a job-step is attributable to requests for input/output, it was possible to make an estimate for the supervisor state time of each job-step based on the I/O counts of the job step, since these were available. The precise details of how this was done are described in the Appendix.

This situation has the advantage that it illustrates the use of the methodology in a context where one is working with partial

information, not untypical of the real world, and is representative of a design situation where one is using statistics from an existing system to predict the performance of a system which is partially designed and certainly not implemented. If, in fact, an existing system was being studied with a view to making improvements, e.g., IBM/370 with DOS/VS, the cost of acquiring the supervisor state time for each job-step could undoubtedly be justified.

The job-steps in the sample fall into two broad categories: production runs of compiled programs and the compilation and testing of new programs. The majority of programs are written in COBOL or PL/1, with some assembler and other languages. The work performed by the installation is typical of medium sized commercial systems, namely file maintenance, stock control and payroll etc. Many of the production runs are done on a specific day each week, thus it was felt that by taking one week's work done on the system, a reasonable sample of all programs run on the system would be obtained. In total, statistics were obtained on 3186 job-steps.

The first measure to be evaluated was the benefit measure B1. The values obtained for the various partitions and modes of execution are given in Table 6.2.

The most significant values of the measure occur in the cases of BG-virtual and F2-virtual, on average 0.58 and 0.49 respectively, since the majority of job-steps are executed in these partitions in virtual mode. The difference in these two values probably arises because of the different nature of the job-steps run in the two partitions. The BG partition is used for batch production work and so the job-steps executed there consist primarily of compiled and link-edited object modules. F2 is used for compilations and test runs.

	Monday	Tuesday	Wednesday	Thursday	Friday	Average
BG - real	.35	.61	.27	.38	.38	.40
BG - virtual	.57	.57	.57	.57	.61	.58
BG - real + virtual	.55	.57	.56	.57	.61	.57
F1 - real	.94	.94	.94	.94	.94	.94
F2 - real	.32	.32	.33	.33	.34	.33
F2 - virtual	.49	.49	.49	.50	.49	.49
F2 - real + virtual	.48	.48	.48	.48	.47	.48
BG + F2	.51	.51	.51	.52	.53	.52
BG + F1 + F2	.51	.51	.51	.52	.53	.52

Table 6.2

Evaluation of benefit measure B1 by partition and mode of execution

From Table 6.2, it is evident that apart from BG-real, there is very close agreement for the value of the measure for each partition and mode of execution on each day of the week. The large fluctuations occurring in the case of BG-real can be accounted for by the very few job-steps executed each day in that partition and mode (18,8,7,6,5 job-steps on each day of the week respectively). The very large value of the measure occurring in the case of the F1 partition is due to the fact that it is only used in real mode by two job-steps each day, and the real memory allocation of F1 is fitted to the requirements of these job-steps.

The contribution to the benefit measure by individual job-steps varied widely as was to be expected since a large number of programs

were run each day all with greatly varying memory requirements. However, the very small variation in the values of the benefit measure for each day of the week shows that the value of the benefit measure averaged over a large number of job-steps is significant.

Assume that a typical job-step requires 40K bytes of the virtual memory allocation of BG or F2 for its execution and that the ratio of problem state time to supervisor state time is 10:1. For this single job-step, B1 evaluates to .53. Consider now the improvement that is gained in the benefit measure by restricting the job-step to 40K bytes in the problem state domain, rather than the 72K bytes of the virtual memory allocation to BG or F2, as happens under DOS/VS. The value of B1 with a 40K byte allocation is 0.94. Thus a dramatic improvement in protection can be obtained by improving the protection afforded to the job-step in the problem state domain. This arises because programs spend relatively little time in supervisor state. (The ratio 10:1 of problem state time to supervisor state time is typical of the sample set of job-steps.)

Though the time spent in the supervisor state domain is small, the effect of an undetected error could be devastating since the programs executing in the other partitions, or indeed the supervisor itself, could be corrupted. In the problem state domain under DOS/VS, a process can write into the unused portion of the partition in which the process executes. A process which does this is in error, but has no direct effect on other processes, and thus this can be viewed as not so serious. Therefore, it may be desirable to apply weights to the various domains involved in the measure B1, to reflect to some extent the relative consequences of an undetected error occurring in each of the domains.

The average value of the measure B1, namely .52, demonstrates that the typical job-step has access to approximately twice as much memory as it requires to perform its function. The alternative protection system therefore represents a significant improvement in protection over that afforded by DOS/VS.

The extra cost of the alternative protection system in terms of increased processor time can be obtained by evaluating the measure C1 derived in Section 6.3.2. The number of times each primitive is invoked is as follows (n_i is the number of times primitive i is invoked):

n_1, n_2 : total number of memory blocks used by job-steps executed in real mode

n_3, n_4 : total number of pages used by job-steps executed in virtual mode

n_5 : total number of page faults (which is proportional to the total execution time of all job-steps executed in virtual mode)

n_6 : total number of memory blocks allocated to job-steps executing in real mode as a result of requests for extra storage

n_7 : total number of pages allocated to job-steps executing in virtual mode as a result of requests for extra storage

n_8 : total number of requests for extra storage issued by all job-steps

The totals n_1, n_2, n_3 and n_4 were calculated for the sample set of job-steps. For n_5, the total execution time of all job-steps executed in virtual mode was calculated and multiplying this by the

average page fault rate γ will yield n_5. These results are given in Table 6.3 for one day (Tuesday). Similar results were obtained for the other days of the week. Since statistics were not gathered on requests for extra storage, n_6, n_7 and n_8 were not determined.

partition	n_1, n_2	n_3, n_4	n_5
BG	96	5405	3168325γ
F1	46	0	0
F2	230	7692	257247γ

Table 6.3

Statistics from Tuesday's data

(The units of time used in determining n_5 are 1/300th of a second)

It would be possible to determine the costs $c_1, c_2, \ldots, c_8$ of one invocation of each primitive by coding each primitive and calculating the execution times of the primitives from the individual instruction execution times. If statistics were available on requests for more memory, it would then be possible to evaluate the measure C1. This has not been done because it is evident that the execution time of each primitive is very small, and the number of times each primitive is invoked is such that the total increase in execution time under the alternative protection system is likely to be almost insignificant compared to the total execution time of the sample set of job-steps.

The alternative protection system is seen therefore to offer greatly improved protection at little extra cost.

6.3.4 Evaluation of Other Benefit Measures

Measure B1 was derived on the assumption that all references to memory (both reads and writes) are checked by the protection system. It is instructive to derive a form of the benefit measure in the case where only writes to memory are checked. If we assume that the ratio of writes to all references is a constant w for all job-steps, this benefit measure, B2, is:

$$\left(\sum_{j=1}^{N} \left((wr_j + (1-w)\beta_j) t_{jp} + \alpha_j t_{js} \right) \cdot \frac{1}{(t_{jp}+t_{js})} \right) \cdot {}^{1}/_{N} \qquad \text{(B2)}$$

where
$$\alpha_j = \begin{cases} \dfrac{s+r_j p_j}{R} \\[2ex] \dfrac{s+h+r_j p_j}{M} \end{cases} \quad \beta_j = \begin{cases} \dfrac{r_j p_j}{R} & \text{for job-steps running in real mode} \\[2ex] \dfrac{r_j p_j}{M} & \text{for job-steps running in virtual mode} \end{cases}$$

Essentially, each job-step executes in four domains: two problem state domains, one for reads and one for writes, and two supervisor state domains. As the whole of memory (real or virtual depending on mode of execution) is accessible in supervisor state, the two supervisor state domains are identical regardless of the type of access to memory.

The quantitative improvement in protection benefit afforded by read protection can be directly related to a financial cost, namely the extra cost of having the computer supplied with read protection.

To evaluate B2, it is necessary to have a value for the constant w. w is assumed constant since almost all the sample job-steps have an execution time of the order of seconds and thus involve the execution of many 100,000's of instructions and include a wide mix of the

available instructions. Studies of Algol programs (Wichmann 1970), (Wyeth 1973) have shown that for references to operands the proportion of writes to all references is approximately .23. Burnell and Coffman (1975) quote a figure of 80% for the percentage of instructions requiring an operand (data) reference to memory. From these figures, it is possible to deduce that the proportion of writes to all references lies in the range .1 to .2. The measure B2 was evaluated for w = .15 for each day of the week in which statistics were gathered.

Table 6.4 gives the values of the measure B2 (w = .15) obtained for each partition and mode of execution. In the case of BG-virtual, on average B2 is .18 compared with the corresponding average value of B1, .58. A similar difference in values occurs with F2-virtual, namely .15 and .49. This large difference in the values of the two measures is to be expected, since in the case of B2, only 15% of memory references are being checked for possible protection violations. Thus, a very significant improvement in protection can be obtained by the implementation of read protection as well as write protection.

From the point of view of the partitions, other than the one containing the currently executing job-step, the supervisor and the shared virtual area, a write protection violation is probably much more serious than a read violation. A write violation will almost certainly cause the eventual failure of the program whose code or data has been overwritten, whereas a read violation will at worst lead to a breach of confidentiality, though this itself may be very damaging in certain circumstances. To the executing job-step, a read violation and a write violation are equally serious, since they indicate that the program is in error.

	Monday	Tuesday	Wednesday	Thursday	Friday	Average
BG-real	.15(.35)	.23(.61)	.14(.27)	.16(.38)	.17(.38)	.17(.40)
BG-virtual	.18(.57)	.18(.57)	.18(.57)	.18(.57)	.19(.61)	.18(.58)
BG-real + virtual	.18(.55)	.18(.57)	.18(.56)	.18(.57)	.19(.61)	.18(.57)
F1-real	.37(.94)	.37(.94)	.37(.94)	.37(.94)	.37(.94)	.37(.94)
F2-real	.15(.32)	.14(.32)	.15(.33)	.14(.33)	.15(.34)	.15(.33)
F2-virtual	.15(.49)	.16(.49)	.15(.49)	.16(.50)	.15(.49)	.15(.49)
F2-real + virtual	.15(.48)	.15(.48)	.15(.48)	.16(.48)	.15(.47)	.15(.48)
BG+F2	.16(.51)	.16(.51)	.16(.51)	.17(.52)	.17(.53)	.16(.52)
BG+F1+F2	.16(.51)	.16(.51)	.16(.51)	.17(.52)	.17(.53)	.16(.52)

Table 6.4

Evaluation of benefit measure B2 by partition and mode of execution (w=.15) Corresponding values of B1 are given in brackets

If the computer system is operated in a situation where confidentiality of information is not an overriding consideration, so that write errors are the main source of concern with respect to inter-process protection, it may be appropriate to give greater weight to the domains for writes than those for reads in the measure B2. It might be assumed that if reads are not checked, a read violation will eventually lead to a write violation or some other error which will be detected by the hardware. This is probably largely true, but the point of detection of the error is then that much further from its source than would have been the case if read protection had been employed.

It is necessary therefore to consider the environment of the computer system and then to derive an appropriate form for the measure B2 which reflects the functional requirements of the protection system. The evaluation of B2 and knowledge of the cost of installation of read protection would then provide a quantitative basis on which to make a decision.

It is possible to extend the analysis of the protection benefit to include input/output operations carried out on behalf of a user job-step by the channels. The channel address word (CAW) in a channel program contains the (real) address of the start of the physical memory area from which or into which data is to be transmitted by the channel. The CAW also contains the protection key of the partition in which the job-step resides, and, during the transmission of data, the hardware protection system checks that the key in the CAW matches the storage key(s) of the memory block(s) which are accessed. When transfers are done on behalf of the supervisor, the CAW contains the master key zero, so no checking is performed. Input/output transfers done on behalf of user job-steps are thus 'executed' in the problem state domain. If the total duration time of input/output transfers for each job-step is available, the measures B1 and B2 can be adjusted to include input/output operations by increasing the time spent in the problem state domain by this amount. For B2, the time of input/output transfers will have to be split into input transfer time and output transfer time.

Yet another possible application of this analysis would be to compare the protection achieved by DOS/VS and the alternative protection system with the protection which could, in theory at least, be attained if job-steps could be restricted to the maximum amount of storage they require rather than having to round up all memory

requirements to a whole number of pages or blocks. This analysis would essentially be examining the effect of internal fragmentation (Randell 1969) which is a consequence of memory being allocated in units of 2K bytes.

Using the notation of Section 6.3.2, job-step j actually requires a maximum q_j bytes. Let the actual supervisor requirement be s' bytes and that of the shared virtual area be h' bytes. Assuming that both reads and writes are checked by all protection systems and ignoring access to memory by input/output devices, the benefit measure for DOS/VS is:

$$\left(\sum_{j=1}^{N} \left(\frac{q_j t_{jp}}{p_j} + \alpha'_j \; t_{js} \right) \cdot \frac{1}{(t_{jp} + t_{js})} \right) \cdot \frac{1}{N} \qquad \text{(B3)}$$

and the corresponding measure for the alternative protection system

$$\left(\sum_{j=1}^{N} \left(\frac{q_j t_{jp}}{r_j p_j} + \gamma'_j \; t_{js} \right) \cdot \frac{1}{(t_{jp} + t_{js})} \right) \cdot \frac{1}{N} \qquad \text{(B4)}$$

where
$$\alpha'_j = \begin{cases} \dfrac{s'+q_j}{R} & \\[2ex] \dfrac{s'+h'+q_j}{M} & \end{cases} \qquad \gamma'_j = \begin{cases} \dfrac{s'+q_j}{s+r_j p_j} & \text{for job-steps running in real mode} \\[2ex] \dfrac{s'+h'+q_j}{s+h+r_j p_j} & \text{for job-steps running in virtual mode} \end{cases}$$

The utility of these measures would be that by evaluating the measures and comparing B3 and B4 with B1 and 1 respectively, they would show the effect of internal fragmentation. The exact maximum memory requirement of each job-step was not recorded by DOS/VS and so these measures could not be evaluated. Internal fragmentation is a function of the page size, and so B3 and B4 would be expected to

vary if the page size was altered.

A further productive application of the benefit measure B1 is to use the measure as a figure of merit to assess the effect of proposed minor changes to a system from the point of view of protection. The first such change considered was that of making use of the knowledge that only 10K bytes of the shared virtual area are actually used. To investigate the relaxation of the DOS/VS constraint that the minimum allocatable area in virtual memory is 64K, the benefit measure B1 was evaluated for two memory organisations. The first in which the shared virtual area was 10K bytes and the total memory size (real + virtual) was 464K bytes, and the second in which the total memory size was 410K bytes, representing a reduction in the size of the shared virtual area from 64K to 10K bytes. Table 6.5 shows the results of evaluating B1 for these two memory organisations.

	shared virtual area = 10K total memory size = 464K	shared virtual area = 10K total memory size = 410K
BG-real	.611	.611
BG-virtual	.556	.559
BG-real+virtual	.558	.560
F1-real	.938	.938
F2-real	.324	.324
F2-virtual	.480	.482
F2-real+virtual	.467	.469
BG+F2	.500	.502
BG+F1+F2	.501	.503

Table 6.5

Evaluation of B1 for a shared virtual area of 10K using one day's data (Tuesday)

The values of B1 have been given to three decimal places so that the slight improvement obtained by reducing the total memory size can be seen. The values of B1 for the real mode of execution do not alter since the organisation of real memory is the same for both cases. The values of B1 for the first memory organisation (shared virtual area 10K bytes, total memory size 464K) are slightly lower than those obtained for a shared virtual area of 64K bytes and total memory size 464K bytes (Table 6.2). This is due to the fact that the two organisations are quite different, since in Section 6.3.3 it was assumed that all 64K bytes of the shared virtual area were used, and thus the benefit measures for the two situations cannot be compared directly. The reason that the increase in B1, as shown in Table 6.4, is so slight is that only the term in B1 representing the supervisor state domain is altered by this change and most job-steps spend a low proportion of their time in supervisor state (typically 0-20%).

A second obvious change to the memory organisation is the removal of the virtual allocation to the F1 partition. The values of B1 for the initial memory organisation (shared virtual area = 64K and total memory size = 464K) and with the F1 virtual allocation removed (shared virtual area = 64K and total memory size = 400K) are given in Table 6.6.

As in the previous case, the measure shows only a slight improvement which again reflects the small proportion of execution time spent in supervisor state by most job-steps.

The combined effect of reducing the shared virtual area to 10K and removing the F1 virtual allocation is shown in Table 6.7.

	total memory size 464K	total memory size 400K
BG - real	.611	.611
BG - virtual	.568	.573
BG - real+virtual	.569	.574
F1 - real	.938	.938
F2 - real	.324	.324
F2 - virtual	.490	.494
F2 - real+virtual	.477	.480
BG+F2	.510	.514
BG+F1+F2	.511	.515

Table 6.6

Evaluation of B1 for total memory sizes 464K and 400K, reflecting removal of the F1 virtual allocation (shared virtual allocation = 64K) using one day's data (Tuesday).

	shared virtual area = 10K total memory size = 464K	shared virtual area = 10K total memory size = 346K
BG-real	.611	.611
BG-virtual	.556	.562
BG-real+virtual	.558	.564
F1-real	.938	.938
F2-real	.324	.324
F2-virtual	.480	.485
F2-real+virtual	.467	.472
BG+F2	.500	.505
BG+F1+F2	.501	.506

Table 6.7

Evaluation of B1 showing the combined effect of reducing the shared virtual area to 10K and removing the F1 virtual allocation

Mention was made in the introduction to this chapter of the possibility of including certain input/output devices in the analysis. One feasible extension would be to include the file store. Belady and Weissman (1974) have suggested that the use of storage keys could be extended to backing store. If the details of such a scheme were worked out, it would be possible to compare their scheme with that used by DOS/VS to access the file store.

6.4 Evaluation of the Comparison

The above analysis has compared DOS/VS with an alternative protection system which uses the same protection mechanism, viz. storage keys, but which has been shown to offer greatly improved protection at little extra cost.

In this analysis, we have examined how well the protection system of DOS/VS restricts user processes to the portion of memory they require. The protection considered here is a first-level inter-process protection. Any further structuring or protection method imposed by the process itself, e.g., an Algol-type display mechanism, has been ignored.

Further improvements could probably be made to the alternative protection system if more information was available. For instance, if it was known which parts of the supervisor were required for each type of supervisor call, it is likely that improved protection could be obtained by making use of more values of the storage keys to implement a ring-like protection structure (Graham 1968).

The comparison of DOS/VS and the alternative protection system, based on the statistics obtained from the IBM 370/135 configuration, strictly represents only a comparison of the two protection systems in the context of the particular configuration and type of work load

used to supply the statistics. It is probably a reasonable assumption that this computer system is 'typical' of installations using DOS/VS, but before definitive conclusions concerning the alternative protection system could be reached, it would be necessary to gather similar statistics from other DOS/VS installations.

Before considering the comparison of the two protection systems, it is useful to look at the cost of performing the analysis. The statistics used in the analysis were collected by the operating system as part of its provision of an accounting facility, so it was only necessary to arrange for the statistics to be retained in a machine readable form by dumping them onto cards. A number of experiments were carried out on an IBM 360/67 system to establish the basis for estimating the supervisor-state time of each job-step. These experiments consisted of writing and then running short programs on the system. Finally, a program was written to perform the evaluation of the various measures. The total cost of evaluating the measures was thus relatively small in terms of man hours and machine time.

Apart from the usefulness of the comparison in its own right, it could be viewed as an exercise to examine the relative benefits and costs of the two protection systems, using available information, to see if there were grounds for justifying a substantial investment in a more detailed and widely based comparison. Considering the results of this comparison, it could be argued that there was indeed a case for collecting statistics on supervisor state time and gathering similar statistics from other DOS/VS installations. A further extension would be to consider gathering information on requests for more storage by user processes and analysing the costs involved in incorporating such a mechanism into a system.

As well as providing the basis for the comparison of the two

protection systems, the measure B1 and its derivatives have been used as figures of merit to investigate the protection implications of the memory organisation of DOS/VS. So a comparison such as this can also be useful in evaluating relatively minor changes to an existing system.

This chapter illustrates, therefore, that using a combination of available information and estimation, it is possible to perform a meaningful comparison of protection systems based on the proposed methodology.

Chapter 7

PROTECTION AND STRUCTURE

This chapter considers the use of structure to restrict further the accessible set and ways in which such structure may be provided. The concept of a domain is extended to that of a structured domain, where the relationship between objects contained within the domain can be exploited to provide additional protection.

A measure is proposed to assist in the comparison of systems which exploit structure for protection purposes. Use of the measure and of the concept of a structured domain are illustrated in brief comparisons of various protection systems.

Programmer defined types are a possible way of defining such structured domains, and a form of types and a possible implementation are presented in outline so that the protection aspects can be considered and compared with the protection available using conventional block structure.

Finally, the complementary notion of possible structured relationships between domains is discussed.

7.1 Structured Domains

The two studies contained in Chapters 5 and 6 have taken a one-level view of domains as have nearly all other people writing on the topic (e.g., (Lampson 1971),(Graham and Denning 1972),(Graham 1968), (Price 1973)). In the comparison of Algol W implementations, a domain was defined as a set of accessible areas of the stack and in the study of DOS/VS a domain was taken to be the accessible part of real or virtual memory. This approach to domains is sufficient for achieving gross comparisons, but ignores any structural information

contained in domains which could be used for protection purposes.

Domains are typically not composed of a set of uniform items, but are composed of many disparate items each with its own structure. Many domains can be usefully described by a tree representation. Each node in the tree represents an object, and the branches emanating from a node to other nodes represent the objects which are accessible from the given node. The domain itself corresponds to the root node.

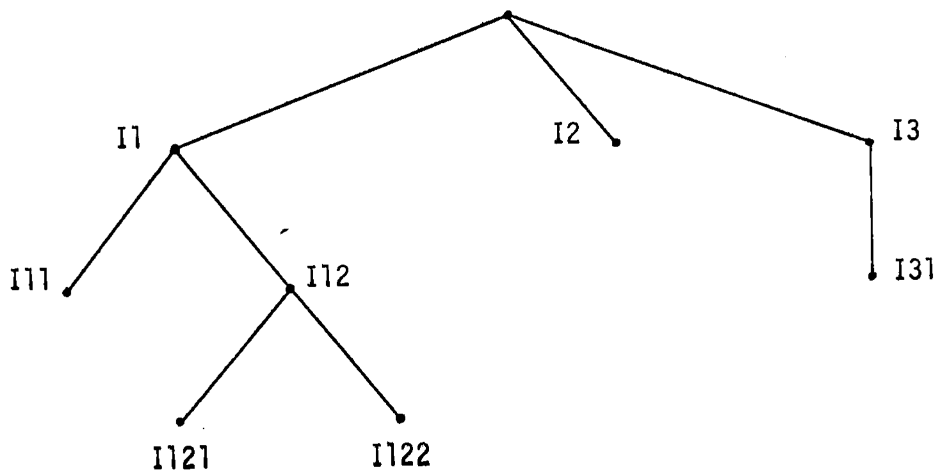


Figure 7.1

The tree representation of domain D

Domain D, shown in Figure 7.1, permits direct access to objects I1,I2 and I3. Object I31 has to be accessed via I3, and I121 via I1 and I12. To the programmer, I12 is an abstraction of the two objects I121 and I122. Similarly, I1 can be regarded as an abstraction of the objects contained in the subtree emanating from the node labelled I1. The objects which appear at the non-leaf

nodes may simply be descriptors permitting access to a vector of objects. In other words, it is convenient to regard a descriptor as an object in the same way that Lampson regards a capability as an object (Lampson 1969b).

The domains in which an Algol W program executes can be described in this way. Figure 7.2 shows a procedure P and the representation of the domain in which the activation of P executes. The domain is represented by a root node which has branches to the simple variables in the domain and the descriptors for any arrays or parameters. Thus, the root node permits access to the objects contained in the primary allocations of the accessible activation records. Certain of these objects, namely descriptors for arrays or parameters, themselves permit access to other objects such as array elements. Array descriptors permit access to parts of the secondary allocation of an activation record and parameter descriptors to objects stored anywhere on the stack. When executing in P, direct access is permitted to A,C and R, the actual parameter J is accessed via a descriptor and the elements of array X are also accessed via descriptors.

Not all domains are hierarchically structured; when there is more than one access path to an object, a lattice-like representation may be more appropriate (Lampson 1974).

To access objects in a domain which is hierarchically structured, it is necessary to traverse the tree representation using appropriate operations. The whole of the domain is not immediately accessible; the part of the domain addressable by the programmer at any instant is typically a true subset of the domain. A further protection device is to label the branches of the tree representation with the allowed operations which the programmer can invoke along each branch. Then the operation requested by the programmer limits the accessible set to

```
BEGIN
    INTEGER A;
    REAL ARRAY X (1::2,1::3);
    PROCEDURE P (INTEGER I);
    BEGIN
        INTEGER C;
        REAL R;
        :
    END P;
    :
    BEGIN
        INTEGER J;
        :
        P(J);
        :
    END;
    :
END.
```

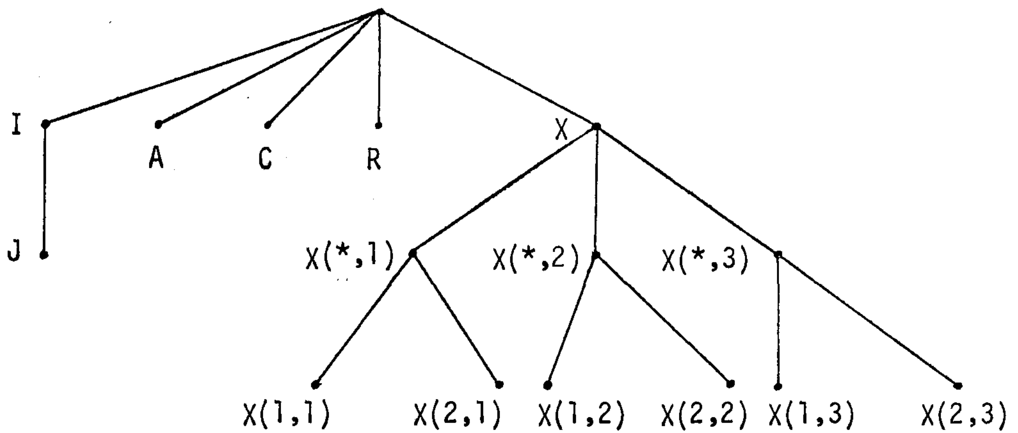


Figure 7.2

An Algol W program and the representation of the domain in which the activation of procedure P executes

the branches emanating from the current node of interest tagged with that operation. We have seen in Chapter 6 how read and write access select different accessible sets when only store protection is implemented on IBM 370 computers.

In this chapter, we are primarily concerned with the form of protection which Lampson (1974) terms defensive protection as opposed to absolute protection. Defensive protection tries to make it unlikely rather than impossible that protection violations will occur, that is, the system is defending against accident rather than malice. Absolute protection purports to guarantee that no matter what program is executing in a domain, it will be unable to break the protection barriers imposed by that domain. Absolute protection depends on correct hardware and software and the correct setting up of domains.

Between processes which are meant to be entirely independent (e.g., processes initiated from different users who do not even know of each other's existence) absolute protection is required. That is, the protection system should ensure that there is no interference between the processes. However, when there is intended to be cooperation between processes (e.g., users accessing a common data base, or even parallel processes in a single program put there in order to make good use of parallel hardware), absolute protection is required between the cooperating processes and other independent processes, but structure within domains can be exploited in the interests of defensive protection between the cooperating processes.

We define the direct scope of a program executing in a domain as the objects contained in the domain which the program can access with a single operation. Clearly this definition depends on the meaning of 'single operation'. For instance, fetching an array

element A(i) on a non-structured machine using an optimising compiler would be one rather than two operations. Thus, the direct scope of a program has to be considered in a context where the nature of operations is clearly understood.

The direct scope will vary during execution as changes occur in the set of objects which the program can access with a single operation. For a program executing in domain D shown in Figure 7.1, the initial direct scope includes the objects I1,I2 and I3. If object I1 is accessed the direct scope changes to objects I11 and I12. Thus, whenever the program accesses a non-terminal node of the tree-representation of the domain, the direct scope changes. The union of all possible direct scopes within a domain is termed the _total scope_ of the domain.

Absolute protection is not directly concerned with this structure, though it is useful conceptually for writing programs and for sharing objects between domains. A defensive protection system would attach value to the fact that certain objects can be accessed directly, while others cannot be referenced except by the execution of a carefully chosen sequence of operations.

In the case of defensive protection, small direct scopes are significant since they limit the instantaneous possibilities, but the total scope can be large without seriously affecting the protection. A measure of structured domains based on the idea of direct scope is presented in the next section.

7.2 A Measure of Structure within a Domain

A measure of the structure contained within a domain is the average direct scope within the domain compared to the total scope of

the domain,

$$\text{i.e.,} \quad \frac{\text{average direct scope}}{\text{total scope}}$$

There are various ways of quantifying the direct scope and total scope, similar to the possibilities available in measuring the 'size' of an accessible set. The obvious method is to use the number of objects contained in the direct scope and total scope. To accommodate the operations which can be applied to each object, an object 0 can be considered as a set of pairs $(0,\alpha_1),(0,\alpha_2),\ldots,(0,\alpha_n)$, where $\alpha_1,\alpha_2,\ldots,\alpha_n$ are the permitted operations for object 0, and each pair $(0,\alpha_i)$ considered as a separate object.

The average direct scope can be computed as the average of the direct scopes actually entered by a process during execution within the domain weighted by the number of operations invoked within each scope. An alternative is to take the average of all direct scopes within a domain, thus avoiding any dependence on the execution which actually takes place within the domain.

If the domain varies during execution due to domain changes, the measure can be evaluated as:

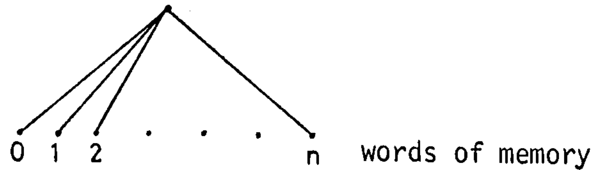
$$\text{average of} \quad \frac{\text{current direct scope}}{\text{current total scope}}$$

where the average is taken over those direct scopes entered by the process.

The measure for a particular domain will have a value between 0 and 1. A value of 1 indicates that the direct scope equals the total scope and thus the domain contains no structure which is used for protection. Values approaching 0 indicate a very highly structured situation with few objects contained within each direct scope.

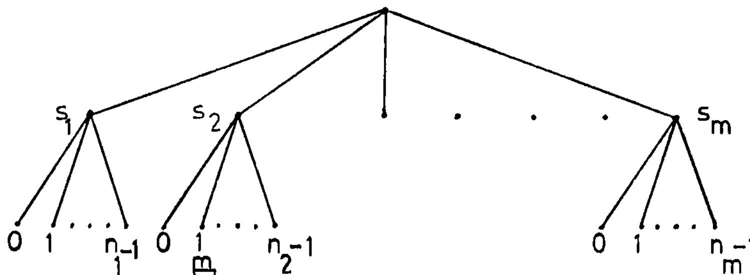
7.3 Comparison of Structured Protection Systems

A program executing in a domain consisting of a contiguous area of memory and using absolute addresses to access words in the memory area has the simplest of structured representations, viz.:



In such a domain, the direct scope equals the total scope and hence the value of the measure equals 1.

This can be compared with a program executing in a domain consisting of a number of distinct segments $s_1, s_2, \ldots, s_n$. Addresses used by the program take the form (s,l), where s selects a segment and l selects a word within that segment. The domain has a two-level structure:



Assuming that $n = \sum_{i=1}^{m} n_i$, that is the total accessible memory area is the same as in the previous domain, the direct scope is considerably reduced in comparison with the previous case. The average direct scope over the whole domain = $\dfrac{m+n_1+n_2+\ldots+n_m}{(m+1)}$. Thus, if the

$n_i \simeq {}^{n}/_{m}$, the average direct scope $\simeq \dfrac{m+n}{m+1}$ and the measure evaluates to

approximately $\dfrac{1}{(m+1)}$. As the number of segments, m, increases the measure tends to 0, which, compared to the previous case, indicates a significant improvement in protection. A refinement of this analysis would be to include the permitted accesses to each segment. For instance,

there may be three access modes to segments, Write, Read and Execute, but if some segments had less than the full set of access modes permitted, the average direct scope would be further reduced.

The second example concerns the use of codewords as proposed by Iliffe and Jodeit ((Jodeit 1968),(Iliffe 1969),(Iliffe and Jodeit 1962)) and implemented on the Rice computer (Feustel 1972). The purpose of using codewords is to retain structural information as an essential component of the representation of a program and its data.

Structuring is provided by the array mechanism, each array is named and contains as elements data or subarrays. The elements of an array form a block, a set of consecutive memory locations. Each block is labelled by a codeword (a word corresponding to the name of the array). If A is an array, the i-th element of A is denoted (A,i). If the elements of A are subarrays, the i-th word in the block for A is a codeword which labels the array (A,i).

Thus, an array is a tree structure. The codeword corresponding to the complete array is called the primary codeword. All subarrays and data elements of an array are addressed relative to the primary codeword.

Scalar quantities are represented directly in a value table. Other entities, such as programs, vectors, multi-dimensional matrices, structured data, etc., are represented by arrays whose primary codewords occur in the value table. The value table can be addressed directly by an executing program, thus the value table essentially defines the domain of the executing program. Elements of an array are addressed indirectly through the primary codeword for the array held in the value table. The indirect addressing is hardware implemented leading to an efficient implementation of codewords.

To access a particular element in an array, regardless of its position in the tree-structured representation of the array, the program inserts subscripts into a set of registers indicating the path to be taken through the array to reach the element. The program then addresses the primary codeword for the array and invokes the array element accessing instruction. The hardware traces a path through the array structure, using the subscripts in turn to determine the branch to be taken from each node in the tree representation of the array. Each subscript is checked, before being used, against information held in the current codeword to ensure that it is valid.

A two-dimensional square matrix M of n rows and n columns is thus represented by n vectors each n words long (see Figure 7.3).

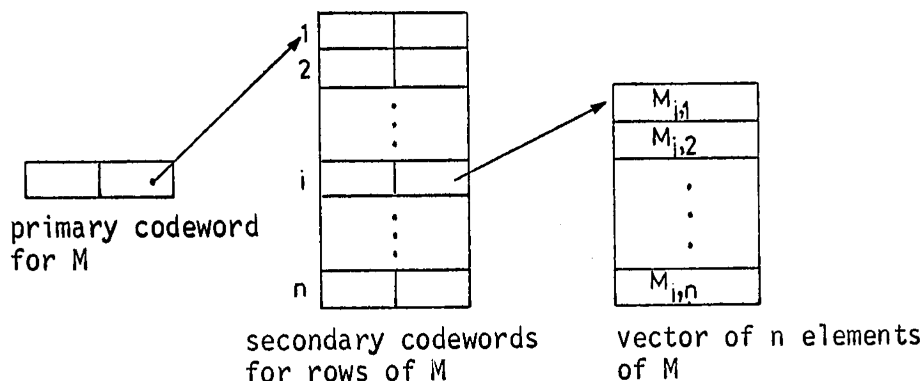


Figure 7.3

Codeword representation of a two dimensional square matrix

Confining attention to this array alone, the average direct scope is n compared to the total scope (n^2+n). The measure of structure evaluates to $\frac{1}{n+1}$, which compares favourably with the value 1 for a Fortran-type implementation of the same array.

The benefit of the codeword approach is a significant reduction in the size of the average direct scope (n compared to n^2 for a Fortran-like implementation of the above array) and automatic checking of array subscripts. The cost of the retained structure in this instance is (n+1) words of memory and the time taken to set up the codeword structure. The time taken to access an array element is likely to be similar to the time taken in a Fortran-like implementation, since it is implemented in hardware and the address of the element does not have to be calculated explicitly.

The use of a scalar measure to compare structured situations obviously has its limitations, but as illustrated above, it does lend some relative quantitative assessment in comparing structured protection systems. A particular limitation is that the measure of structure is defined in terms of operations and thus its evaluation will depend to a considerable extent on what is deemed to constitute an 'operation' in a specific situation. The array element accessing operation on the codeword machine, where the subscripts are put into registers and the hardware traces a path through the array structure to the designated element, can be treated as a single operation involving no switch in domains, but instead a number of changes of direct scope. In contrast, on a machine such as the Burroughs B5000, to access an array element it is necessary to program explicitly each individual indexing instruction involved in accessing the element. These instructions can be considered as separate operations some of which may involve a switch in domain. A third variation is the Algol W array element accessing operation which performs subscript checking. The programmer can consider this operation as a single operation, but he relies on the compiler to generate the correct code sequence to carry out the checking and accessing. This

introduces the question of abstraction; by viewing array element accessing as a single operation the programmer is forming an abstraction of a sequence of machine language operations. In practice, there are likely to be many levels of abstraction; an operation is an abstraction of a sequence of sub-operations which in turn can be broken down into sequences of sub-sub-operations, etc. The question arises as to the appropriate level at which to view operations. Also, even though an operation is regarded as atomic, the sequence of operations from which it is formed can perhaps be interleaved with operations from other processes, thus introducing questions of parallel process interference.

The assessment of structure poses the question of what is an operation and raises other issues such as what is a domain switch, what 'hardware' checking is actually being done, etc. Rather than discuss these issues in general, they are most clearly explicated in terms of a particular system design, specifically in terms of a type-based design.

7.4 Evaluation of a Type System

The intended function of types is to allow the programmer to define new forms of data structure together with matching operations to access instances of such data structures. Current interest in types arises because of their potential for increasing programmer effectiveness. The sole aim in presenting types here is to investigate the enhanced protection facilities available with types compared to the situation where blocks are the only structuring device which can be used to restrict access to program variables.

Standard languages such as Fortran, Algol and PL/1 have

independent means of structuring data and programs. For example, Algol W provides arrays and records for structuring data and procedures for programs. Types provide a uniform and coordinated means for structuring data and instructions.

The types considered are embedded in an Algol-like programming language, the details of which do not concern us here. The block structure can be used to restrict the scope of variables within a program, that is a specification of domains, and types ensure that access to programmer defined data structures is only permitted via the set of operations defined for that type of data structure, by providing a convenient means to specify and change direct scopes.

In the present situation, lacking a widely accepted typed language, indeed when much research into the subject is being carried out, we are forced into having to make certain choices regarding the facilities to be provided in such a language. Having made a choice, it provides a basis for the discussion of protection aspects. The particular choice of facilities is meant to be representative rather than definitive.

It is not appropriate to attempt a survey of all related work in this fast-moving area, but the form of types presented in this section is an amalgam of ideas from existing work on types: classes of SIMULA 67 (Dahl et al 1970), modes of Algol 68 (Wijngaarden et al 1974), types of Morris (1973a) and Wang (1974). The notion of types and the supporting architecture, which are briefly described in the next two sections, are the result of joint work between the author and Roy Campbell (Campbell and Wyeth 1974); the evaluation of the protection aspects of types was performed solely by the author.

7.4.1 Types in a Block Structured Language

Many languages, e.g., Fortran, Algol, PL/1, support simple data-types such as integer, real, boolean, character, etc., and operations (read, write, add, subtract,...) provided by the hardware which can be applied to variables of these data-types. The types described here include the primitive types mentioned above but also allow the programmer to create new types and specify a set of operations which can be applied to instances of the type.

A new type is introduced by a type definition. A type is a characterisation of a class of objects and, once defined, instances of that type can be declared. Example 1 shows the definition of a type Item, the notation used has been adopted for convenience and is not a proposed syntax.

A type definition must occur at the beginning of a block along with any other declarations contained in the block and applies for the duration of the block. Thus, instances of the type can be declared within the block containing the type definition and any inner blocks. We do not consider here the dynamic creation and deletion of objects in the sense of Algol W records (Wirth and Hoare 1966).

A type definition includes a schema or template for a data structure, which characterises individual objects of the type, and a set of operations which can be performed on such a data structure. In Example 1, the template specifies that an instance of an Item is composed of two integers, referred to as Part_No and Stock, and a real number Price. The only means of manipulating an object of a user defined type is by invoking an operation defined for that type of object. These operations are defined within the template and are termed object-operations.

```
begin
   :
   type  Item (integer P,S; real Cost);
         integer No_Items;
         initially: No_Items:=0;
         operation  No_of_Parts (returns integer I):
                    I:=No_Items;

         template
                integer  Part_No, Stock;
                real     Price;

                initially:
                   begin
                        Part_No :=P; Stock :=S; Price :=Cost;
                        No_Items :=No_Items + 1;
                   end;

                finally: No_Items :=No_Items - 1;

                operation  Allocate (accepts integer request):
                   if request ≤ stock then stock :=stock - request
                                      else error;

                   :

         end template;
   end type;
   Item Clutch(10451,100,10.41), Brake(10651,50,2.96);
   :
end;
```

Example 1

The declared Items, Clutch and Brake, can only be accessed via the operation Allocate and other operations defined within the template. The significant point of types is the protection of objects from forms of access other than the operations defined for that type of object.

The type Item provides the programmer with an abstraction of say car parts, and once this type definition has been specified he no longer need concern himself with the concrete representation of the objects he creates. The three components of the Item Clutch, viz.: Clutch.Part_No, Clutch.Stock and Clutch.Price, cannot be accessed directly outside the type definition. Thus the components of the Item Clutch can only be manipulated by invoking the appropriate object-operations.

An object-operation can only be invoked by qualifying the operation by the object to which it is to be applied, e.g.,

Clutch.Allocate(20)

Two distinguished operations may be declared: initially and finally. initially is invoked automatically on the execution of a declaration of an object of that type and finally is invoked when the object is deleted. The scope of the object-operations is the same as that of the type definition.

For completeness, a type definition is considered to be the declaration of a type object, an object of type TYPE, another predefined but non-simple type. The notion of a type being represented by an object is taken from the Hydra system (Wulf et al 1973). Treating type definitions as the declaration of an object of type TYPE, means that type definitions can be represented at run-time in a manner similar to other objects. A TYPE object includes any objects declared within the type definition but outside the template.

Such objects (e.g., No_Items in Example 1) correspond to 'own' variables of the class of objects of this type and characterise the whole class. These objects are only accessible within the environment (i.e., domain) of the type definition. Indirect access to these objects is gained via the type-operations which are declared within the type definition but outside the template. Type-operations have the same scope as the type definition in which they occur. The type-operation No_of_Parts, in Example 1, is intended to return the number of Items which have been declared. A distinguished type-operation initially may be declared with other type-operations, in which case it is invoked on execution of the type definition to initialise the objects declared in the type definition.

Object-operations, declared within a type definition, can access the component parts of the object to which they are applied by means of the names of the components as defined in the template. Such names are automatically interpreted with respect to the object on which the operation is acting. Type operations need no qualification to be invoked by a program.

To increase the flexibility in the use of types, parameters are allowed to type definitions. Such parameters are bound on declaration of an object of that type. Ideally, any form of parameter would be permitted including types themselves. However, certain parameters and the use of parameters in a way which affects the precise form of object created, when an object of the given type is declared, can lead to complications such as polymorphic types (see (Morris 1973a), (Liskov and Zilles 1974) and (Wang 1974) for a fuller discussion of this topic). Thus, we restrict parameters to simple variable parameters (i.e., variables of a non-user defined type) used in a

'simple' way. Example 1 used parameters to the type Item for the initialisation of the components of an object of type Item.

7.4.2 A Computer Architecture for Types

The purpose of presenting in outline a computer architecture for types is to demonstrate that it is practicable to design an architecture which supports at run-time the structure contained in a program written using blocks and programmer defined types. The protection rules of types are enforced at run-time by the architecture.

The prototype for the implementation is that used in Algol-60 run-time systems where activation records are allocated at the block level. Such a system was briefly described in Chapter 5 and is covered in detail by Randell and Russell (1964) and Wichmann (1973).

Since Algol-like scope rules apply to all objects, it is feasible to maintain all descriptors, representations of objects, working storage and linkage information on a stack. An activation record is created on block entry, procedure call, operation call and on the declaration of a variable of a user defined type (the term program segment will be used for the code corresponding to a block, procedure, operation or creation of an object). An activation record consists of four parts: linkage information, parameter section, primary storage and secondary storage. Access to activation records is via a set of hardware maintained display registers. Each active display register contains a pointer to the base of the appropriate activation record and has its limit extension set to the extent of the parameter section plus primary allocation.

Each object declared in the program segment corresponding to a given activation record has a descriptor in the primary storage. Objects of a simple type (i.e., integers, reals, etc.) have their

value contained in the descriptor for efficiency reasons. Descriptors in the primary storage are accessed via the appropriate display register.

The representations of arrays and objects of a user defined type, declared in the program segment, are contained in the secondary storage of the activation record. These representations are accessed via the descriptors in the primary storage.

Objects of simple types (i.e., integer, real, character, etc.) are represented directly by a word containing the value and tagged with the appropriate type (see Figure 7.4(A)). Objects of non-simple types (i.e., strings, arrays or user defined types) are represented, in the first instance, by a descriptor in the primary storage (Figure 7.4(B)). The type information contained in the first field of both

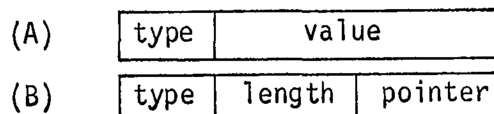


Figure 7.4

Descriptor Formats

descriptors takes the form of a pointer to the descriptor defining the type. The address of a descriptor which defines a type thus serves as the identity of the type. The number of contiguous 'words' to which a descriptor of format B permits access is contained in the length field, and the location of these words in the stack is given by the contents of the pointer field.

A non-simple object is defined recursively as either a simple object or an ordered set of non-simple or simple objects (c.f., Jones (1973)). Figure 7.5 illustrates the structure of a non-simple object. The descriptor representing the non-simple object in

primary storage (labelled A in the Figure) points to the ordered set of objects of which it is composed (B). Objects in B may be simple or non-simple and constitute the primary storage of the object.

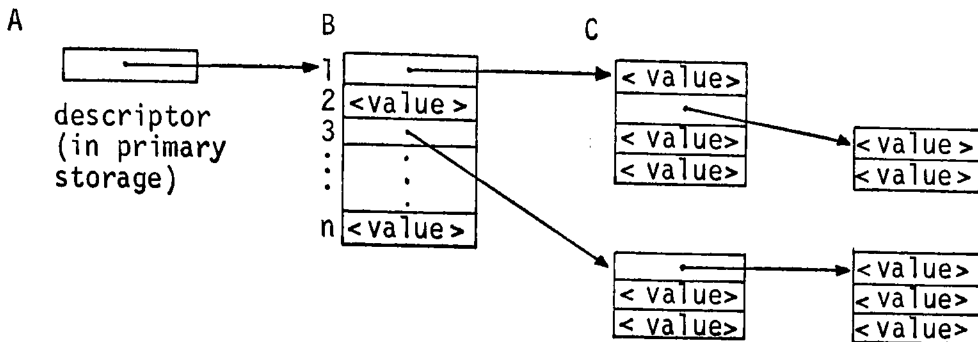


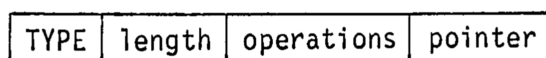
Figure 7.5

The hierarchical representation of an object

Descriptor A only permits access to the descriptors in B, not the total representation of the object. B_1 is a non-simple object and the descriptor points to its representation C. B_2 is a simple object and so its value is stored in the descriptor B_2. This process is carried on recursively until all the objects have been represented in terms of simple objects. The resulting structure is a hierarchical representation of an individual object.

Apart from the descriptor contained in the primary storage of the declaring block, the tree structure drawn in Figure 7.5 is kept in a contiguous set of words in secondary storage.

The descriptor for an object of type TYPE is a slight variant on the normal object descriptor and is shown below. The extra field,



operations, in this descriptor contains the total number of type-operations and object-operations declared within the type definition.

The representation of a type on the stack includes a descriptor for each of the 'own' variables of the type definition and a descriptor for each code component produced by the compiler. Figure 7.6 outlines the storage representation of a type definition.

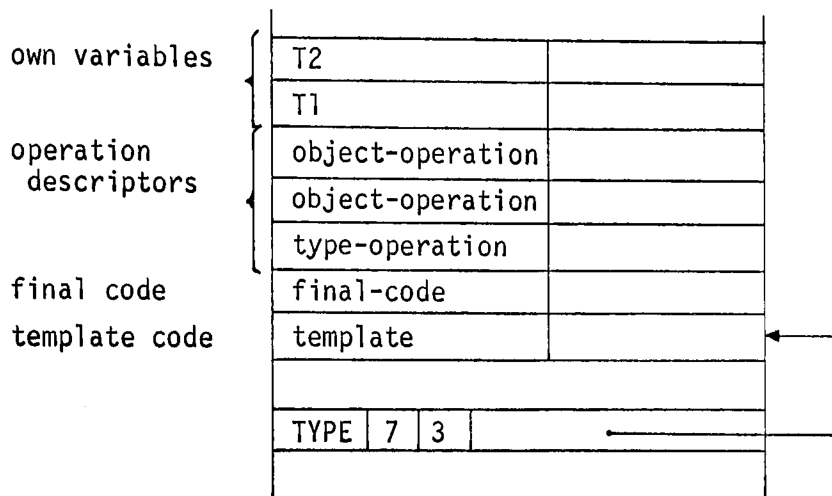


Figure 7.6

Outline of a type representation

On entry to a program segment the primary storage in the activation record is requested and allocated. Each declared object (including new types) gives rise to a CREATE instruction in the compiled code of the program segment. The descriptor for the object is tagged with the appropriate type and for objects of simple type no further action is required.

The representation of non-simple objects is built up in the secondary allocation by the CREATE instruction invoking the template code for the corresponding type. This is addressed by referencing the descriptor for the template code (by convention the first descriptor in the primary storage of the type object) in the appropriate type object. The template code sets up the representation of the object and performs any necessary initialisation. It is analogous

to the code produced by an Algol compiler for setting up a dynamic array.

On invocation of the template code, the display registers are updated to reflect the environment of the type definition, primary storage is requested and the address and extent of the primary storage are automatically put into the object descriptor. The template code creates on the stack, in its primary allocation, the first level representation of the object. In a similar manner, the representation of objects occurring in the first-level representation of the object are implemented in the secondary allocation of the template by invoking the template code for each such object in turn. This process is carried on recursively until the whole representation for the object has been created. On return from template code, the activation record of the template code is not removed from the stack since it forms the representation of the object. During calls of template code, linkage information can be stored on a separate control stack so it is not incorporated into the representation of the object being created.

Objects are deleted on exit from a program segment. For each object declared in the program segment, the distinguished operation finally is invoked, if defined for that type of object. The objects are then deleted by retracting the stack.

Procedures and operations are essentially similar, but an object-operation has a distinguished parameter, namely the object to which it is applied, whereas a procedure has a pointer to its associated environment (static link) included in its descriptor. A procedure or operation descriptor, shown below, contains the segment name of the code segment to be invoked, the static link and the type procedure, type-operation or object-operation.

procedure or operation	static link	segment name

The use of descriptors for procedures and operations means that they can be treated as objects implemented off the stack. A procedure is invoked by a call instruction which takes as its argument an address couple referencing the procedure descriptor.

Object-operation calls in the source language of the form

<object-name> . <object-operation-name>

are translated into

object-operation-call <address couple specifying object>
<operation number>

The operation descriptor is located by following the pointer in the type field of the descriptor for the object to the descriptor for the type. Indexing the descriptor for the type with the operation number yields the operation descriptor. The operation number is checked against the total number of operations contained in the type descriptor to ensure it is within the allowed range.

The domain of an object-operation is a concatenation of the domain of the type definition, the type definition, the local variables of the operation and the object to which the operation is applied. The descriptor for the object could either be the first descriptor in the parameter space of the activation record or be stored in a separate register. If the operation was compiled at program level k+1, one higher than its true program level k, the descriptor for the object could occupy the display register D(k). This would arise naturally if the template in the type definition was regarded as a block surrounding the operations. The first level representation of the object could then be referenced by

address couples, thus avoiding another addressing variant.

Type-operations are translated into

type-operation-call <type-address-couple> <operation-number>

The operation descriptor is located by indexing the type descriptor by the operation number. The management of display registers etc., for object-operations and type-operations is the same as that for procedures.

Only an outline of the machine architecture has been presented here, sufficient to facilitate a discussion of the protection aspects of types. A fuller exposition can be found in (Campbell and Wyeth 1974).

7.4.3 Evaluation

The previous section outlined an architecture which aims to achieve an implementation of block structure and types closely mirroring the source language structures apparent to the programmer. The philosophy underlying the run-time system is that the best time to check that an allowed operation is being applied to an object is just prior to the invocation of the operation. This does not run counter to policies of compile-time and load-time checking, but is to be seen as an additional checking time. A high degree of protection is achieved at run-time through a combination of a display mechanism, a tagged architecture, a dynamic type table and special hardware functions.

The advantages of base and displacement addressing have already been discussed in Chapter 5. Here, this addressing mode has been extended to the naming of the parts of which a composite object is formed. There is thus a uniformity of attitude to blocks and objects, each is considered to be structured into primary and secondary

allocations. Further, this addressing scheme enables objects to be named within the correct scope, but there is no means by which to manipulate the representation of an object except by invoking the appropriate operations. Object-operations are invoked via the descriptor for the type definition corresponding to the type of the object. This descriptor is located using the type tag in the descriptor for the object. Hence, only those operations defined for the object can be invoked to manipulate the object. When an object-operation is invoked, the object to which it is applied is automatically included in the domain and access permitted to the object's first-level representation.

Type information, in the form of type objects, is maintained on the stack during execution. Access to the type objects, which represent the various programmer defined types, is limited by the display mechanism. The maintenance of this type information thus constitutes a dynamic type table. Hydra, a system based on a similar form of types (Wulf et al 1974b), keeps type information (for the system and <u>all</u> users) in a single large type table. By keeping each user's type information in his own data-stack, the problems of accessing and protecting a large table are avoided. If a protection violation or an error occurs in a stack it only affects one user, whereas similar occurrences with the Hydra type table could affect other users.

A change in context, which takes place on the transfer of control from one program segment to another, will involve a change in the contents of some of the display registers and the stacking or unstacking of linkage information. Following the Burroughs B6700 hardware procedure call mechanism, the type machine would provide hardware or emulated instructions which perform the functions

associated with a context change. The integrity of the display registers and the linkage information is thus assured (hardware errors apart) as a program segment then has no means to access the registers or linkage information directly.

An evaluation of the protection aspects of the above architecture which supports programmer defined types or a comparison with other type systems is a difficult problem because of the need to consider structure. However, the notion of a structured domain and the proposed measure can help such an evaluation whereas the one-level view of the set of objects in a program, adopted earlier in this thesis, would not be satisfactory.

In evaluating the protection afforded by the implementation, we are interested in the improved protection offered within a block over the typical Algol implementation, and the cost at which this improvement is attained.

The increased benefit depends on the extent to which a programmer structures his domains using types, thus providing structural information to the machine. It would be quite possible to ignore the type mechanism altogether and write essentially normal Algol programs. There would be no special benefit offered by the architecture to such a user over a conventional Algol implementation and the costs would be similar to those of an Algol system.

We are concerned, principally, with the situation where programmers make extensive use of the facility to create new types to suit the requirements of each individual program.

In the type machine, arrays would be implemented in a manner analogous to that described for codewords in Section 7.3. In terms of the measure, such an implementation was seen to give a dramatic

improvement in protection, measured in terms of the reduction in the size of the direct scope. Some Algol implementations (e.g., Algol W) use a dope vector to assist in the calculation of the address of an array element and include sufficient information in the dope vector to permit subscript checking. Thus, there is not necessarily any improvement in protection in the case of arrays. However, these implementations provide run-time protection by means of compile-time provided conventions for use of the machine. If the compiler is subverted such conventions may not hold.

By using types, a programmer can impose on an address space a highly structured situation so that the direct scope will typically be much smaller than the total scope.

Consider the type <u>Person</u> defined in Example 2 and the structure of an object of type <u>Person</u> shown in Figure 7.7. An object of type Person is seen to be a reasonably complex data structure, but by defining it in this way various aspects of protection can be enforced at run-time which cannot be achieved with a conventional implementation of block structure. The average direct scope of the object shown in Figure 7.7 is 2.6, and the total scope is 13. Hence, the value of the measure for this object alone is .2.

If a <u>Person</u> is represented by individual variables: day, month, year, person-identity, address(1),...,address(5), the direct scope equals 9. So comparing an object of type Person with the variables implementing the corresponding object in a block implementation, the average direct scope is 2.6 in the first case and 9 in the second. This order of improvement was to be expected since the proposed measure evaluates to .2. For ease of discussion, only the structure of a single object has been considered, but with extensive use of types it is not unlikely that similar improvements could be expected for whole domains.

```
begin
    type Date;
        template
            integer Day,Month,Year;
            operation Set_Date (accepts integer A,B,C):
                begin  Day:=A; Month:=B; Year:=C;end;
            operation  Read_Year (returns integer I):
                .      I:=Year;
                .
                .
        end template;
    end type;

    type Identity;
        integer Number;
        initially: Number:=0;
        template
            integer Identity_No;
            initially: begin
                Number:=Number+1; Identity_No:=Number; end;
            operation Access (returns integer I):I:=Identity_No;
        end template;
    end type;

    type Address;
        template
            string(30) array Lines(1:5);
            operation Assign ...;
            operation Getline1 ...;
            .
            .
            .
        end template;
    end type;

    type Person;
        integer No_People;
        initially: No_People:=0;
        template
            Date Birth;
            Identity Person_Identity;
            Address Person_Address;
            operation Age (returns integer I):
                I:=Current_Year - Birth.Read_Year;
            .
            .
            .
        end template;
    end type;
    Person P1,P2,P3;
    .
    .
    .
end;
```

Example 2

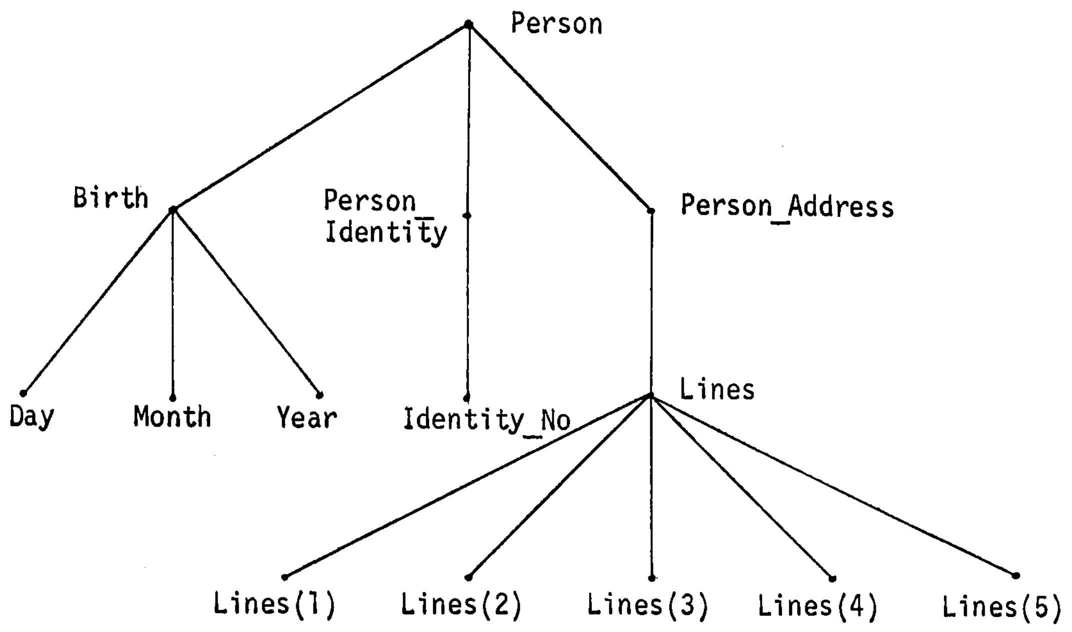


Figure 7.7

The structure of an object of type Person

The significant reduction in direct scope is not the only protection benefit to be expected from types. For instance, the components of the object named Birth within an object of type Person cannot be manipulated directly. From within the type definition Person, Birth can only be accessed by the operations defined for an object of type Date. Outside this type definition, even these operations cannot be invoked on Birth because there is no means of specifying the Birth component of an object of type Person. Access to Birth is restricted to the operation Age defined within the type definition Person. This protection would be enforced in the first place by the compiler but be validated by the proposed architecture. Similar comments can be made regarding the accessibility of other components of objects.

The type Identity is used to assign each Person a unique identity, namely Person_Identity. The generation of unique numbers requires an integer Number in the type definition for Identity. Since no type-operation is defined, Number can only be altered from within the template. Similarly, the unique number of each object of type Identity cannot be changed because there is no object operation defined to alter it.

The type mechanism in this example provides a restricted set of operations which can be used to manipulate objects. Since not all operations are valid for all objects, the direct scope will in practice be smaller in terms of (object,operation) pairs, than that calculated above.

The potential benefit of types and the proposed architecture is a significant reduction in direct scope of objects and operations compared to the use of conventional block structure without programmer

defined types. The reduction in direct scope results in increased reliability; errors are likely to be caught sooner than with a conventional Algol implementation and certain previously undetected errors will now be signalled as protection violations. When an error does occur, since objects are represented at run-time in a way which can be related to their source program definition, errors can be reported in source language terms, thereby assisting the diagnosis and correction of errors.

The costs of this protection benefit can be considered in terms of space and time. Space is required on the stack for the representation of type objects, the descriptors defining the structured relationships of the components of each object and the linkage information which must be stored when each new direct scope is entered. Extra processing time is required to set up type objects and the representations of individual objects, and to perform the increased number of context changes. A context change will occur on each change of direct scope and will involve alteration of the display registers and the storing of linkage information. In addition, the proposed hardware itself is more complex than conventional computers and this could increase the hardware cost and the basic cycle time of the machine.

The proposed machine architecture reflects closely at run-time the source language structure of the program. Retention of structure enables the protection provided at run-time to match closely the protection specified by the programming language. The structure is exploited to achieve an efficient implementation of protection through addressing. It is the author's belief that in a suitably designed system, the costs of maintaining and using the system will be outweighed by the benefits.

This discussion has concentrated on the use of types within a single process. The extension to a machine design supporting cooperating processes which share objects of programmer defined types is not yet complete due to unresolved problems of such an architecture in connection with parallelism. Ways of structuring the relationships between domains, discussed in the next section, and the work of Campbell (Campbell and Habermann 1974) on process synchronisation, may provide techniques to overcome these problems.

Other proposed or extant implementations of types include those of Wang (1974) and Morris (1973a) and the Hydra system (Wulf et al 1974b).

In his thesis, Wang proposes a method of implementing his type language but he does not explicitly represent the structure of an object at run-time, reducing all objects to collections of primitive objects at compile-time. The consequence of this is minimal run-time protection. The proposals of Morris for types are similar to those presented in this Chapter, but his implementation is radically different, using an unstructured storage space whereas we propose implementing all objects on a stack.

The Hydra system also presents a similar view of types but without the overall block structure. Types encompass the whole computer system and thus present a uniform protection method to the programmer. In the Hydra system, an object has two parts: a data part and an object reference part. The objects which a process can name directly are specified by the capabilities in the process's local name space, which is akin to the capability list of Dennis and Van Horn (1966). An object which is a composition of other objects has capabilities for these objects in the object reference part of the given object. Circular links are prevented, thus the resulting

set of objects composing a domain and their structure can be represented by a tree. Objects in the domain are referred to by a path-name which indicates the path to be taken through the structured domain to reach the object. For instance, the path-name i.j.k specifies the k-th capability in the (capability part of the) object named by the j-th capability in the object named by the i-th capability in the local name space of the process. The Hydra system offers more flexibility than the proposed type machine but with the result that the cost of a domain switch is high, thus discouraging the use of a large number of small domains and domains with a large number of small direct scopes.

7.5 Complementary Notion of Structured Relationships between Domains

The use of structure within a domain for protection purposes has been discussed at length in this chapter and, in this final section, the complementary notion of structured relationships between domains is briefly considered. Two possible types of relationship are presented as examples. The first type of structuring is that found in block structured languages where domains are linked to blocks and the second is the strict nesting of domains.

A program written in an Algol-like language can be represented by a tree structure where each node corresponds to a block or procedure. The branches emanating from a node link the given node to nodes representing blocks or procedures declared within the given block or procedure. The root node represents the outermost block of the program. Figure 7.8 shows the tree representation of a simple Algol program.

The tree is a representation of the static structure of the program. Execution of the program, or the program's dynamic structure,

```
begin
    procedure A;
    begin ... end;
    :
B:  begin
        :
     C: begin
            :
        end;
        :
     D: begin
            :
        end;
    end;
end.
```

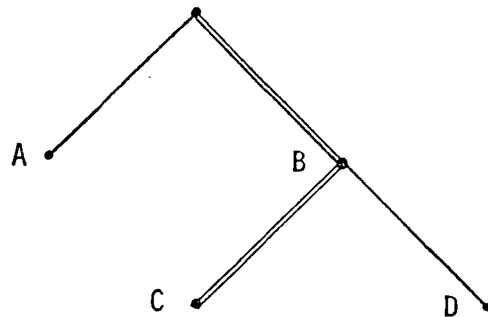


Figure 7.8

An Algol program and its tree representation

is represented by a path from the root node to the node representing the block in which execution is currently taking place. In Figure 7.8, the path marked indicates execution within the block labelled C. If each node is also deemed to represent the set of objects declared within the corresponding block or procedure, the path through the tree, indicating the current execution state, also indicates the currently accessible objects, apart from parameters to procedures.

Each node corresponds to a block or procedure, and since domains are tied to blocks and procedures, a node represents a possible domain of execution. The tree structure is thus a representation of the

relationship between the domains. The interpretation of the tree structure is that the domain represented by a node includes the objects associated with that node (that is objects declared within the corresponding block or procedure) and objects associated with nodes on the path from the given node to the root node. Hence a domain includes those domains on the path to the root as subsets.

A second type of relationship is illustrated by the process structure of the Cambridge CAP computer (Walker 1973),(Needham and Walker 1974). Processes are structured in a hierarchical manner so that any process can only access a subset of its father's resources.

The protection implemented by CAP is memory protection, the basic unit of protection being a segment which is viewed as a contiguous set of words. Input/output protection can be achieved via memory protection by assigning each device an address and either including or not including that address in the process's address space.

The segments which a process can access are defined by the process's capability segment (PsCS). For each segment which the process can access, there is a capability in its PsCS containing, inter alia, the identity of the segment, in terms of a capability in the PsCS of the father process, and the type of access the process can make to the segment.

A process can act as a coordinator for a number of processes, that is a process can spawn junior processes, providing any operating system functions such as interlocks and the control of the allocation of the CPU for the junior processes. A junior process is created by defining its PsCS in terms of the capabilities contained in the PsCS of the coordinator. The capabilities in the

PsCS of the junior process are refinements of those in the coordinator's PsCS, that is represent a subsegment or a reduction in access permissions.

Thus, the architecture of the CAP computer supports a hierarchy of processes, where each process knows of its father and its junior processes but does not know if a junior process has set itself up as a coordinator and is multiprogramming among a set of junior-junior processes.

A process addresses a word in a segment by specifying a capability in its PsCS and the offset of the word within the segment. The access type is checked against the allowed access types defined within the capability and the offset is checked that it lies within the segment. The address is then converted, by means of information contained within the capability, to an address in the universe of discourse of the coordinator of the process. This address in turn is converted to an address valid within the universe of discourse of the coordinator's coordinator, etc., until the address space of the master coordinator is reached. The capabilities in the master coordinator's PsCS (alternatively termed the master segment list) define the actual identities of the segments, which may then be mapped onto physical storage addresses by a paging mechanism.

Providing a process is not passed a capability for its own PsCS or any PsCS on the path from its own PsCS to the master segment list, coordinators on this path are fully protected from the effects of junior processes (save what a process may do to the segments for which it has been passed capabilities).

A domain of execution of a process is defined by its PsCS, which in turn is defined in terms of the surrounding domain. The strict hierarchical structure of processes, and hence of domains, achieves

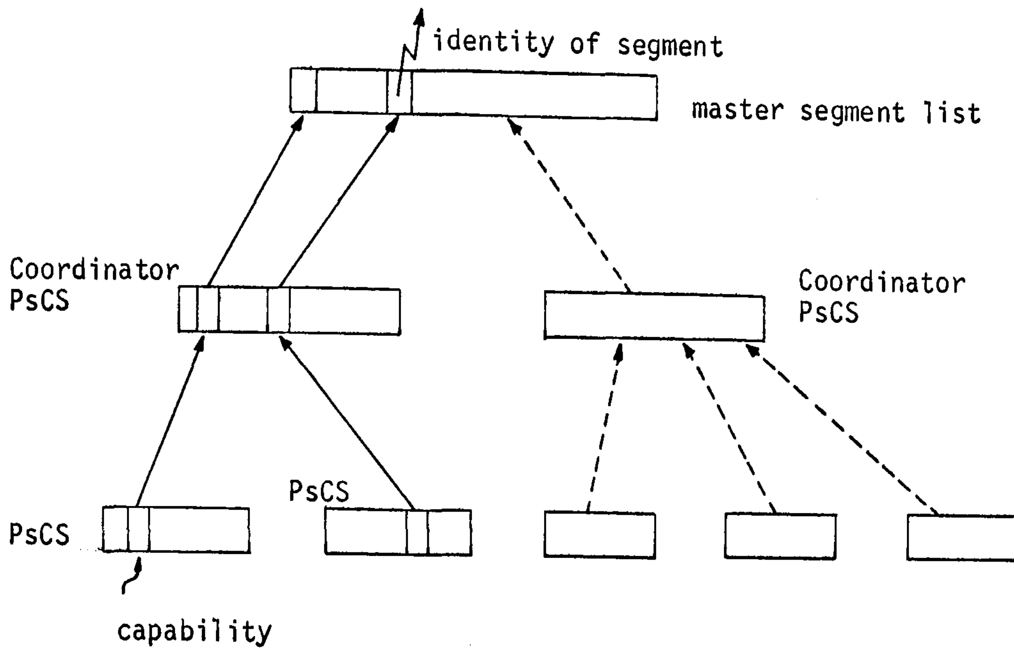


Figure 7.9

Domain and process structure of CAP

inter-process protection.

Figure 7.9 shows the hierarchical structure of a small set of domains, represented by the processes' PsCS's, and indicates that a capability in a PsCS is defined in terms of capabilities in PsCS's on the path to the master segment list. Though the domains are related by a tree structure, as in the previous example, the tree structure now has to be interpreted in terms of containment. A domain, represented by a node, is contained within the domain represented by the next node on the path to the root node. This contrasts sharply with the previous case where the node further from the root represented a domain containing the domain corresponding to the node nearer the root.

Lauer and Wyeth (1973) have presented a machine design which also supports a hierarchically structured set of domains and is basically similar to the CAP architecture presented above. The advantage of both these designs is that there is no need to treat any of the information which describes a domain as sacred to the basic protection system. Each level, i.e., process, can be responsible for setting up the data structures which define the domains for the next junior level and the memory protection system simply interprets these structures. Since any addresses produced by a process will be interpreted in the domain provided by the coordinator, there is no need to constrain the addresses a process can produce. The domain structure provides multiple levels which are never collapsed into a single level. An error made in describing the domain which a coordinator presents to a junior process may cause harm to the coordinator but not to the coordinator's coordinator.

In contrast to the hierarchical process structure, within process protection in CAP is completely flexible. The unit of protection is the 'procedure', which may or may not correspond to a high level language procedure, the choice is left open. The segments the procedure can access when being executed are defined by capabilities in a set of four indirectories. The details of these indirectories are described fully by Walker (1973). Basically, the indirectories used by a procedure invocation define the domain of the procedure as a subset of the PsCS. To be strictly correct, Figure 7.9 should be drawn with indirectories interspersed between PsCS's.

It should be mentioned that there are difficulties with architectures along the lines of the CAP computer and the Lauer-Wyeth architecture concerning efficient implementation when parallelism exists, also that CAP restricts itself to a two-level

hierarchy because of protection problems with ENTER capabilities (an ENTER capability embodies permission to call a procedure). The Lauer-Wyeth architecture has similar problems in connection with invocations of operations by a junior process, where the operation is implemented by an ancestor process, and interrupts from external devices.

The complementary notion of a structured relationship between domains has been discussed in terms of two examples. Thus, it has been demonstrated that it is possible to gain protection from structure between domains and from structure within a domain. The two views of structure in practice tend to merge into one, since a direct scope can be considered to be a domain and then a change of direct scope becomes a switch in domain. In this section, emphasis has been placed on the structure existing between domains, but, as the second example demonstrated, domain structure is often closely tied to process structure. Various forms of process structuring have been discussed in detail by Horning and Randell (1973). In the design and comparison of protection systems, the structure within a domain should not be considered in isolation, but in conjunction with the structure of the relationship between domains and the structure between processes.

Chapter 8

CONCLUSIONS

The work reported in this thesis has demonstrated that it is possible to develop quantitative ways of comparing protection systems and that the assessment based on cost and benefit measures is indeed both practical and useful.

Comparisons of protection systems based on the cost and benefit measures are in a sense gross comparisons since much of the fine detail of a protection system is ignored. Because of this, the comparison methodology is not claimed to be a complete assessment method, but is to be viewed as a part of a comprehensive evaluation. In addition to the gross quantitative comparison, such an evaluation would need to examine the detailed mechanics of protection systems and for this the protection models of Lampson (1971) and Jones (1973) would be useful. Also, it would be necessary to assess the influence of a protection system on its environment, that is on the structure of the operating system, the writing of user programs, etc.

The quantitative comparison methodology is thus complementary to the protection models developed by Lampson and Jones. Apart from the suitability measure of Jones, these models do not provide a yardstick to be used in any form of quantitative comparison. The limitations of the suitability measure were discussed in Chapter 3.

The use of cost and benefit measures in comparing protection systems is new. The advantage of this quantitative comparison methodology is that one gains an appreciation of how much better one protection system is compared to another and at what extra cost, if any, this increased protection is obtained. Qualitative comparisons based on formal protection models are hampered by the attention to

detail and the disregard paid to costs. A further advantage of the quantitative approach is that various features of a protection system can be individually assessed in terms of the protection benefit they afford and their cost. Unlike the suitability measure of Jones, the comparison methodology does not require detailed knowledge of the accessing patterns of executing programs but only knowledge of domains and the corresponding implemented accessible sets.

The utility of the comparison methodology is exemplified by the two detailed comparisons performed in Chapters 5 and 6, but to extend the applicability of this approach it is necessary to take account of structure. Tools for the assessment of protection systems which exploit structure to achieve protection, namely the concept of a structured domain and a measure of structure, were described in Chapter 7.

The comparison methodology and the techniques for assessing the exploitation of structure are probably most profitably used in the design of new protection systems and their implementations. Two areas where this work may be of assistance are the design of new machine architectures and the design of new run-time systems for high-level languages. The methodology provides a means of performing a quantitative comparison of a proposed system with an existing similar system. Even if the full quantitative comparison cannot be carried out, it may be possible to use the protection model and measures developed in Chapter 4, together with available statistics or estimates, to perform a semi-quantitative assessment. As mentioned above, the quantitative method of assessment also allows particular aspects of a protection system to be compared with each other. Further factors which may assist the design of new protection

systems are the concept of a domain, the relationship between protection and addressing, and the utilisation of structure for protection; all three aspects have been considered in detail in this thesis.

The comparison of structured situations is recognised to be a very complex problem. A number of existing protection systems retain structural information at run-time so that it can be exploited for protection. The concept of a structured domain describes many structured situations in a useful manner, but it is only a first step. The suggested measure of structure, though crude, attempts to quantify the benefit of the structured situation over the unstructured. The assessment of structure relates back to the relationship between protection and addressing, discussed in Chapter 2, since the retained structure is typically used by the addressing mechanism to limit accessibility.

Chapter 7 is only the start of the work which remains to be done in this area. A further step in the assessment of structure and its exploitation to achieve effective and efficient protection mechanisms would be to gain a combined understanding of process structuring, domain structuring and the structuring of objects within a domain. Such an understanding should lead to an appreciation of how processes should be structured to realise the maximum benefit from domains, and the identification of improved ways to specify domains and domain switches. A further outcome would probably be pointers to the further development of protection systems, particularly how the protection embodied in user defined types could be improved or extended.

An area of research suggested by the protection model, described in Chapter 4, but not pursued in this thesis, is a study of the

relationship between domains and locality, that is between the accessible set A and the referenced set R. A study of symbolic address tapes, where references to objects are recorded in source language terms, to analyse the correlation between static program structure and dynamic behaviour, could lead to an investigation of better ways to represent dynamic behaviour statically. Static structure is significant to protection since in the form of context information it is used to indicate domain definitions. In addition, if the static structure is sufficiently precise, it can convey predictive information concerning the objects likely to be accessed in a given period of execution. The importance of locality to language and machine design has been underlined in a recent Department of Industry report:

> "There is a need for a systematic representation of the concept of locality within programming languages, and for a uniform implementation of this representation in computer architecture. Until this is achieved, it is likely that both language and architectural developments will continue in an ad hoc manner, and be in conflict with one another."
>
> (Department of Industry 1975)

Related to this work is research into the best way of specifying redundancy (Randell 1975). Not all redundancy is 'useful', that is can be exploited to predict future actions of a process or detect error situations, indeed any form of redundancy will lengthen the program text, thereby increasing the opportunities for clerical errors.

The relation between machine architecture and programming language structure is a topic of current interest of direct relevance to protection (Department of Industry 1975). The aim of present work is the development of programming languages and machine architectures which encourage in a 'natural' way use of the facilities provided by the protection system. The two areas of work mentioned above are directly relevant to this aim.

A possible way of achieving this aim is to design a machine and programming language such that one uniform protection mechanism is used throughout the system. The systems likely to succeed in this respect are those which allow domains to be specified and modified so conveniently that the decomposition of a process into domains need not be unduly constrained by the cost of transmitting arguments or switching domains. Languages permitting user defined types provide a uniform protection mechanism to programmers if supported by a suitable machine architecture. Such a language and architecture were outlined in Chapter 7, but there is still a lot more work to be done on languages which provide user defined types.

Appendix

ESTIMATION OF SUPERVISOR STATE TIME

To estimate the time spent in supervisor state by a job-step, required in Chapter 6 to evaluate benefit measure B1, it is assumed that the supervisor state time arises from input/output requests issued by the executing job-step. The servicing of page faults and interrupts, and the functions of scheduling, dispatching and resource allocation etc., are treated as part of the general supervisory function of the system. Therefore, the time spent by the supervisor on these functions is not allocatable to user job-steps.

The amount of supervisor state time (as opposed to I/O transfer time) necessary to transmit n_d records (card images, lines to be printed, disk records etc.) to or from device d is assumed to take the general form

$$\begin{cases} \alpha_d + \gamma_d n_d & n_d > 0 \\ 0 & n = 0 \end{cases}$$

where α_d and γ_d are constants. α_d represents some set-up cost, e.g., establishing the logical-physical device mapping, and γ_d the incremental cost for each record transmitted. The total supervisor state time for job-step j is then given by

$$\text{supervisor state time for job-step } j = \sum_{\substack{d \\ (\text{s.t. } n_{jd} \neq 0)}} (\alpha_d + \gamma_d n_{jd})$$

where the summation is taken over those devices d for which $n_{jd} \neq 0$, and the problem state time is then the total execution time of the job-step less the supervisor state time.

The form of this function to represent the supervisor state time was deduced as an approximate model from initial experiments carried out on an IBM 360/67 system.

The numbers of records transmitted to or from each device by each job-step (i.e., n_{jd} d=1,2,...,m) were included in the statistics collected by DOS/VS. To estimate the supervisor state time, therefore, it was only necessary to obtain values for the device constants α_d and γ_d for each device.

The following devices were supported by the 370/135 system:

console
printer
card reader
card punch
optical character reader
disks
remote job entry lines
dummy card reader } used by the
dummy card punch } spooling system
dummy printer }

To reduce the number of constants to be determined, the optical character reader was assumed to have the same characteristics as a card punch and the remote job entry lines were treated as card reader/punches. On average, only seven job-steps each day used the optical character reader and only one job-step, the POWER spooling program in the first shift each day, used the remote job entry lines, so any errors introduced by these assumptions would be slight.

Ideally, the device constants should be determined on the system from which the statistics were gathered, or at least on a 370/135 system with a similar configuration. Unfortunately, such a system was not available, and so the method adopted was to determine the constants on an IBM 360/67 system running under the Michigan Terminal System, and then to adjust the constants for the difference in speed of the 135 and 67.

On the 360/67 system, the constants were determined by recording the supervisor state time used in the transmission of various numbers of records to or from each device under consideration. From these results, the constants for the 67 could be estimated and in turn those for the 135 derived.

The method of timing statements was essentially that used by Wichmann (1969) to time Algol statement execution times. The various corrections to the measured times, suggested by Wichmann, turned out to be negligible compared to the times being measured and thus could be ignored.

The 360/67 system running under the Michigan Terminal System is a paging system and thus has many of the same characteristics as the 370/135 system, though in almost all respects it is a much more sophisticated system. It was felt that estimating the device constants by using the 360/67 system was a reasonable approach, but to increase confidence in the values obtained some experiments were performed to verify the results.

The proportion of supervisor state time to total execution time for a number of programs on the 360/67 system was measured. The proportion of supervisor state time to total execution time was calculated for all job-steps run during one day on the 370/135 system, using the device constants and input/output counts to estimate the supervisor state time. The average proportion of supervisor state time and the variation in the proportion were sufficiently similar in both cases to justify the method used to estimate the supervisor state time.

REFERENCES

Baker F.T. (1972)
Chief Programmer Team Management of Production Planning
IBM Systems Journal 11,1 (1972) pp 56-73

Bauer H., S. Becker and S. Graham (1968)
Algol W Implementation
Technical Report No CS98 (May 1968), Computer Science Department
Stanford University

Belady L.A. and C. Weissman (1974)
Experiments with Secure Resource Sharing for Virtual Machines
Proc. of International Workshop on 'Protection in Operating Systems', IRIA (August 1974) pp 27-34

Birch J.P. (1973)
Functional Structure of IBM Virtual Storage Operating Systems
Part III: Architecture and Design of DOS/VS
IBM Systems Journal 12,4 (1973) pp 401-411

Brinch Hansen P. (1970)
The Nucleus of a Multiprogramming System
CACM 13,4 (April 1970) pp 238-241

Brinch Hansen P. (1973)
Operating System Principles
Prentice Hall (1973)

Brundage R.E. and A.P. Batson (1974)
The Performance Enhancement of Descriptor-Based Virtual Memory Systems Through the Use of Associative Registers
Proc. of 2nd Annual Symposium on Computer Architecture in
Computer Architecture News (ACM-SIGARCH) 3,4 (December 1974)

Burnell G.J. and E.G. Coffman (1975)
Analysis of Interleaved Memory Systems Using Blockage Buffers
CACM 18,2 (February 1975) pp 91-95

Burroughs Corporation (1972)
Burroughs B6700 Reference Manual
Form 1058633, Burroughs Corporation, Detroit, Michigan (1972)

Campbell R.H. and A.N. Habermann (1974)
The Specification of Process Synchronisation by Path Expressions
Lecture Notes in Computer Science, Volume 16, Springer-Verlag, Berlin (1974)

Campbell R.H. and D. Wyeth (1974)
A Computer Architecture for a Language Permitting Programmer Defined Types
SRM/95, Computing Laboratory, University of Newcastle upon Tyne (July 1974)

Dahl O-J., E. Dijkstra and C.A.R. Hoare (1972)
Structured Programming
Academic Press, New York 1972

Dahl O-J.B. and K. Nygaard (1970)
The Simula 67 Common Base Language
Norwegian Computing Centre (1970)

Denning P.J. (1968)
The Working Set Model for Program Behaviour
CACM 11,5 (May 1968) pp 323-333

Dennis J.B. and E.C. Van Horn (1966)
Programming Semantics for Multiprogrammed Computations
CACM 9,3 (March 1966) pp 143-155

Department of Industry (1975)
The Future of Real Time Technology
A Report to the Computers, Systems and Electronics Requirements Board
Edited by I.M. Barron, Department of Industry (August 1975)

Digital Equipment Corporation (1972)
PDP 11 Handbook
Maynard, Massachusetts (1972)

England D.M. (1972)
Operating System of System 250
Proc. of International Switching Conference,
Boston, Massachusetts (June 1972)

Fabry R.S. (1971)
List-Structured Addressing
Ph.D. Thesis, University of Chicago (March 1971)

Fabry R.S. (1973)
Dynamic Verification of Operating System Decisions
CACM 16,11 (November 1973) pp 659-666

Fenton J.S. (1974)
Memoryless Subsystems
Computer Journal 17,2 (May 1974) pp 143-147

Feustel E.A. (1972)
The Rice Computer - A Tagged Architecture
AFIPS SJCC Volume 40 (1972) pp 369-377

Feustel E.A. (1973)
On the Advantages of Tagged Architecture
IEEE Transactions on Computers C-22,7 (July 1973) pp 644-656

Fraser A.G. (1971)
On the Meaning of Names in Programming Languages
CACM 14,6 (June 1971) pp 409-416

Graham G.S. and P.J. Denning (1972)
Protection: Principles and Practice
AFIPS SJCC Volume 40 (1972) pp 417-429

Graham R.M. (1968)
Protection in an Information Processing Utility
CACM 11,5 (May 1968) pp 365-369

Hatfield D.J. (1972)
Experiments on Page Size, Program Access Patterns and Virtual Memory Performance
IBM Journal of Research and Development 16,1 (January 1972) pp 58-65

Hatfield D.J. and J. Gerald (1971)
Program Restructuring for Virtual Memory
IBM Systems Journal 10,3 (1971) pp 168-192

Hauck E.A. and B.A. Dent (1968)
Burroughs B6500/B7500 Stack Mechanism
AFIPS SJCC Volume 32 (1968) pp 245-251

Hawkins E.N. and D.H.R. Huxtable (1963)
A Multi-Pass Translation Scheme for Algol 60
in Annual Review in Automatic Programming Volume 3 pp 163-205
Edited by R. Goodman, Pergamon Press, 1963

Hoffman L.J. (1969)
Computers and Privacy - A Survey
Computing Surveys 1,2 (June 1969) pp 85-103

Hoffman L.J. (1971)
The Formulary Model for Access Control
AFIPS FJCC Volume 39 (1971) pp 587-601

Horning J.J. and B. Randell (1973)
Process Structuring
Computing Surveys 5,1 (March 1973) pp 5-30

IBM (1968a)
IBM System/360 Principles of Operation
IBM Report No GA22-6821-7 (1968)

IBM (1968b)
IBM System/360 Operating System: Concepts and Facilities
IBM Report No GC28-6535-4 (1968)

IBM (1971)
Concepts and Facilities for DOS and TOS
IBM Report No GC24-5030-11 (1971)

IBM (1972)
IBM System/370 Principles of Operation
IBM Report No GA22-7000 (1972)

IBM (1973)
Introduction to DOS/VS
IBM Report No GC33-5370-2 (1973)

Ichbiah J.D., J.P. Rissen and J.C. Heliard (1973)
The Two-Level Approach to Data Independent Programming in the LIS Implementation Language
Proc. of ACM SIGPLAN-SIGOPS Interface Meeting on Programming Languages and Operating Systems, Savannah, Georgia
SIGPLAN Notices 8 (September 1973) pp 74-79

Iliffe J.K. (1969)
Elements of BLM
Computer Journal 12,3 (August 1969) pp 251-258

Iliffe J.K. and J.G. Jodeit (1962)
A Dynamic Storage Allocation Scheme
Computer Journal 5,4 (1962) pp 200-209

Jodeit J.G. (1968)
Storage Organisation in Programming Systems
CACM 11,11 (November 1968) pp 741-746

Jones A.K. (1973)
Protection in Programmed Systems
Ph.D. Thesis, Carnegie-Mellon University (1973)

Lampson B.W. (1968)
A Scheduling Philosophy for Multiprocessing Systems
CACM 11,5 (May 1968) pp 347-360

Lampson B.W. (1969a)
Dynamic Protection Structures
AFIPS FJCC, Volume 35 (1969) pp 27-38

Lampson B.W. (1969b)
On Reliable and Extendable Operating Systems
Working Material Volume II, NATO Conference on Software Engineering Techniques, Rome, (October 1969)

Lampson B.W. (1971)
Protection
Proc. 5th Annual Conference on Information Sciences and Systems
Princeton University (March 1971) pp 437-443

Lampson B.W. (1973)
A Note on the Confinement Problem
CACM 16,10 (October 1973) pp 613-615

Lampson B.W. (1974)
Redundancy and Robustness in Memory Protection
Information Processing 74, North-Holland (1974) pp 128-132

Lauer H.C. and C.R. Snow (1972)
Is Supervisor State Necessary?
Proc. of the ACM-AICA International Computing Symposium
Venice (1972) p 293

Lauer H.C. and D. Wyeth (1973)
A Recursive Virtual Machine Architecture
Proc. of ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems
Harvard (March 1973) pp 113-116

Liskov B. and S. Zilles (1974)
Programming with Abstract Data Types
Proc. Symposium on Very High Level Languages, SIGPLAN Notices 9
(April 1974) pp 50-59

Madison A.W. and A.P. Batson (1975)
Quantification of Program Locality
Proc. of ACM Computer Science Conference, Washington D.C. (February 1975) p 8

Morris J.H. (1973a)
Protection in Programming Languages
CACM 16,1 (January 1973) pp 15-21

Morris J.H. (1973b)
Types are not Sets
Proc. of ACM Symposium on Principles of Programming Languages Boston, Massachusetts (October 1973)

Morrison J.E. (1973)
User Program Performance in Virtual Storage Systems
IBM Systems Journal 12,3 (1973) pp 216-237

Myers G. (1975)
Reliable Software Through Composite Design
Petrocelli (1975)

Needham R.M. (1972)
Protection Systems and Protection Implementations
AFIPS FJCC Volume 39 (1972) pp 571-578

Needham R.M. and R.D.H. Walker (1974)
Protection and Process Management in the CAP Computer
International Workshop on 'Protection in Operating Systems'
IRIA (August 1974)

Opderbeck H. and W.W. Chu (1974)
Performance of the Page Fault Frequency Replacement Algorithm in a Multiprogramming Environment
Information Processing 74, North-Holland (1974) pp 235-241

Organick E.I. (1973)
Computer System Organisation: The B5700/B6700 Series
Academic Press, New York (1973)

Parnas D.L. and W.R. Price (1973)
The Design of the Virtual Memory Aspects of a Virtual Machine
Proc. of the ACM SIGARCH-SIGOPS Workshop on Computer Systems
Harvard (March 1973) pp 184-190

Price W.R. (1973)
Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems
Ph.D. Thesis, Carnegie-Mellon University (June 1973)

Randell B. (1969)
A Note on Storage Fragmentation and Program Segmentation
CACM 12,7 (July 1969) pp 365-372

Randell B. (1975)
System Structure for Software Fault Tolerance
Proc. of International Conference on Reliable Software
Los Angeles (April 1975)

Randell B. and L.J. Russell (1964)
Algol 60 Implementation
Academic Press, London, 1964

Satterthwaite E. (1972)
Debugging Tools for High Level Languages
Software Practice and Experience 2,3 (1972) pp 197-217

Sattley K. (1961)
Allocation of Storage for Arrays in Algol 60
CACM 4,1 (January 1961) pp 60-64

Simon H.A. (1962)
The Architecture of Complexity
In Proc. Am. Phil. Soc. 106 (1962) pp 467-482
(Reprinted in The Sciences of the Artificial, MIT Press, Cambridge, Massachusetts 1969)

Vanderbilt D.H. (1969)
Controlled Information Sharing in a Computer Utility
MAC TR-67, MIT, Cambridge, Massachusetts (October 1969)

Walker R.D.H. (1973)
The Structure of a Well-Protected Computer
Ph.D. Thesis, Cambridge University (December 1973)

Wang A. (1974)
Generalised Types in High Level Programming Languages
Ph.D. Thesis, University of Oslo (January 1974)

Wichmann B.A. (1969)
A Comparison of Algol 60 Execution Speeds
CCU Report No 3, National Physical Laboratory (January 1969)

Wichmann B.A. (1970)
Some Statistics from Algol Programs
CCU Report No 11, National Physical Laboratory (August 1970)

Wichmann B.A. (1973)
Algol 60 Compilation and Assessment
Academic Press, London, 1973

Wijngaarden A. et al (1974)
Revised Report on the Algorithmic Language Algol 68
Supplement to Algol Bulletin No 36
(Also: Technical Report TR74-5 (March 1974) Department of Computing Science, University of Alberta)

Wirth N. and C.A.R. Hoare (1966)
A Contribution to the Development of Algol
CACM 9,6 (June 1966) pp 413-431

Wortman D.B. (1972)
A Study of Language Directed Computer Design
Technical Report CSRG-20, University of Toronto (December 1972)

Wulf W.A. (1974a)
Alphard: Towards a Language to Support Structured Programs
Technical Report, Department of Computer Science,
Carnegie-Mellon University, (1974)

Wulf W.A. et al (1974b)
Hydra: The Kernel of a Multiprocessor Operating System
CACM 17,6 (June 1974) pp 337-345

Wulf W.A. and M. Shaw (1973)
Global Variable Considered Harmful
SIGPLAN Notices 8,2 (February 1973)

Wyeth D. (1973)
Estimates for the Size of the Recursive Cache
SRM/71, Computing Laboratory, University of Newcastle upon Tyne
(December 1973)