INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

REDUNDANCY AND ROBUSTNESS IN MEMORY PROTECTION

Butler W. LAMPSON

Xerox Research Center
Palo Alto, California, USA

Proc. IFIP Cong., North-Holland, 1974, pp 128-132

(INVITED PAPER)

The control of access to memory is a major problem in the design of protection mechanisms, especially for systems which allow files to be mapped into addressable memory. We use an abstract model of memory addressing to explain the space of possible organizations and examine their implications for protection. Analysis of the access paths to data allows some conclusions to be drawn about the amount of redundancy in a memory addressing scheme, and the possible consequences of an error for the integrity of the protection system.

1. Introduction

The topic of this paper is memory protection, that is, the intersection of memory architecture and protection. Since each of these subjects is both large and strongly connected, the intersection operation leaves quite a few loose ends. We will try to tie them up as best we can, and to point out places where the surgery appears to be fatal.

It is well to bear in mind that memory protection was born in the realm of the engineers, and has generally remained there. This history has a great effect on the way we think about the subject, one which this paper tries to counteract by an emphasis on abstract structure and on the unity of the whole subject of protection.

We will start with a brief survey of the parent subjects. Both depend on the concept of a protection context or *domain*. By this term we mean an environment within which a program can be executed, and which defines the actions that program is allowed to perform [4]. Unfortunately, this definition is not extraordinarily precise. We could sharpen it by saying that a domain *is* the actions which it authorizes, but at the cost of changing the meaning. We want a domain to be able to acquire and give up power while still maintaining its identity, just as a company can buy and sell property, or even change its line of business, and still be the same company.

In any particular system, of course, it is possible to point to entities whose existence is supported by that system, and identify them as domains. Thus we can define the term quite precisely for that system, but in a way which does not help us toward an abstract definition. Attempts to give a precise definition of the term "process" run into similar difficulties, and are usually resolved in the same way: by accepting the concept as a primitive one and explaining its meaning in terms of the operations in which it participates.

2. A model for protection

Since a domain is by definition the basic unit of protection, we can confine ourselves to the correct handling of interactions between domains, which we will describe in terms of messages sent by one domain and received by another. There are two issues:
. communication – how do messages get from one domain to another without being intercepted or altered;
. authentication – how does a domain know when to believe a message.

Communication has to stop somewhere if any work is to get done, so we must also equip each domain with *chattels* which it *owns* and can access directly. If a domain wants to affect anything which is not one of its chattels, it will have to *persuade* the owner to go along. Sharing of chattels is not allowed. This may seem too austere, since it forces us to describe a memory fetch from a shared object as an exchange of (at least two) messages, but it saves a lot of trouble by providing a single, consistent framework within which to talk about protection. Any form of communication can be clearly described as a sequence of messages, and this viewpoint exposes the logical structure of the communication, without saying anything about the cost. Specialized kinds of message exchange (such as a memory reference) can be implemented very cheaply using well-known techniques, and it is even possible to do this without sacrificing generality, if the specialized implementation allows for a trap to a more general environment when things get complicated.

2.1 *Why protection?*

A few words on the purpose of protection in general may also be helpful, since we will later be considering the purpose of memory protection in particular. Protection is a defense against some class of dangers or *threats*. The nature of a threat will depend to a great extent on its cause, which may be *accident* or *malice*. A threat may
. *expose* data (i.e. improperly allow it to be read);
. *damage* data (i.e. improperly allow it to be modified).

An accident may be a hardware failure, a program bug, or a mistake made by a user. We fear damage from an accident; exposure is possible as well, but is not likely to do any harm unless malice is also present. Protection against accidents tries to
. limit the damage, so that as much work as possible may proceed unaffected;
. detect it quickly, so that the cause of the accident can be better localized and hence more quickly found.

Malice is purposeful, and hence must be expected to seek out weakness. Furthermore, we fear exposure (espionage) more than damage (sabotage), and unobtrusive modification more than outright destruction. Unfortunately, the threats which we fear more are also more difficult to detect after the fact. For two reasons, then, defense against malice places more demands on the protection system than does defense against accident.

These observations suggest that it may be profitable to distinguish between *absolute* and *defensive* protection. An absolute system purports to guarantee that no matter what program is running inside a domain, it will be unable to break the protection barriers imposed by that domain. This guarantee will depend on the correct operation of various components, both hardware and software, with which the program can interact. In addition, of course, the guarantee is only as good as the definition of the domain; if a mistake was made in setting it up, the intended result will not be obtained. Absolute systems are seductive, because of the warm feeling of confidence and security which they engender in their

of the world, and it is therefore dangerous to take them too seriously. On the other hand, if the goal is to protect against a malicious, competent and determined intruder, an absolute approach is necessary in spite of its difficulties.

A defensive system, by contrast, tries to make it unlikely, rather than impossible, that bad things will happen (where the use of the word "unlikely" implies that it is primarily accident, rather than malice, that we are defending against). For example, many systems (including several which the author helped to design) provide debugging facilities which are protected from the programs being debugged, but do not have any way of preventing an undebugged program from deleting all its user's files. Such systems have defensive rather than absolute protection against bugs in a user's own programs. These bugs, of course, are accidental threats, but a library routine, compiler or utility program might well present a malicious threat of the same kind [6]. The point of all this is that absolute protection is very hard to come by, and in many case the defensive kind is more appropriate.

3. A model for memory referencing

A memory architecture has three parts, which we list in order of distance from the executing program:
. addressing - what is the form of a memory address, and how can a program make one;
. protection - when is a given operation on a given address legal;
. mapping - how is the address made by a program converted into a reference to some physical storage medium. In general, of course, this is a multi-step process.

In this section we develop a way of talking about the wide variety of memory protection schemes which have been described and, in some cases, even implemented. Since our interest is in protection, we have ignored distinctions which may be very important for the efficient encoding of programs or the implementation of mapping.

The most basic function of a memory protection system is the isolation of domains which are represented on hardware in such a way that they share the physical access paths to memory. If each domain had its own processor and memory, and the domains were interconnected by wires carrying messages, memory protection would not be essential (which is not to say that we might not want it anyway, since we might choose to simulate a memory-sharing system on such hardware). A protection system which simply simulates complete memory isolation is not very interesting from our point of view, however, even though it may be extremely robust, and even though it may be the best approach.

To maintain consistency with our view of protection in general, we will describe the interaction of a domain D with its memory in terms of messages passing between D and another domain MD which is responsible for the memory. Messages from D to MD have the form fetch(address) or store(address, value), and messages in the other direction are values returned in response to fetches. Both addresses and values have a rigidly defined format. MD may also accept other *configuration* messages, either from D or from other domains, which will affect its subsequent handling of fetch and store messages.

We define the *memory environment* of D as a reentrant tree with arcs labeled by strings or integers. An address has the form (item, link), where an item specifies a node in the tree, and a link is a string or integer which is usually the name of an arc (see figure 1a). If item i has an arc labeled b, we write i.b for the node at the other end of b. To process the message fetch(i, a), for example, MD takes i.fetch as a function which is applied to (i, a). This schema allows different functions to be implemented on the same data, as in figure 1b. It can also handle situations in which a memory reference is converted into an arbitrary procedure call, as can happen in Multics, for example [10].

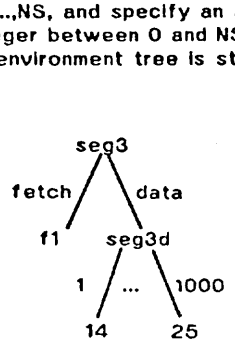
Every domain owns a single tree node called its *access point*. An item is specified by a link which is interpreted as the name of an arc leaving the access point. Thus in Multics the access point is the descriptor segment, and items are specified by integers called segment numbers. In the Plessey 250 system [2], on the other hand, an item is one of the eight capability registers.

There are a number of reasons for adopting such a stylized form of communication:
. efficient hardware implementation is possible for almost all messages;
. standard conventions for accessing data are necessary in any practical programming system;
. MD can implement facilities which allow several domains D1, D2, ... to share access to common data. Although the same effect can in principle be achieved by interchanging messages among the Di, this is often inconvenient in practice. Consider, for example, the global variables of Algol 60 or the pointers of most system programming languages;
. generality need not be lost, since MD can perform arbitrary computations (via traps to software), and can call on other domains to interpret the fetch and store messages for certain addresses.

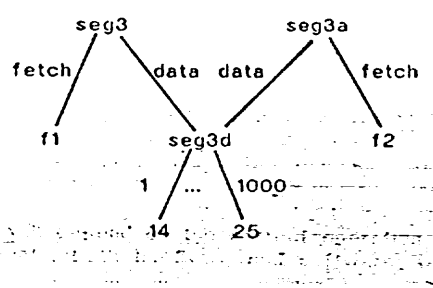
4. Power, convenience, and precision

The items which a domain D can specify determine the elements of its environment which it can access *directly*, i.e. with a single fetch or store. We will call these elements the *direct scope* of D; in general it will vary during execution as changes occur in the set of items which D can specify with a memory address. In particular, if some of the elements D can fetch are themselves items, the direct scope can change. The union of all possible direct scopes we will call the *scope* of D. The power of a domain, i.e. the things it can do if it tries hard, is determined by its scope.

If we take an absolute view of protection, there is no reason to distinguish elements of the scope by the access path which must be followed to reach them, since D can get any element of the scope into the direct scope at any time by simply executing a few instructions. In fact, we might as well simply number the items in the scope 1,2,...,NS, and specify an address as (n, link) where n is an integer between 0 and NS. Even from this point of view, the environment tree is still a useful description of



(a) a 1000-word segment



(b) a segment with two access functions

Figure 1: A simple memory environment

the structure of D's memory, which is important for actually writing programs and for concise description of how parts of the memory can be shared with other domains. And of course a defensive protection system would attach value to the fact that certain elements can be accessed directly, while others cannot be referenced except by the execution of a carefully chosen sequence of instructions.

4.1 *Continuity of protection*

Unfortunately, things are not really as simple as the preceding discussion suggests, since our model allows arbitrary changes in the behavior of MD, and therefore in the environment, to occur as a result of interactions between D and other domains. If the definitions we have given are taken literally, any element which could appear in the environment as a result of such changes is in the scope. This view is logically consistent, but it is disastrous for our attempt to separate the treatment of memory protection from the general topic of protection. The example of systems like Multics, which allow a domain to obtain access to a file as a result of arbitrarily complex interactions with other domains, and then to access the file as a segment, show that this difficulty is a real one.

There are really two aspects to this problem. First, we observe that in Multics, for example, it is the path name of a file and not the segment number that must be used as the item specifier. Second, the contents of the scope cannot be defined in any simple way, since it may change as a result of arbitrary computations performed by other domains which are not part of the memory system. We are forced to the conclusion that in a system which allows the memory environment to be changed as freely as Multics or Tenex [8], it is not possible to usefully separate the memory protection from the rest of the protection system. This fact imposes a strong requirement on the memory system: it must be able to support the facilities provided by the overall system. For example, if it is possible to take away access rights to a file, and the file happens to be mapped into memory, then the memory protection system must be able to find out about the file's change in status and adjust the memory access accordingly.

4.2 *Efficient communication*

Aside from these global issues, memory protection may also have an important role to play in communication between two domains. If the messages which implement this communication contain only data values, then memory protection does not enter in, since the data can be stored in strictly local memory in both domains. If, however, they contain pointers instead, then it is the memory protection system which will have to determine the validity of those pointers. The advantages of communicating pointers are well known: results can be returned by modifying a data structure, and copying of large quantities of data can be avoided. The disadvantages are also familiar: it is difficult to control the use of pointers precisely, and to ensure that they are erased when they are no longer needed.

The special contribution which a memory protection system can make, then, is to
. allow the pointers which are communicated between domains to specify exactly the kind of access actually required (e.g. access to a single word, record or array, rather than to an entire page or file);
. ensure that these pointers are either destroyed or invalidated at the right time;
. make all this cheap, since otherwise it will be better to communicate by value.

In the next two sections we shall examine some attempts to achieve these goals.

4.3 *Authentication*

A memory protection system can also contribute to a solution of the authentication problem. The basic concept needed to deal with this problem is the *trademark* introduced by Morris [7]. The purpose of a trademark is to authenticate the contents of an object; its usefulness depends on restricting the ability to affix it, or, to put it another way, on preventing forgery of trademarks. The manipulation of trademarked data can be aided by making it possible to tag a data record so that it can be copied and read, but not modified. The authentication of a trademark requires another kind of tag which can only be affixed by a system-provided *registrar*.

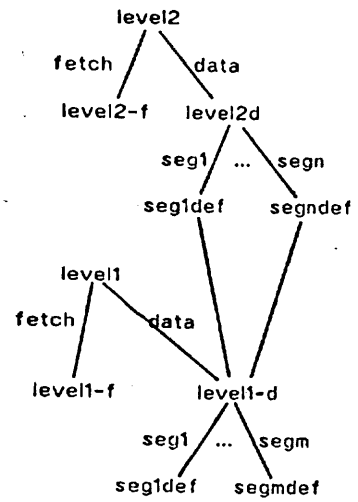
In terms of the memory addressing model, both kinds of tag correspond to special nodes through which access to the tagged data is constrained to pass, and which have suitably defined fetch and store sons. Unfortunately, no existing system provides anything at all close to this.

5. Kinds of memory protection

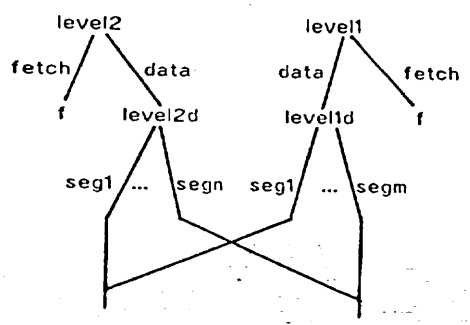
We are now in a position to flesh out the model with some examples which will serve to illustrate the range of possibilities, and perhaps to motivate some of the characteristics of the model.

The simplest memory protection system is one which enforces total isolation. The virtual machine systems are typical examples, of which the BBN PDP-1 system [1] is the earliest, and CP/67 [6] the most famous. In these systems the environment contains a single non-terminal item which we may call the memory, and its branches are named by the integers. At the end of each branch is a single word of the memory.

This description is actually an oversimplification of CP/67, which in its later versions allows the virtual machine construction to be iterated. There are two ways of describing this iteration, illustrated by figures 2a and 2b. In one case the hierarchical structure is traversed at each access; in the other, which corresponds closely to



(a) Full hierarchy



(b) Flattened hierarchy

Figure 2: A two-level memory hierarchy

the actual implementation, the environment describes a flattened version of the hierarchy, which must be reconstructed whenever changes are made at the lower levels.

The memory of an Algol program can be described by the model in an obvious way. Since Algol has no pointers and no facilities for manipulating variable bindings except on procedure calls, the scope of a domain is fixed. Whenever a procedure is called, of course, a new domain is created and its environment must be constructed according to familiar rules. Algol thus illustrates static sharing of memory.

For sound examples of dynamic sharing we must turn to operating systems again. Multics [10], Tenex [8] and other systems which allow named files to be mapped into the addressable memory of a domain illustrate what might be called *slow sharing*. The unit of sharing is quite large, and the mechanism for changing the environment is quite slow and clumsy, so that domains must be rather large. The memory protection is quite general, and whatever fundamental deficiencies it has are simply those of the overall protection system. Because of the clumsiness, however, it is of limited usefulness for frequent communication between domains.

In our addressing model, systems of this kind make heavy use of the reentrancy of the environment tree. A file or segment (or sometimes a page) is the smallest subdivision of memory from the point of view of the protection system, and the data of each file appears as a leaf in the tree. One access path to such a leaf is through the directory hierarchy, in which the arcs are labeled by strings and the access functions do fairly elaborate checking to ensure that the domain attempting the access is in fact entitled to it. This path cannot, of course, even be specified, much less followed directly by a memory reference.

There may, however, be additional access paths for files which are mapped into memory. Usually each domain has a unique access point (called a descriptor segment in Multics, a page table in Tenex). The arcs leaving the access point are labeled by integers (usually called segment numbers), and there is no access checking beyond that which is built into the tree. In other words, if an arc with a given label exists, access along that arc is automatically allowed.

There has been considerable interest recently in systems which realize the ideas of section 4.2, by allowing environments to be modified so conveniently that the decomposition of a computation into domains need not be constrained by the cost of transmitting arguments and switching protection contexts. The Cambridge Capability System [9] and Schroeder's proposal for dynamic validation of addresses [11] are two quite different solutions to this problem; both allow *fast sharing*. The Plessey 250 [2] is a commercial system which demonstrates still another approach.

All of these designs allow a domain to add structure to the environment tree without invoking any system software, and to pass an item (i.e. a node in the tree) as a message to another domain. They differ greatly in the form of the new structure, however, and consequently in their robustness.

6. Redundancy

Simple kinds of memory protection which enforce isolation between domains, and leave sharing to be handled by message communication, allow little room for mistakes in setting up the environment, and can use brute-force techniques for checking. Fabry [3] shows how capability and access-key implementations of protection can be cascaded in this case to obtain a very high degree of redundancy.

In more complicated situations it is hard to see how to obtain two independent descriptions of the protection rules, both of which are complete. Two techniques do exist, however, for obtaining partial redundancy: *static checking* and *hints*.

The design of the Multics system allows a static consistency check to be performed which verifies that the descriptor segments and page tables which define the memory environment are consistent with the file system information from which that environment is supposed to be constructed. In fact, the system permits any part of the environment to be invalidated and automatically reconstructed (if appropriate) from the file system, and this capability is in fact used, although the static check is never performed.

The BCC 500 [4] uses two-part pointers to specify the environment. One part of the pointer, called the *hint*, contains the physical address where the information is supposed to reside. The other part, called the *unique name*, contains a bit pattern which uniquely identifies the item and which must match a label stored with the item. The correspondence between unique name and label is checked whenever the hardware map is loaded and whenever a reference to secondary storage is made. This implementation illustrates a general technique for increasing robustness by adding redundancy, which can be applied whenever pointers are used. Alternatively, it can be viewed as a variation on access-key protection.

A "pure" capability system such as the Plessey 250 [2] has no redundancy except what is provided in the storage of the capabilities themselves (which happens to be a good deal). If any mistake is made in setting up a capability or in deciding where it should be put, there is no way of detecting the error during execution. It is not even possible to perform a static check for correctness of the environment defined by the capabilities, since every capability is as good as every other, regardless of how it was created or where it happens to be sitting. This approach allows a great deal of flexibility, but at some cost in robustness because of the lack of redundancy and consequent impossibility of doing any cross-checking.

An entirely different approach is taken by the Cambridge system [9]. It uses a hierarchy of descriptor segments, which is traced through whenever the hardware map must be loaded (see figure 2a). This structure provides multiple levels of compartmentation which are never collapsed into a single level. An error made in describing the environment which level *i* presents to level *i+1* cannot possibly affect any levels below *i*.

Somewhat similar in its underlying idea, although entirely different in flavor, is Schroeder's scheme for providing capabilities in the Multics environment [11]. He observes that if a hierarchical protection structure is carried to its logical limit, there is no need to treat *any* of the information which describes the environment as sacred to the basic protection system. Each level can be responsible for setting up the data structures which define the environment for the next higher level, and the memory protection system simply interprets these structures. Since anything level *i* says will be interpreted in the environment provided by level *i-1*, there is no need to constrain what level *i* can say in any way. Furthermore, the idea that a capability is a protected address is misleading. All the addresses generated by a domain are unprotected (i.e. integers or strings) in any system, and the protection is provided by the environment which interprets them.

7. Conclusion

We have tried to show how memory protection fits into the larger scheme of things. A good beginning is the observation that any kind of memory protection which does more than strictly isolate domains is introduced solely for efficiency, and not because it can provide any fundamentally new facility. From this viewpoint, we can think of capabilities, descriptor segments or whatever as clever encodings for certain specialized kinds of communication between domains. These encodings can be evaluated on the basis of the performance improvement they provide, the extent to which they make it possible to write cleaner programs (which in the end is a consequence of performance improvement), and their robustness in the face of hardware errors or misuse.

References

[1] S. Bolen et. al., A time-sharing debugging system for a small computer, *AFIPS Conference Proceedings* 23, 1963 SJCC, pp 51-58.

[2] D. M. England, Architectural features of System 250, *State of the Art Report* 14, Infotech, Maidenhead, Berks., 1972, pp 397-427.

[3] Robert S. Fabry, Dynamic verification of operating system decisions, *Communications of the ACM* 16, 11, November 1973, pp 659-668.

[4] Butler W. Lampson, Dynamic protection structures, *AFIPS Conference Proceedings* 35, 1969 FJCC, pp 27-38.

[5] Butler W. Lampson, A note on the confinement problem, *Communications of the ACM* 16, 10, October 1973, pp 613-615.

[6] P. A. Meyer and L. H. Seawright, A virtual machine time-sharing system, *IBM Systems Journal* 9, 3, July 1970.

[7] James H. Morris, Protection in programming Languages, *Communications of the ACM* 16, 1, January 1973, pp 15-21.

[8] Daniel L. Murphy, Storage organization and management in Tenex, *AFIPS Conference Proceedings* 41, 1972 FJCC, pp 23-32.

[9] Roger M. Needham, Protection systems and protection implementations, *AFIPS Conference Proceedings* 41, 1972 FJCC, pp 571-578.

[10] Elliott I. Organick, *The Multics System: An Examination of its Structure*, M.I.T. Press, Cambridge, Mass., 1972.

[11] Michael D. Schroeder, *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*, MAC TR-104, M.I.T., Cambridge, Mass., 1972.