PROJECT MAC

October 4, 1974

Computer Systems Research Division

Request for Comments No. 60

THE PROTECTION OF INFORMATION IN COMPUTER SYSTEMS

by J.H. Saltzer and M.D. Schroeder

The enclosed paper is a tutorial prepared by invitation for possible publication in the Proceedings of the IEEE, and for use in M.I.T. subject 6.033, "Information Systems". Any comments you might have would be welcomed. We are particularly interested in verifying the accuracy of the many historical attributions which appear throughout the paper, and adding any which have been omitted. Of course comments on the technical content are also of interest.

Address for correspondence:

Jerome H. Saltzer
M.I.T. Project MAC
Room NE43-505
545 Technology Square
Cambridge, Mass. 02139

Telephone (617) 253-6016

ARPANET mailbox: Saltzer (or SALTZER) at MIT-Multics (Host 44)

DRAFT

date October 3, 1974

THE PROTECTION OF INFORMATION IN COMPUTER SYSTEMS

by

Jerome H. Saltzer and Michael D. Schroeder

Abstract--This tutorial paper explores the mechanics of protecting information stored in computer systems. It begins by discussing desired functions, design principles, and examples of elementary protection and user authentication mechanisms. The second major part examines in depth the principles of modern descriptor-based protection architectures and the relation between capability systems and access-control-list systems. This discussion ends with a brief analysis of protected subsystems and protected objects. The last part of the paper reviews the state of the art and current research problems and also provides suggestions for further reading.

The authors are with Project MAC and the Computer Science section of the Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139

CONTENTS

FIGURES

TABLES

GLOSSARY

The following glossary provides definitions for several terms as used in this paper in the context of protecting information in computers.

access-control-list - A list of principals which are authorized to access some object.

authentication - Used to denote mechanisms and techniques that check the identity of the person making a request.

authorization - Granting a user permission to access certain information.

capability - In a computer system, an unforgeable ticket which, when presented by a process, can be taken as incontestable proof that the process has permission to access the object named in the ticket.

certification - Used to denote the checking of the accuracy, correctness, and completeness of a security or protection mechanism.

complete isolation - A protection system which separates users into groups between which no flow of information or control is possible.

descriptor - Used to denote a protected value which is (or leads to) the physical address of some protected object.

domain, protection - An abstraction used to refer to the set of objects which currently may be directly accessed by a process.

encipherment - The (usually) reversible scrambling of data according to a secret transformation key, so as to make it safe for transmission or storage in a physically unprotected environment.

hierarchical control - referring to ability to change authorization, a scheme in which each authorization is controlled by another authorization, resulting in a hierarchical tree of authorization.

list-oriented - Used to describe a protection system in which each protected object has a list of authorized accessors.

password - A secret character string which may be used to authenticate the claimed identity of a user.

permission - Used to indicate a particular form of access which is allowed, e.g., permission to read as contrasted with permission to write.

prescript - A rule which must be followed before access to an object is permitted. Its purpose is to provide an impediment to access so that abuse of the access is discouraged.

principal - The entity directly responsible for the activities of a process. A principal is a precise abstraction for the responsibility; principals may be associated with users on a one-to-one basis or in other more complicated arrangements.

privacy - The ability of an individual (or organization) to decide when and to whom personal (or organizational) information is to be released.

privileged state - A processor mode which allows use of instructions which can potentially modify protection information such as descriptor values.

propagation - When a user, having been granted access to some object, grants access to someone else.

protected object - A data structure whose existence is known to a user, but whose internal organization is not accessible, except by invoking the protected subsystem (q.v.) which manages it.

protected subsystem - A collection of procedures and data objects which are encapsulated in a way that the internal structure of a data object is accessible only to the procedures of the protected subsystem, and the procedures may be called only at designated entry points.

protection - 1. Security (q.v.) 2. Used more narrowly to denote mechanisms and techniques which control the access of executing programs to stored information.

protection group - A principal which may be used by several different individual users.

revocation - The action of a user taking previously authorized access away from some other user.

security - Used to denote mechanisms and techniques which control the use, modification, and release of information.

self-control - Referring to ability to change authorization, a scheme in which each authorization contains within it information on who may change it.

ticket-oriented - Used to describe a protection system in which each active accessor (e.g., a process or a procedure, as appropriate) maintains a list of unforgeable bit patterns, called tickets, one for each object it is authorized to access.

unforgeability - The inability to duplicate an object, such as a badge or key, or a register value, such as a descriptor.

user - Used imprecisely to refer to the person who assumes responsibility for the activities of a process.

THE PROTECTION OF INFORMATION IN COMPUTER SYSTEMS

This tutorial paper explores the protection of computer-stored information from unauthorized use or modification. It concentrates primarily on those architectural structures--whether hardware or software--that are necessary and useful to support information protection. The paper is organized in three main sections. The first part covers basic principles and should be understandable to any reader familiar with computers. The second part, on general descriptor-based protection systems, requires some familiarity with modern descriptor-based computer architecture. The third part contains some general observations about the state of the art and some unsolved research problems.

I: Basic Principles of Information Protection

Introduction

As computers become better understood and more economical, every day brings new applications. Many of these new applications involve both storing information and use by several individuals, often simultaneously. The key concern in this paper is multiple use. For those applications in which all users should not have identical authority, some scheme is needed to insure that the computer system implements the desired authority structure. For example, in an airline seat reservation system, a reservation agent might have authority to make reservations and to cancel reservations for people

whose names he can supply. A flight boarding agent might have the additional authority to print out the list of all passengers who hold reservations on the flights for which he is responsible. The airline might wish to withhold from the reservation agent the authority to print out a list of reservations, so as to make less likely his cancelling a reservation of some legitimate seat-holder in order to make room for a customer who has offered a bribe.

The airline example is one of protection of corporate information for corporate self-protection (or public interest, depending on one's view.) A different kind of example is an on-line warehouse inventory management system which generates reports about the current status of the inventory. These reports not only represent corporate information which must be protected from release outside the company, but also they may provide an indication of the quality of the job being done by the warehouse manager. In order to preserve his personal privacy, it may be appropriate to restrict the access to such reports, even within the company, to those who have a legitimate reason to be judging the quality of the warehouse manager's work.

Many other examples of systems requiring protection of information are encountered every day: credit bureau data banks, law enforcement information systems, time-sharing service bureaus, on-line medical information systems, and government social service data processing systems. These examples span a wide range of needs for organizational and personal privacy. All have the common element of controlled sharing of information among multiple users. All, therefore, require that some plan be made to insure that the computer system helps implement the correct authority structure. Of course, there are some applications in which no special provisions in the computer system are necessary.

It may be, for instance, that an externally administered code of ethics or a lack of knowledge about computers provides adequate protection of the stored information. However, although there exist situations in which the computer need provide no extra aids to assure protection of information, more commonly the computer system is called upon to enforce a desired authority structure.

> As suggested in the above examples, the term "privacy" denotes a socially defined goal of an individual (or organization) concerning when and to whom information about himself (or itself) is to be released.

This chapter will not be explicitly concerned with privacy*, but instead with the mechanisms used to help achieve it.

> The terms "protection" or "security" (the two terms being synonymous in ordinary usage**) denote techniques which control the use, modification, and release of information.

Some examples of security techniques which are sometimes applied to computer systems include:

- labeling of files with lists of authorized users
- verifying the identity of a proposed user by demanding a password
- shielding the computer to prevent interception and interpretation of electromagnetic radiation
- enciphering information sent over telephone lines

---

* A thorough and scholarly discussion of the concept of privacy may be found in the book Privacy and Freedom by Alan Westin (Atheneum, 1967). An interesting study of the impact of technology on privacy is The Assault on Privacy, by Arthur R. Miller (University of Michigan, 1971; Signet, 1972).

** In 1967, Willis Ware [AFIPS Conf. Proc. 30 (1967), 287-290] suggested that the term security be used for systems which handle classified defense information, and privacy for systems handling non-defense information. This suggestion has never really taken hold outside the defense security community, but literature originating within that community often uses Ware's definition.

- padlocking the room containing the computer
- controlling who is allowed to make changes to the computer system (including both hardware and software changes)
- using redundant circuits or programmed crosschecks which maintain correct operation in the face of hardware or software failures
- certifying that the hardware and software is actually implemented as intended

It is thus apparent that a wide range of considerations and techniques are pertinent to the engineering of security of information. Historically, the literature of computer systems has more narrowly defined the term "protection" to be just those techniques that control the access of executing processes to stored information. An example of a protection technique is the labeling of computer-stored files with lists of authorized users. Similarly, the term "authentication" is used for those techniques that check the identity of a remote human user of a computer system. An example of authentication is the demanding of a password. The next sections concentrate primarily on protection and authentication mechanisms, with only occasional reference to the other equally necessary security mechanisms.*

---

* Some authors have widened the scope of the term "protection" to additionally include mechanisms designed to prevent or limit the consequenses of accidental mistakes in programming or in applying programs. With the wider definition, even computer systems used by a single person might include "protection" mechanisms. The effect of this broader definition of "protection" would be to also include in our study, for example, interlocks which can be bypassed by the user, on the basis that the probability of accidental bypass can be made as small as desired by careful design. Such accident-reducing mechanisms are often essential, but one would be ill advised to apply one to a situation in which a systematic attack by another user is to be prevented. Therefore, we will insist on the narrower definition. Protection mechanisms are very useful in preventing mistakes, but mistake preventing mechanisms may not be helpful in providing protection.

One should, however, recognize that in concentrating on protection and authentication mechanisms one is taking a narrow view of the information security problem, and that there is a special reason for a narrow view to be dangerous. The functional objective of a secure system is to prevent all unauthorized information release, which is a negative kind of requirement. It is usually quite hard to prove that this negative requirement has actually been achieved, for one must demonstrate that no means of unauthorized information release exist. Thus, an expansive view of the problem is probably most appropriate, to help assure that no gaps appear in the strategy.*

Functional levels of information protection

Many different designs and mechanisms have been proposed, and sometimes implemented, for protecting information in computer systems. The significant differences among protection schemes usually lie in the different functional properties of the schemes, that is in the kinds of access control which can be expressed and enforced. It is convenient to divide protection schemes according to their goals, as follows:

0. Unprotected systems. Some systems have essentially no provision for preventing access by a determined user to every piece of information stored in the system. Although these systems are not directly of interest here, they are worth mentioning since, as of 1974, many of the most widely used commercially available batch data processing systems--for example, the Disk Operating System for the IBM System 370 [ref. **********] --fall into this category.**

---

* The broad view, encompassing all the considerations mentioned above and more, is taken in several current books, e.g., Security, Accuracy, and Privacy in Computer Systems, by James Martin (Prentice-Hall, 1973), and the chapter entitled " ********** " by James Anderson in Rubinoff's Advances in Computers, Vol. 12 (Academic Press, 1973).

** One can develop a spirited argument as to whether systems originally designed as unprotected, and later modified to implement some level of protection goals, should be reclassified or continue to be considered unprotected. The argument arises from skepticism that one can successfully change the fundamental design decisions involved. Most large-scale commercial batch processing systems fall into this questionable area.

1. All-or-nothing systems. These are systems which provide either complete isolation or complete sharing of each piece of information. If only complete isolation is provided, then each piece of information is accessible to only one user, and the user of such a system may just as well be using his own private computer, as far as protection is concerned. More commonly, such systems have public libraries which every user may access. In some cases the public library mechanisms may be extended to accept user contributions, but still on the basis that all users have equal access. Most of the first generation of commercial time-sharing systems provide a protection scheme with this level of function. Examples include the Dartmouth Time-Sharing System [ref. **********], IBM's system VM/370 [Meyer and Seawright, IBM Sys. J. 9 (1970), 199-218], and innumerable privately constructed time-sharing systems.

2. Controlled sharing. The next level of functional capability that is of interest and that introduces significantly more complex machinery is one which permits explicit control of who may access each data item stored in the system. For example, such a system might provide each file with a list of authorized users, and allow an owner to distinguish several common patterns of use, such as reading, writing, or executing the contents of the file as a program. Although conceptually a straightforward idea, actual implementation is surprisingly intricate, and only a few complete examples exist. These include M.I.T.'s Compatible Time-Sharing System [CTSS Programmers' Guide, (M.I.T. Press, 1965)], Digital Equipment Corporation's DECSystem/10 [ref. **********], System Development Corporation's ADEPT [Weissman, AFIPS Conf. Proc. 35 (1969), 119-133], and Bolt Beranek and Newman's TENEX [Bobrow, CACM 15, 3 (1972), 135-143].

3. User-specified sharing controls. For some applications, the user wishes to specify some form of restriction of access to a file that is not provided in the standard facilities for controlling sharing. For example, he may wish to allow access only on weekdays between 9:00 a.m. and 4:00 p.m. Possibly, he may wish to permit access to only the average value, or some other statistically aggregated summary, of the data in a file. Maybe he wishes to require that modification of a file should be done only if agreement of two users is obtained. For such cases, and a myriad of others which can be imagined, a general escape is to provide for user-defined protected objects and subsystems. A protected subsystem is a collection of programs and data with the property that direct access to the data (that is, the protected objects) is restricted to the programs of the subsystem. Access to the programs is limited to calling specified entry points. Thus, the programs of the subsystem exercise complete control over the use made of the data. By constructing a protected subsystem, a user can develop any programmable form of access control to the objects he creates. Permitting user-specified protected subsystems raises several difficult issues, which probably explains why only a few of the most advanced system designs have tried it. These include Honeywell's Multics [Corbató et al., AFIPS Conf. Proc. 40 (1972), 571-583], The University of California's CAL System [Sturgis, Ph.D. thesis, University of California at Berkeley, 1973], Bell Laboratories' Unix System [Thompson and Ritchie, CACM 17, 7 (1974)], and two systems currently under construction: the Cambridge Capability system of Cambridge University [Needham, AFIPS Conf. Proc. 41 (1972), 571-578, and the HYDRA System of Carnegie-Mellon University [Wulf, et al., CACM 17, 6 (1974), 337-345]. Exploring alternative mechanisms for implementing protected subsystems is a research topic of current interest.

4. Putting strings on information. One of the concerns of the previous three levels has been establishing conditions for the release of information to an executing program. The fourth level of functional capability is to somehow maintain some control over the use of the information even after it has been released. Such control is what is desired, for example, in releasing income information to a tax advisor: constraints are desired which prevent him from passing the information on to a firm which prepares mailing lists. The printed labels on classified military information which declare a document to be "TOP SECRET" are another example of a constraint on information after its release to a person authorized to receive it. One may not (without risking severe penalties) release such information to others, and the label serves as a notice of the restriction. Computer systems which implement such strings on information are rare and mechanisms are partial. For example, the ADEPT system keeps track of the classification level of all inputs used to create a file; all output data is automatically labeled with the highest classification encountered during execution.

In examining any particular design, some care should be taken to distinguish among:

- The protection intentions of the person responsible for the information.
- The mechanisms for enforcing those intentions which are external to the computer system, and
- The mechanisms for enforcing those intentions which are internal to the computer system.

Considering the four levels of functional capabilities, the person responsible for the information may have intentions which require any or all of the four levels. On the other hand, he may insist on computer-aided enforcement of only some of these intentions, and use external mechanisms such as contracts, ignorance, or barbed wire fences, for others. As indicated, this discussion is focused on the internal mechanisms.

Finally, there is a set of functional properties that cuts across all four levels of functional capability: the dynamics of use. This term refers to the provisions for initializing and changing the specification of who may access what. At all four levels it is relatively easy to envision (and design) systems that statically express a particular protection intent. Most of the complexity in protection systems is introduced by the need to change protection specifications dynamically, and for such changes to be requested by executing programs. Most of the existing systems differ primarily in their methods of handling the protection dynamics. To gain some insight about the kind of complexity introduced by program-directed changes to protection specifications, consider that to answer the question "can user A access file X?" one must examine not only the static specification of who may access file X, but also the specification of who may change the statement of file X's accessibility, and also who may change that specification, etc. Another example of an interesting problem of dynamics is what to do if the owner revokes access to a file while it is being used by a (previously) authorized user. The alternative of permitting the previously authorized user to continue access until he is "finished" with the information may not be acceptable if the owner has suddenly realized that the file contains sensitive data. On the other hand, immediate withdrawal of access may be disruptive to the previously authorized user; the disruption may effectively invade his privacy. It should be apparent that provisions for the dynamics of use are at least as important as those for static specification of protection intent.

Design principles and evaluation considerations

Security specialists (e.g., Anderson, in Advances in Computers 12) have found it useful to place potential security violations in three categories:

1. Unauthorized information release: an unauthorized person is able to read, and take advantage of, information stored in the computer. This category of concern sometimes extends to "traffic analysis", in which the intruder observes only the patterns of use of information and from those patterns can infer some information content.
2. Unauthorized information modification: an unauthorized person is able to make undetected changes in stored information, a kind of sabotage. Note that this kind of violation does not necessarily require that the intruder succeed in seeing the information he has changed.
3. Unauthorized denial of use: an intruder can prevent an authorized user from accessing or modifying information, even though the intruder may not be able to access or change the information, for example by disrupting a scheduling algorithm or causing a system "crash". Denial of use is another kind of sabotage.

The thing which most complicates the situation in a general-purpose, remote-accessed computer system is that the "intruder" in these definitions may be an otherwise legitimate user of the computer system.

The term "unauthorized" in the three categories listed above means that release, modification, or denial of use occurs contrary to the desire of the person responsible for the information, and possibly even contrary to the constraints supposedly enforced by the protection mechanisms of a system. Whatever the level of functionality provided, the usefulness of a set of protection mechanisms depends upon the ability of a system to prevent these three kinds of security violations. In practice, producing a system at any level of functionality (except perhaps level 0) that actually does prevent all such unauthorized acts has proven extremely difficult. Sophisticated users of most currently available systems are probably aware of at least one way to crash the system, thus denying other users authorized access to stored information. Penetration exercises involving a large number of different systems have shown that in most

existing systems a user can contruct a program which can obtain unauthorized access to information stored within the system. Even in systems designed and implemented with security as an important objective, localized design and implementation flaws have provided paths by which the access constraints intended by the system could be circumvented. This phenomenon is a direct result of the negative quality of the requirement to prevent all unauthorized actions. Design and construction techniques which systematically exclude such flaws are the topic of much current research activity, but no complete method applicable to the construction of large, general-purpose systems exist yet.

In the absence of such methodical techniques, experience has provided some useful design principles which can guide the design and contribute to an implementation without security flaws. Here are eight examples of design principles which particularly apply to protection mechanisms:

1. Economy of mechanism. Keep the design as simple and small as possible. This is a well-known principle applying to any aspect of system design, but it bears repeating with respect to protection mechanisms, since there is a special problem: design and implementation errors which result in unwanted access paths will not be immediately noticed during normal use, (since normal use usually does not include attempts to exercise improper access paths). Therefore, techniques such as complete, line-by-line inspecition of software and hardware which implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential.

2. Failsafe defaults. Base protection mechanisms on permission rather than exclusion. This principle, suggested by E. L. Glaser in 1965, means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, seems to present a wrong psychological base for secure system design. A conservative design must be based on arguments on why objects should be accessible, rather than on why they should not. In a large system some objects will be inadequately considered, so a default of lack of permission is more fail-safe. Similarly, a design or implementation mistake in a mechanism which gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. A design or implementation mistake in a mechanism which explicitly excludes access tends to fail by not excluding access, a failure which may go unnoticed. This principle applies both to the appearance of the protection mechanism to its users and also to the underlying implementation.

3. Complete mediation. Every access to every object must be checked for authority [Anderson, Air Force Technical Report ESD-TR-73-51 (1973)]. This principle, when systematically applied, is the primary underpinning of the protection system. It forces a system-wide view of access control which in addition to normal operation includes initialization, recovery, shutdown, and maintenance. It also implies that a foolproof method of identifying the source of every request must be devised. In a system designed to operate continuously, this principle requires that when access decisions are remembered for future use, careful consideration be given to how changes in authority are propagated to include the remembered decisions.

4. Open design. The design should not be secret [Baran, Rand report RM 3265-PR (1964), Vol. 9]. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords. This strong decoupling between protection mechanisms and protection keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards. In addition, any skeptical user may be allowed to convince himself that the system he is about to use is adequate for his purpose.

5. Separation of privilege. Where feasible, a protection mechanism which requires two keys to unlock it is more robust and flexible than one which allows access to the presenter of only a single key. The relevance of this observation to computer systems was pointed out by R. Needham in 1973. The reason is that, once the mechanism is locked, the two keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. From then on, no single accident, deception, or breach of trust is sufficient to compromise the protected information. This principle is often used in bank safe-deposit boxes, and is also at work in the defense system which requires that firing a nuclear weapon can be done only if two different people both give the correct command. In a computer system, separated keys are applicable to any situation in which two or more conditions must be met before access should be permitted.

6. Least privilege. (Suggested by D. Edwards) Every program and every user of the system should operate using the least amount of privilege necessary to complete the job. The primary purpose of this principle is to limit the damage that can result from an accident or error. Another purpose is to reduce the number of potential interactions among privileged programs to the minimum necessary to operate correctly, so that one may develop

confidence that unintentional, unwanted, or improper uses of privilege do not occur. Thus, if a question related to misuse of a privilege occurs, the number of programs which must be audited is minimized. Put another way, if one has a mechanism available which can provide "firewalls", the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of "need-to-know" is an example of this principle.

7. Least common mechanism. (Suggested by G. Popek.) Minimize the amount of mechanism that is common to more than one user and that every user has no choice but to depend on. The reason is that every shared mechanism (especially a shared data base) represents a potential information-passing path between users, and must be designed with great care to be sure it does not unintentionally compromise security. Further, any mechanism which serves all users must be certified to the satisfaction of every user, a job presumably harder than satisfying only one or a few users. Thus, for example, given the choice of implementing a new function as a supervisor procedure shared by all users or as a library procedure which can be handled as though it were the user's own, the latter course would be preferable. Then, if one or a few users are not satisfied with the level of certification of the function, they can provide a substitute or else not use it at all. Either way, they cannot be harmed by a mistake in it.

8. Psychological acceptability. It is essential that the human interface be designed for naturalness, ease of use, and simplicity, so that users will routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals match the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors.

As is apparent, these principles do not represent absolute rules, but instead serve best as warnings: if some part of a design violates a principle, the violation is a symptom of potential trouble, and the design should be carefully reviewed to be sure that the trouble has been accounted for, or is unimportant.

In a similar vein, two evaluation considerations suggested by Turn and Shapiro [RAND report P-4871 (1972)] can help in the review of a proposed design. These are:

1. Compare the cost of the protection mechanism with the value of the information being protected. Although precise evaluation of cost and of information value is very hard, it may be sufficient to identify only the order of magnitude of each to decide whether a protection mechanism is reasonable or extravagant.

2. Compare the cost of circumventing the mechanism with the resources of a potential attacker. The cost of circumventing, commonly known as the "work factor", may in some cases be easily calculated. For example, the number of experiments needed to try all possible four-letter alphabetic passwords is $26^4 = 456,976$. If the potential attacker must enter each experimental password at a teletype, one might consider a four-letter password to be adequate. On the other hand, if the attacker could use a large computer capable of trying a million passwords per second, as might be the case where industrial espionage or military security is being considered, a four-letter password would be a minor barrier for a potential intruder.

Finally, we may note that mechanisms that reliably record that a compromise of information has occurred are very valuable, and can sometimes be used in place of more elaborate mechanisms which prevent loss. An unbreakable padlock on a flimsy file cabinet is an example of such a mechanism. Although access

is not hard to obtain, with proper design the cabinet will inevitably be damaged and the loss detected by the next legitimate user. For another example, many computer systems record the date and time of most recent use of each file. This record may be useful to discover unauthorized use.

The essentials of information protection

For purposes of discussing protection, the information stored in a computer system is not a single monolithic object. The information is divided into mutually exclusive partitions, as specified by its various creators, with the property that each partition contains a collection of information, all of which is intended to be protected uniformly. The uniformity of protection is the same kind of uniformity that applies to all of the diamonds stored in the same vault: any person who has a copy of the combination can obtain any of the diamonds. Thus the contents of each partition are the fundamental objects to be protected.

It may be convenient for the protection partitions to exactly coincide with those of some system-provided naming scheme. Thus if the system provides symbolically named files for the storage of information, the information in a single file might be protected uniformly, but perhaps differently from information in another file. Coincidence between the standard system naming method and protection is not, however, a requirement. One could devise a system in which the separately protected objects are physical storage devices, each of which may contain many files. Alternatively, it may be appropriate for the separately protected objects to be smaller than a file.

Conceptually, then, it is necessary to build an impenetrable wall around each distinct object which warrants separate protection, provide a door in the wall through which access may be obtained, and provide a guard at the door to control its use. Control of use, however, requires that the guard have some way of knowing which users are authorized to have access, and that each user

have some reliable way of identifying himself to the guard. This authentication is usually implemented by having the guard demand a match between something he possesses and something the prospective user possesses. Thus, a primitive protection scheme has two components: a protection mechanism (a wall with a door and a guard) and an authentication mechanism (a way of identifying authorized users). Protection of information may be handled at several places within a computer system, but every example includes some adaptation of these two components.

Before extending these abstractions, we shall pause to consider two concrete examples, the multiplexing of a simple computer system among several users, and the authentication of a user's claimed identity. These initial examples are complete isolation systems--no sharing of information can happen. We will need later to return to extend our model of guards, walls, and protected objects, in order to discuss sharing.

An isolated virtual machine

A typical simple computer consists of a processor, a linearly addressed memory system, and some collection of input/output streams associated with the processor. It is relatively easy to use a single computer to simulate several machines, each of which is completely unaware of the existence of the others, except for the fact that time passes more slowly than usual. Such a simulation is of interest, since during the intervals when one of the simulated (commonly called virtual) machines is waiting for an input or output operation to complete, another virtual machine may be able to progress at its normal rate. Thus, a single processor may be able to take the place of several. Such a scheme is the essence of a multiprogramming system.

To allow each machine to be unaware of the existence of the others, it is essential that some isolation mechanism be provided, such as a special hardware register, called a descriptor register, as in figure 6-1. In this figure, all memory references by the processor are checked by an extra piece of hardware which is interposed in the path to the memory. The descriptor register serves to control exactly which part of memory is to be accessible. The descriptor register contains two components: a base value and a bound value. The base is the lowest numbered address which the program may use, and the bound is the number of locations beyond the base which may be used.* We will call the value in the descriptor register a descriptor, as it describes an object (in this case, one program) stored in memory. As we go on, we will embellish the concept of a descriptor: it is central to the implementation of protection and of sharing of information.

We can associate this example with our abstractions. The information being protected is the distinct programs of figure 6-1. The impenetrable wall, with a door, is provided by the descriptor register. The guard is represented by the extra piece of hardware which enforces the descriptor restriction, but we note that his authentication scheme is degenerate: the process controlling the processor has full access to everything in the base-bound range.

---

* In most implementations, addresses are also relocated by adding to them the value of the base. This relocation implies that for an address A to be legal, it must lie in the range ($0 \leq A < \text{bound}$). The concept of a base and bound register with relocation was used in the M.I.T. Compatible Time-Sharing System in 1960 [Corbató et al., Proc. WJCC (1962), 335-344]. The concept of a descriptor as used in the B5000 computer was anonymously documented in a 1961 Burroughs publication, The Descriptor.

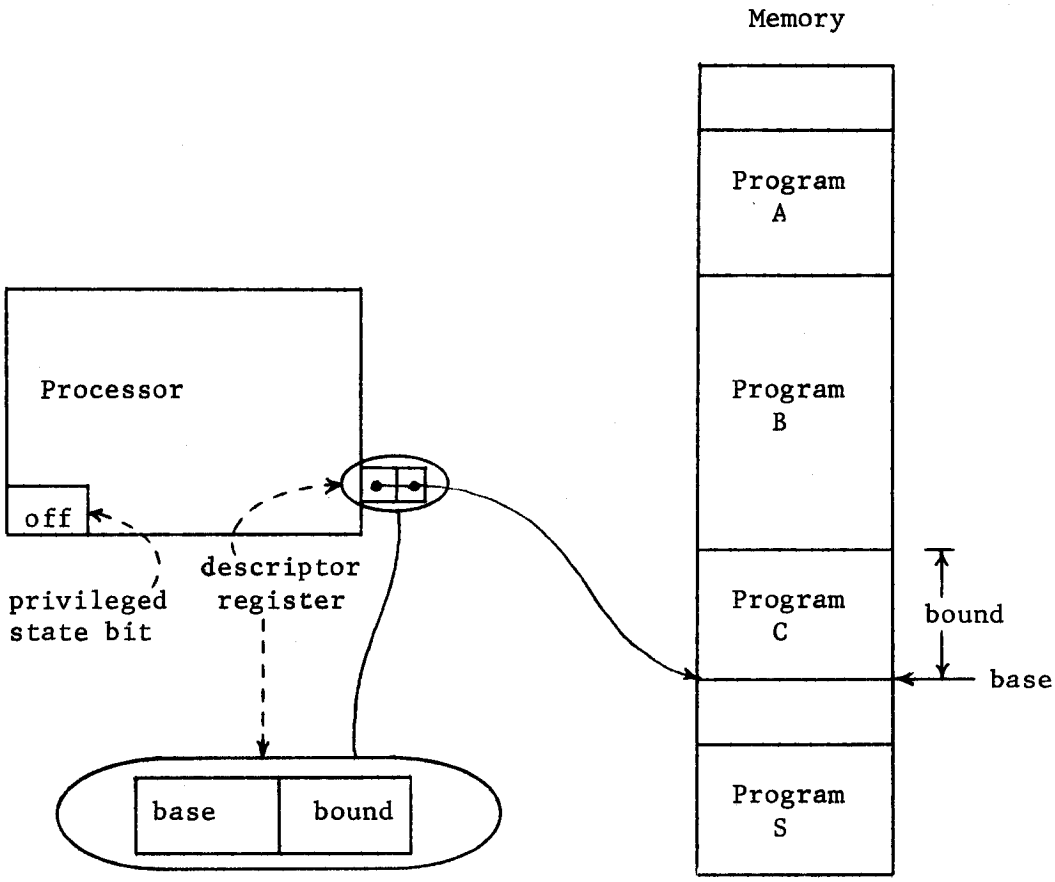


Figure 6-1 -- Use of a descriptor register to simulate multiple virtual machines. Program C is in control of the processor. The privileged state bit has value off, indicating that Program C is a user program. When Program S is running, the privileged state bit has value on. In this (and later) figures, lower addresses are nearer the bottom of the figure.

At least part of the degeneracy stems from our not having provided for the dynamics of a complete protection scheme: we have not discussed who loads the descriptor register. If any running program could load it with any arbitrary value, there would be no protection. Therefore, the instruction which loads the descriptor register with a new descriptor must have some special controls--either on the values it will load, or on who may use it. The easiest form of control is on who may use it, and an historically common scheme is to introduce an additional bit in the processor state, called the privileged state bit*. All attempts to load the descriptor register are checked against the value of the privileged state bit; the privileged state bit must be on for the register to be loaded. One program (say program S in figure 6-1) runs with the privileged state bit on, and controls the simulation of the virtual machines for the other programs. But now, we have an authentication scheme in the privileged state bit; ability to authorize use of memory (e.g., ability to load the descriptor register) requires presentation of a privileged state bit with value on. All that is needed to make the scheme complete is to insure that the privileged state bit cannot be changed by the user except, perhaps, by an instruction which simultaneously transfers control to the supervisor at a planned entry location. (In most implementations, the descriptor register is not used in the privileged state).

One might expect the supervisor to maintain a table of values of descriptors, one for each virtual machine; when the privileged state bit is off, the index in this table of the program currently in control identifies exactly which program--and thus which virtual machine--is responsible for the activity of the processor. Note that for protection to be complete, it

---

* Also called the master/slave bit, or supervisor/user bit.

must not be possible for a program running under the control of one of the virtual machines to be able to make arbitrary changes either to the descriptor currently in the descriptor register, or to the values in the table of descriptors. If we suppose the table to be stored inside the supervisor program, it will be inaccessible to the virtual machines.

With an appropriately sophisticated and careful supervisor program, we now have an example of a "complete isolation" type of system. Similarly isolated permanent storage can be added to such a system by attaching some long-term storage device (e.g., magnetic disk) and developing a similar descriptor scheme for its use. Since long-term storage is accessed less frequently than primary memory, it is common that its descriptor scheme be interpreted by supervisor programs rather than built into the hardware, but the principle is the same.* Long term storage does, however, force us to face one further issue. Suppose that the virtual machine communicates with its user through a typewriter terminal. If a new user approaches a previously unused terminal and requests to use a virtual machine, which virtual machine (and therefore which set of long term stored information) should he be allowed to use? We may solve this problem outside the system, by having the supervisor permanently associate a single virtual machine with its long term storage area with a single typewriter terminal. Then, for example, padlocks can control access to the typewriter terminal. If, on the other hand, a more flexible system is desired, the supervisor program must be prepared to dynamically associate any terminal with any virtual machine.

---

* For an example, see IBM System VM/370 [Meyer and Seawright, IBM Sys. J. 9, (1970) 199-218], which provides virtual IBM 370 computer systems, complete with private storage devices.

Authentication mechanisms

Our second example is of a distinctly different kind of protection system: a system which verifies a user's claim as to who he is. The mechanics of this protection system differ from those of the virtual computer mainly because not all of the components of the system are under uniform physical control. In particular, the user himself and the communication system connecting his terminal to the computer are components to be viewed with suspicion. Conversely, the user needs to verify that he is in communication with the expected computer system. Such systems follow our abstract model of a guard who demands a match between something he possesses and something the user possesses. However, because of the lack of physical control of the user and the communication system, the security of the system then depends on either the secrecy or the unforgeability of this identification.

In time-sharing systems the most common scheme depends on secrecy: the user begins by typing the name of the person he claims to be, and then the system demands that the user type a secret password, presumably known only to that person.

There are, of course, many possible elaborations and embellishments of this basic strategy. In cases where the typing of the password may be observed, passwords may be good for only one use, and the user carries a list of passwords, crossing each one off the list as he uses it. Passwords may have an expiration date, or usage count, to limit the length of usefulness of a compromised one.

The list of acceptable passwords is a piece of information which must be carefully guarded by the system. In some systems, all passwords are passed

through a hard-to-invert transformation* before being stored, an idea suggested by M. Wilkes. When the user types his password, the system transforms it also, and compares the transformed versions. Since the transform is supposed to be hard to invert, even if the stored version of a password is compromised, it may be very difficult to determine what original password is involved. It should be noted, however, that a good measure of "hardness of inversion" is difficult to come by. The attacker of such a system does not need to discern the general inversion, only the particular one applying to some transformed password he has available.

Passwords as a general technique have some notorious defects. The most often mentioned defect lies in choice of password--if a person chooses his own password, he may choose something easily guessed by someone else who knows his habits. For this reason, some systems have programs which generate random sequences of letters for use as passwords, and may even require that all passwords be system-generated and frequently changed. On the other hand, frequently changed random sequences of letters are hard to memorize, so such systems tend to cause users to make written copies of their passwords, inviting compromise.

A second significant defect is that the password must be exposed in order to use it. In systems where the terminal is distant from the computer, the password must be sent through some communication system, during which passage a wiretapper may be able to intercept it.

An alternative approach to secrecy is unforgeability. The user is given a key, or magnetically striped plastic card, or some other unique and difficult-to-duplicate object. The terminal has an input device which examines the object and transmits its unique identifying code to the computer system, which treats

---

* For example, the password is used as the parameter in a high-order polynomial calculated in modulo arithmetic [Purdy, CACM 17, 8 (1974), 442-445]. Evans, Kantrowitz and Weiss [CACM 17, 8 (1974), 437-442] have suggested an alternative, more complex scheme based on multiple functions.

the code as a password which need not be kept secret. Proposals have been made for fingerprint readers [**********] and dynamic signature readers [**********], in order to increase the effort required for forgery.

The primary weakness of such schemes is that the hard-to-duplicate object, after being examined by the specialized input device, is reduced to a stream of bits to be transmitted to the computer. Unless the terminal, its object reader, and its communication lines to the computer are physically secured against tampering, it is relatively easy for an intruder to modify the terminal to transmit any sequence of bits he chooses. It may therefore be necessary to make the acceptable bit sequences a secret after all. On the other hand, the scheme is convenient, resists casual misuse, and provides a simple form of accountability through the physical objects used as keys.

A problem common to both the password and the unforgeable object approach is that they are "one-way" authentication schemes. They authenticate the user to the computer system, but not vice-versa. An easy way for an intruder to actively penetrate a password system, for example, is to intercept all communications to and from the terminal and direct them to another computer which is under the interceptor's control. This computer can be programmed to "masquerade", that is, to act just like the system the caller intended to use, up to the point of requesting him to type his password. After receiving the password, the masquerader gracefully terminates the communication with some unsurprising error message, and the caller may be unaware that his password has been stolen.

A more powerful authentication technique is sometimes used to protect against masquerading. Suppose that a remote terminal is equipped with enciphering circuitry, such as the LUCIFER system [Smith, et al., Proc. ACM 25 Conf. (1972), 282-298], that scrambles all signals from that terminal. Such devices normally are designed so that the exact encipherment is determined by the

value of a key, known as the encrypting or transformation key. For example, the transformation key may consist of a sequence of 1000 binary digits read from a magnetically striped plastic card. In order for a recipient of such an enciphered signal to comprehend it he must have either a deciphering circuit which is primed with an exact copy of the transformation key, or else he must cryptanalyze the scrambled stream to try to discover the key. The strategy of encipherment/decipherment is usually invoked for the purpose of providing communications security when using an otherwise unprotected communications system. However, it can simultaneously be used for authentication, using the following technique first published in the unclassified literature by Feistel [IBM Research Report RC-2827 (1970)]: the user, at a terminal, begins by by-passing the enciphering equipment. He types his name.

This name passes, unenciphered, through the communication system to the computer he plans to use. The computer looks up the name, just as with the password system. Associated with each named principal, instead of a secret password, is a secret transformation key. The computer loads this transformation key into its enciphering mechanism, turns it on, and attempts to communicate with the user. Meanwhile, the user has loaded his copy of the transformation key into his enciphering mechanism, and turned it on. Now, if the keys are identical, exchange of some standard hand-shaking sequence will succeed. If they are not identical, the exchange will fail, and both the user and the computer system will encounter unintelligible streams of bits. If the exchange succeeds, the computer system is certain of the identity of the user, and the user is certain of the identity of the computer.* The secret

---

* Actually, there is still one uncovered possibility: a masquerader could exactly record the enciphered bits in one communication, and then intercept a later communication and play them back verbatim. (This technique is sometimes called spoofing.) Although the spoofer may learn nothing by this technique, he might succeed in thoroughly confusing the user or the computer system. A simple protection technique is for the computer to immediately use the enciphered connection to transmit the current date and time, and request the user to echo it back. Each successive message can then include as a cross-check a short piece of the previous message. This technique is analyzed in detail by Smith et al. [Proc. ACM 25 Conf. (1972), 282-298].

authenticator--the transformation key--has not been transmitted through the communication system. If communication fails because either the user is unauthorized or the system has been replaced by a masquerader, the legitimate party to the transaction has immediate warning of the apparent illegitimacy of the other party.

Relatively complex elaborations of these various strategies have been implemented, differing both in economics and in assumptions about the psychology of the prospective user. For example, Branstad [Proc. AIAA Comp. Network Systems Conf., paper 73-427 (1973)] explored in detail strategies of authentication in multinode computer networks. Such elaborations, though fascinating to study and analyze, are diversionary to our main topic of protection mechanisms, to which we now return.

Shared information

The virtual machines of the earlier section were totally independent, so far as information accessibility was concerned. This property means that each user might just as well have his own private computer system, and with the steadily declining costs of computer manufacture there are few technical reasons not to use a private computer. On the other hand, for many applications some sharing of information among distinct users is essential. For example, there may be a library of commonly used, reliable programs. Individual users may create new programs that other users would like to use. Distinct users may wish to be able to update a common data base, such as a file of airline seat reservations, or a collection of source language programs which implement a biomedical statistics system. In all these cases, the virtual machine is inadequate, because of the total isolation of its users. Before extending the virtual machine example any further, let us return to our abstract discussion of guards protecting objects with walls.

The implementations of protection mechanisms which permit sharing tend to fall into the two general categories described by Wilkes [Time-Sharing Computer Systems (American Elsevier, 1972)]:

1. "List-oriented", in which the guard holds a list of names of authorized users, and the user carries a unique, unforgeable identifier which must appear on the guard's list if access is to be permitted. A store clerk checking a list of credit customers is an example of a list-oriented implementation in practice. The individual might use his driver's license as a unique, unforgeable identifier.
2. "Ticket-oriented", in which the guard holds the description of a single identifier, and each user has a collection of unforgeable identifiers, or tickets, corresponding to the objects to which he has been authorized access. A doorlock which opens with a key is probably the most common example of a ticket-oriented mechanism; the guard is implemented as the hardware of the lock, and the matching key is the (presumably) unforgeable authorizing identifier.

Authorization, which is defined to be giving a user permission to access some object, is different in these two schemes. In a list-oriented system, a user is authorized to use an object by placing his name on the guard's list for that object. In a ticket-oriented system, a user is authorized by giving him a ticket for the object.

We can also note a crucial mechanical difference between the two kinds of implementations: the list-oriented mechanism requires that the guard examine his list at the time access is requested, which tends to mean that the examination is serial with the access. On the other hand, the ticket-oriented mechanism places on the user the burden of choosing which ticket to present, a task he can

combine with deciding which information to access. The guard needs only to do a single comparison of the presented ticket with his own, perhaps in parallel with the physical memory access. Because of their serialization of checking and access, list-oriented mechanisms are not often used in applications where traffic is high. On the other hand, ticket-oriented mechanisms typically require considerable technology to control forgery of tickets and to control passing around of tickets from one user to another. As a rule, most real systems contain both kinds of sharing implementations: a high-level, list-oriented system at the user interface; and a high-speed, ticket-oriented system in the underlying hardware implementation. This kind of arrangement requires introduction of a specialized list-oriented guard* whose only purpose is to hand out temporary tickets which the remaining (ticket-oriented) guards will honor. Considerable complexity then arises from the need to keep authorizations, as represented in the two systems, synchronized with each other. Most of the differences among computer protection systems lie in the extent to which the underlying ticket-oriented system is architectually visible to the user.

Finally, let us consider the degenerate cases of list- and ticket-oriented systems. If in a list-oriented system it happens that each guard's list of authorized users contains exactly one entry, we have a "complete isolation" kind of protection system, one in which no sharing of information among users can take place. Similarly, if we have a ticket-oriented system in which there exists only one ticket for each object in the system, we discover that we again have the identical "complete isolation" kind of protection system. Thus the "complete isolation" protection system turns out to be a particular degenerate case of both the list-oriented and the ticket-oriented protection implementations.

---

* This specialized mechanism is called an agency by Branstad [AIAA Paper 73-427 (1973)].

These observations are important in examining real systems, since they usually consist of several layers of interacting protection mechanisms, some of which are list-oriented, some of which are ticket-oriented, and some of which provide complete isolation and therefore may happen to be implemented as degenerate examples of either of the other two, depending on local circumstances.

We should comment on the nature of the user in these transactions. Since we are concerned with protection of information from programs that are executing, the user is the person who assumes responsibility for the actions of an executing program. Inside the computer system, the appropriate abstraction to use is that of a process, since one or more processes can be identified with the activities directed by the user. Thus we are discussing accesses made by a process. In a list-oriented system the guard is concerned with knowing whose process is attempting to make an access. We can imagine, for example, that the process has been marked with an unforgeable label identifying the user responsible for its actions, and the guard inspects this label when making access decisions. In a ticket-oriented system, the guard cares only that a process present the appropriate unforgeable ticket when attempting an access. The connection to a responsible user is more diffuse, being partially the responsibility of the agency which issued the tickets. In either case, we conclude that in addition to the information inside the impenetrable wall, there are two other things which must be protected:

1. The guard's authorization information.
2. The association between a user and the values of the unforgeable label or set of tickets associated with his processes.

Since an association with some user is essential for establishing responsibility for the actions of a process, it is useful to introduce an abstraction for that responsibility: the principal. A principal is, by definition

the entity responsible for the activities of a process.* In the situations we have discussed so far, the principal is exactly the user outside the system. However, there are situations in which a one-to-one identification of individuals with principals is not adequate. For example, a user may be responsible for some very valuable information, and authorized to use it. On the other hand, he may wish, on some occasion, to use the computer for some purpose unrelated to the valuable information. To prevent accidents, he may wish to identify himself with a different principal, one which does not have access to the valuable information--following the principle of least privilege. In this case there is a need for two different principals which correspond to the same user.

For a different example, one can envision a data base that is to be modified only if a committee of individuals all agree. Thus there might be a single principal, authorized to make the modification, but that cannot be used by any single individual; all of the committee members must agree upon its use simultaneously.

Summarizing, then, a principal is essentially the unforgeable identifier to be attached to a process in a list-oriented system. When a user first approaches the computer system, that user must identify the principal to be used. Some authentication mechanism, such as a request for a secret password, and which itself may be either list- or ticket-oriented or of the complete isolation type, establishes the user's right to use that principal. Then, a computation is begun in which all the processes of the computation are labeled with the identifier of that principal, and all further actions of these processes

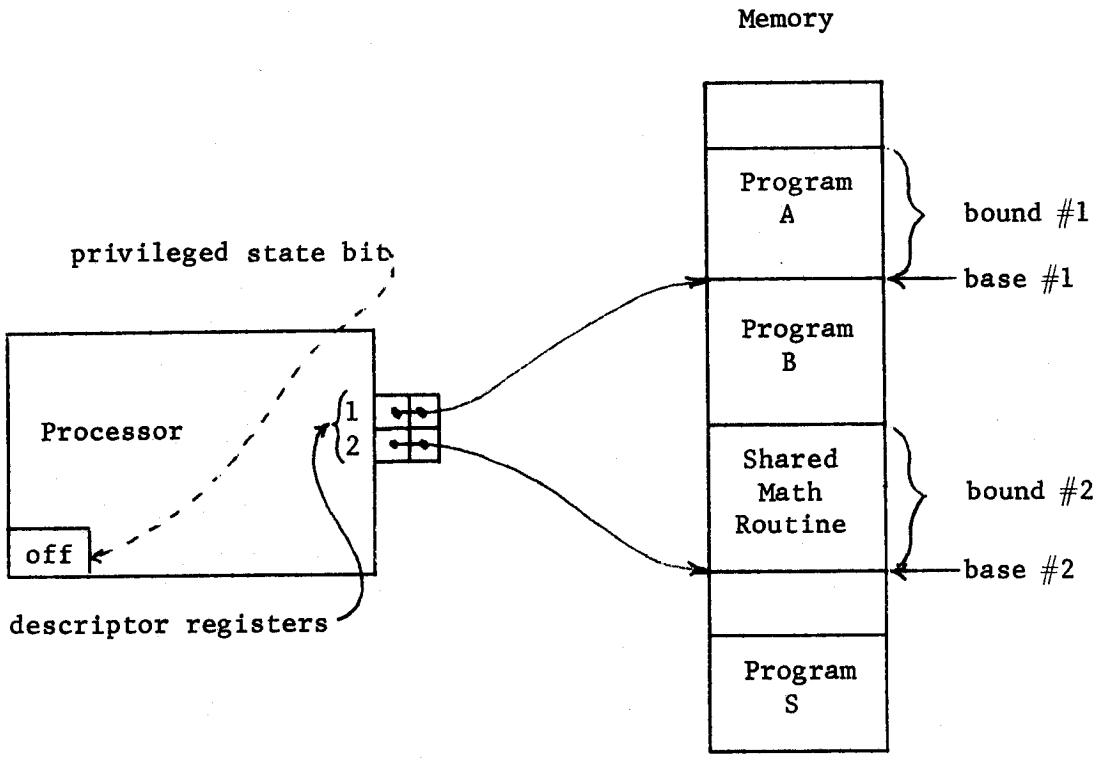
---

* The word "principal", suggested by Dennis and Van Horn [CACM 9, 3 (1966), 143-155], is used for this abstraction because of its association with the legal concepts of authority, liability, and responsibility.

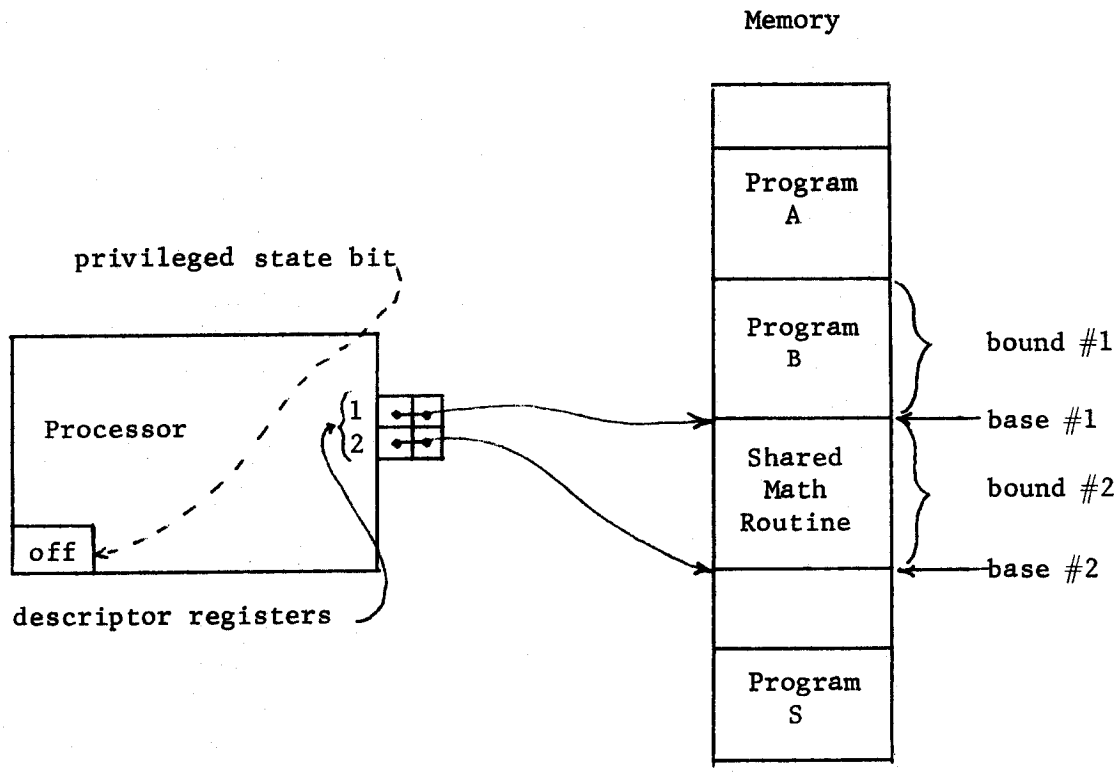
may be considered to be the responsibility of that principal. In a ticket-oriented system, the concept of principal is more diffuse, and responsibility for the activity of a process is correspondingly more difficult to pinpoint.

Some of these ideas will become clearer if we return to our example system and extend it to include sharing. Let us consider for a moment the problem of sharing a library program--say a mathematical function subroutine. We could place a copy of the math routine in the long-term storage area of each virtual machine that had a use for it. This scheme works, but has several defects. Most obviously, the multiple copies require multiple storage spaces. More subtly, the scheme does not respond well to dynamic changes: if a newer, better math routine is written, the upgrading of the multiple copies requires effort proportional to the number of users. These two observations suggest that one would like to have some scheme to allow different users to have access to a single master copy of the program. Then, the storage space will be smaller and the communication of updated versions will be easier.

In terms of the virtual machine model of our earlier example, we can accomplish sharing of a single copy of the math routine by adding to the processor a second descriptor register, as in figure 6-2, placing the math routine somewhere in memory by itself, and placing a descriptor for it in the second descriptor register. Following the previous strategy, we assume that the privileged state bit assures that the supervisor program is the only one permitted to load either descriptor register. In addition, some scheme must be provided in the architecture of the processor to permit a choice of which descriptor register is to be used for each address generated by the processor. A simple scheme would be to let the high-order address bits select the descriptor register. Thus in figure 6-2, all addresses in the lower half of the address range would be interpreted relative to descriptor register 1, and addresses in the upper half of the address range would be relative to descriptor register 2. An alternate scheme, suggested by Dennis [JACM 12, 4 (1965), 589-602], is to add explicitly to the format of instruction words a field which



a. Program A in control of processor.



b. Program B in control of processor.

Figure 6-2 -- Sharing of a math routine by use of two descriptor registers.

selects the descriptor register intended to be used with the address in that instruction. The use of descriptors for sharing information is intimately related to the addressing architecture of the processor, a relation which can be the cause of considerable confusion. The reason why descriptors are of interest for sharing becomes apparent by comparing parts a and b of figure 6-2. When program A is in control, it can access only itself and the math routine; similarly, when program B is in control, it can access only itself and the math routine. By use of descriptors, sharing of the math routine has been accomplished while maintaining isolation of program A from program B.

The effect of sharing is shown even more graphically in figure 6-3, which is just figure 6-2 redrawn with two processors, one executing program A and the other one executing program B. Whether or not there are actually two processors is less important than the existence of the conceptually parallel access paths implied by figure 6-3. Conceptually, every process of the system may be viewed as having its own private processor, capable of access to the memory in parallel with every other process. There may be an underlying processor multiplexing facility which distributes a few real processors among the many processes, but such a multiplexing facility is essentially unrelated to protection. Recall that a processor is not permitted to load its own protection descriptor registers. Instead, it must call or trap to the supervisor program, S, which call or trap causes the privileged state bit to go on and thereby permits the supervisor program to control the extent of sharing among processors.* On the other hand, the processor multiplexing facility must be prepared to switch the entire state of a processor from one process to another, including the values of the protection descriptor registers.

---

* The supervisor is thus a primitive example of a protected subsystem, of which more will be said later.

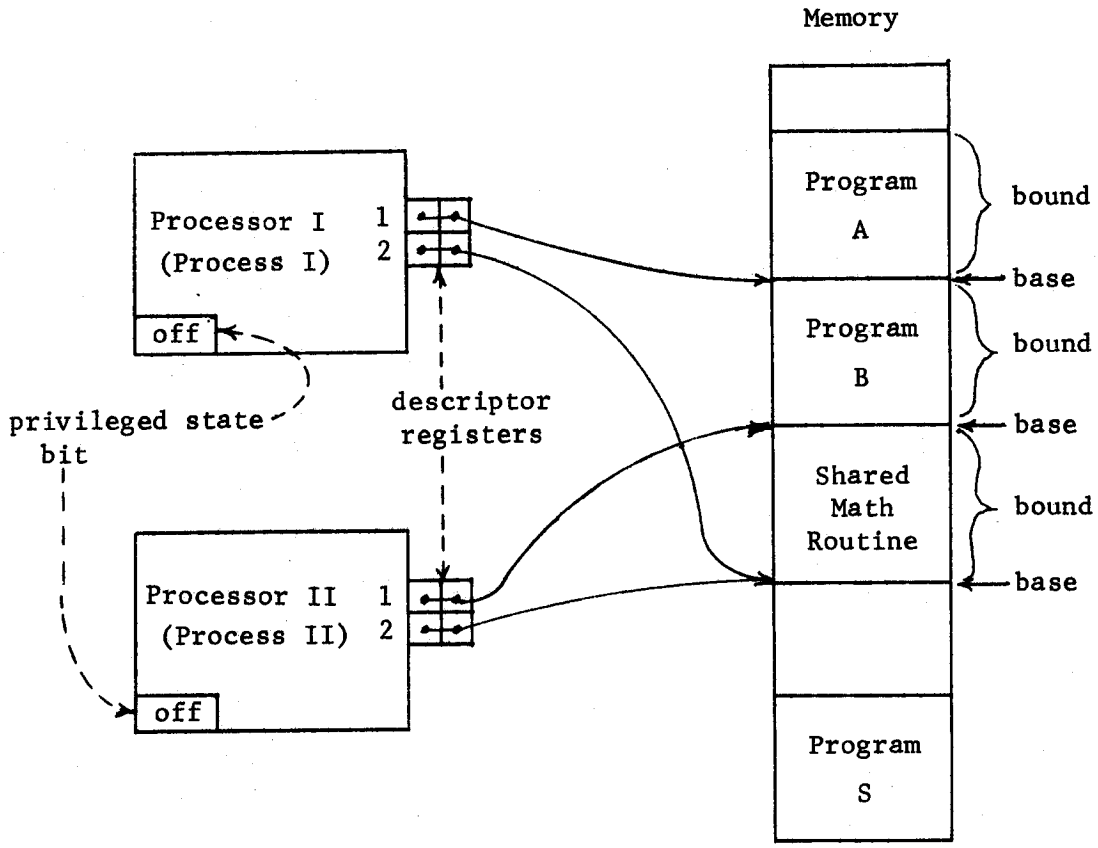


Figure 6-3 -- Figure 6-2 redrawn to show sharing of a math routine by two parallel processors, and thus two parallel processes.

Although the basic mechanism to permit information sharing is now in place, there is a remarkable variety of implications which follow from its introduction and which, to come out in a desirable way, require further mechanisms. These implications include the following: 1) if process I can overwrite the shared math routine, then it could install a trap which could disrupt the work of process II. 2) The shared math routine must be careful about making modifications to itself and about where in memory it writes temporary results, since it is to be used by independent computations, perhaps simultaneously. 3) The scheme needs to be expanded and generalized to cover the possibility that there is more than one program or data base to be shared. 4) The supervisor needs to be informed about which principals are authorized to use the shared math routine (unless it happens to be completely public with no restrictions).

Let us consider these four implications in order. If the shared segment is a procedure, then to avoid the possibility of process I's maliciously overwriting it we can restrict the methods of access. Process I will need to retrieve instructions from the area of the shared procedure, and may need to read out the values of constants enbedded in the program, but it has no need to be able to write into any part of the shared procedure. We may accomplish this restriction by extending the descriptor registers and the descriptors themselves to include accessing permission, an idea introduced for different reasons in the original Burroughs B5000 design [The Descriptor (1961), Burroughs Corp.]. For example, we may add three bits, one controlling permission to read, the second permission to write, and the third permission to retrieve instructions (execute) in the storage area controlled by each descriptor, as in figure 6-4. In process I of figure 6-3, descriptor one would have all three permissions granted, while descriptor two would permit only reading

of data and execution of instructions.* An alternative, but much less satisfactory scheme, would be to attach the permission bits directly to the storage areas containing the shared program or data. Such a scheme is less satisfactory because, unlike the descriptors so far outlined, permission bits attached to the data would provide identical access to all processes which had a descriptor. Although identical access for all users of the shared math routine of figure 6-2 might happen to be acceptable, the case of a shared

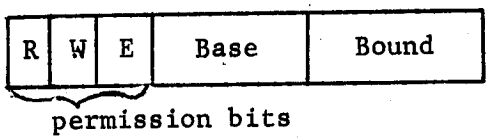


Figure 6-4 -- A protection descriptor containing read, write, and execute permission bits.

data base, with several users having permission to read but a few also having permission to write, could not be accomplished.

The second implication of a shared procedure, mentioned before, is that the shared procedure must be careful in where it stores temporary results, since it may be simultaneously used by several processes. In particular, it should avoid modifying itself. The enforcement of access permission by descriptor bits further constrains the situation: to prevent program A from writing into the shared math routine we have also prohibited the shared math routine from writing into itself, since the descriptors do not change when, for example, program A transfers control to the math routine.** The math routine will find

* In some systems, four or more bits are used, separately controlling, for example, permission to call as a subroutine, to use indirect addressing, or to store certain specialized processor registers. Such an extension of the idea of separately controllable permissions is not important to the present discussion.

** Actually, this constraint has been introduced by our assumption that descriptors must be maintained by the supervisor. With the addition of protected subsystems, described later, this constraint is relaxed.

that it can read but not write into itself, but that it can both read and write into the area of program A. Thus, if program A allocates an area of its own address range for the math routine to use as temporary storage, then the math routine can read arguments, store temporary values, and write results there. Of course, program A cannot allocate any arbitrary set of addresses for this purpose: the specifications of the math routine would have to include details about what addresses it is programmed to use relative to the first descriptor; program A must expect those addresses to be the ones used when it calls the math routine. Similarly, program B, if it wishes to use the shared math routine, will have to reserve the same addresses in its own area. Although these particular address reservation conventions are somewhat constraining, it is clear that it is possible to develop a shared, read-only procedure.*

As for the third implication, the need for expansion and generalization, the set of addresses which may be accessed for a particular loading of the two descriptor registers in our example is called the addressing domain of the processor and of the principal that is responsible for the activities of the processor. It should be clear that a simple generalization of our example to permit several distinct shared items could be accomplished by merely increasing the number of descriptor registers, and informing the supervisor which shared objects are authorized to be in the addressing domain of each process. However, there are two substantially different forms of this generalization, named the capability system and the access-control-list system. In terms of the earlier discussion, the capability system is ticket-oriented, while the access-control-list system is list-oriented. As mentioned earlier, most real systems use a combination of these two forms, the capability system in the

* Most systems which permit shared procedures use additional hardware to allow more relaxed communication conventions. For example, a third descriptor register can be reserved to point to an area used exclusively as a stack for communication and temporary storage by shared procedures; each distinct process would have a distinct stack. See, e.g., Daley and Dennis [CACM 11, 5 (1968), 306-312].

hardware base and the access-control-list system in the user interface. Before we can pursue these generalizations, and the fourth implication, authorization, some more ground work must be laid.

II: General Descriptor-Based Protection Systems

Separation of addressing and protection*

As mentioned earlier, descriptors have been introduced here for the purpose of protecting information, while they are also used in some systems to organize naming, addressing, and storage allocation. For the present, it will be useful to separate the organizational uses of descriptors from their protective use, by requiring that all memory accesses go through two levels of descriptors. It should be realized, however, that in many implementations the two levels are actually merged into one, and the same descriptors serve both organizational and protective purposes.

Conceptually, we may accomplish this separation by enlarging the function of the memory system to provide uniquely identified (and thus distinctly addressed) storage areas, commonly known as segments. For each segment there must be a distinct addressing descriptor, and we will consider the set of addressing descriptors to be part of the memory system, as in figure 6-5. Every collection of data items worthy of a distinct name, distinct scope of existence, or distinct protection would be placed in a different segment, and the memory system itself would be addressed with two-component addresses: a unique segment identifier (to be used as a key by the memory system to look up the appropriate descriptor) and an offset address which indicates which part of

---

* Extension of the discussion of information protection beyond multiple descriptors requires an understanding of descriptor-based addressing techniques. Although the following section contains a brief review, the reader not previously familiar with descriptor-based architecture may find the treatment too sketchy. The books Time-Sharing Computer Systems, by M. V. Wilkes (American Elsevier, 1972) and Timesharing System Design Concepts, by R. W. Watson (McGraw-Hill, 1970), provide tutorial treatments of descriptor-based addressing, while the paper by Dennis [JACM 12, 4 (1965)] provides in-depth technical discussion. Two books by E.I. Organick, The Multics System (M.I T. Press, 1971), and Computer System Organization (Academic Press, 1973) provide a broad discussion and case studies.

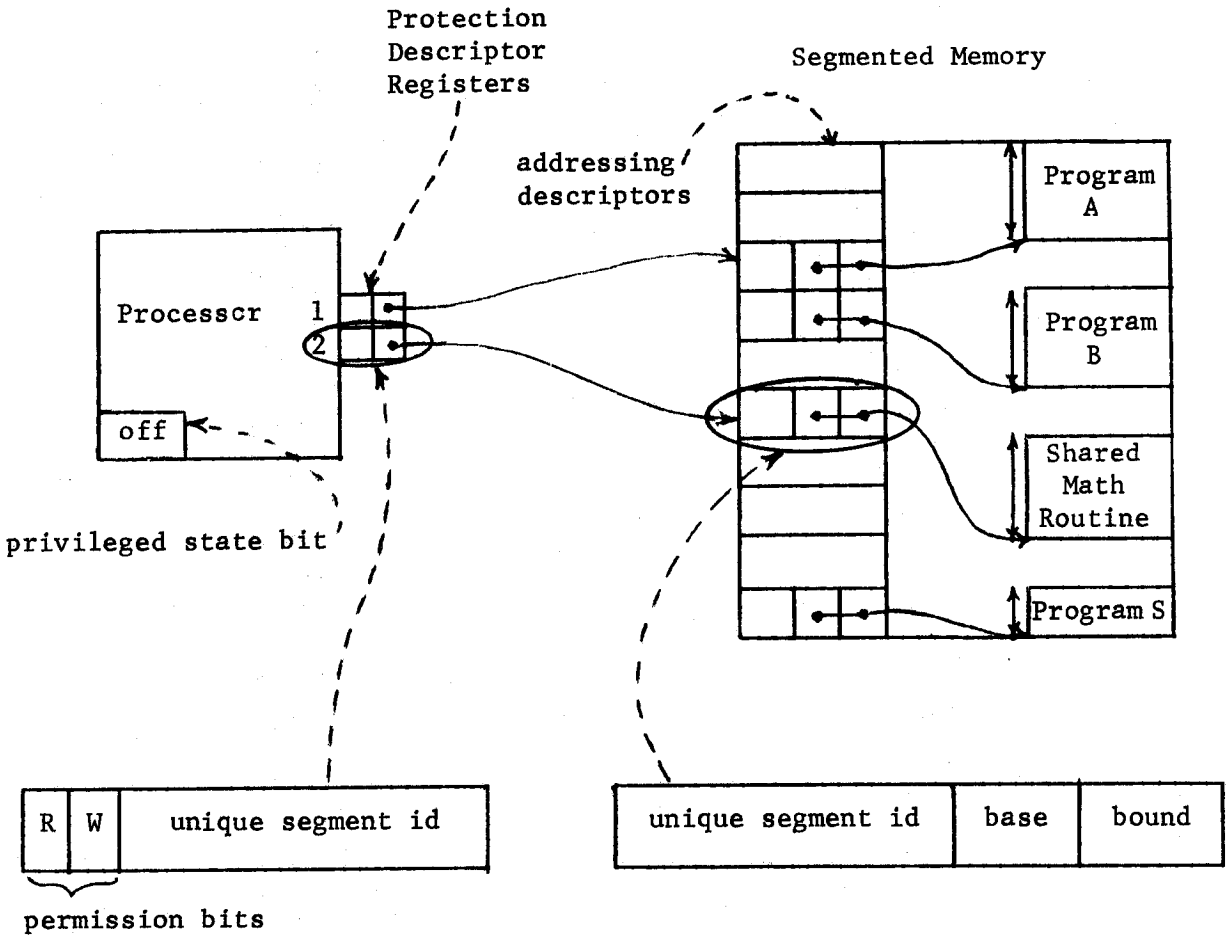


Figure 6-5 -- An organization separating addressing from protection descriptors, using a segmented memory. The address passed from the processor to the memory consists of two parts: a unique segment identifier, and an offset. Program A is in control. (Compare with figure 6-2a.) In later figures the addressing descriptors will be omitted for clarity, but they are assumed to be present in the actual implementation of a segmented memory.

the segment is to be read or written. All users of the memory system would use the same single set of addressing descriptors, and these descriptors would have no permission bits--only a base and a bound value. This scheme is functionally similar to that used in the Burroughs B5700/6700 or Honeywell Multics systems, in that it provides a structured addressing space with an opportunity for systematic and automatic storage allocation.

The unique identifiers used to label segments are an essential cornerstone of this organization. They will be used by the protection system to identify segments, so they must never be reused. One way of implementing unique identifiers is to provide a hardware counter register which operates as a clock, counting, say, microseconds, and large enough to never overflow in the lifetime of the memory system. The value of the clock register at the time a segment is created can then be used as that segment's unique identifier.* As long as the memory system stores anything, the time base of the clock register cannot be changed.

The processor of figure 6-5 contains, as part of its state, protection descriptors similar to those of figures 6-1 and 6-2, and all references by the processor are constrained to be to segments described by the protection descriptors. The protection descriptor itself no longer contains a base and bound; instead it contains the unique segment identifier which the memory system requires as the first part of its two part address for accessing that segment. Thus, from the point of view of a program stored in one of the segments of memory, this system is indistinguishable from that of figure 6-2. Note in figure 6-5 that although addressing descriptors exist for the segments containing program B and program S (the supervisor), they are not accessible to the processor since currently it has are no protection descriptors for those two segments. Given this situation, it is appropriate to now distinguish between the

---

* Since the unique identifier will be relied upon by the protection system, it may be a good idea to guard against the possibility that an accidental hardware error while manipulating a unique identifier results coincidentally in accessing the wrong segment. One form of guard is to encode the clock reading in some larger number of bits, using a multiple-error detecting code, to use the encoded value as the unique identifier, and to have the memory system check the coding of each unique identifier presented to it.

addressing domain, consisting of all the segments for which addressing descriptors exist, and the protection domain, which consists of those segments for which protection descriptors exist. If the supervisor switches control of a processor from one process to another, it would first reload the protection descriptors; the protection domain thus is different for different users, while the addressing domain remains the same for all users.

With this architectural separation of the addressing function from the protection function, we may now examine the two forms of generalization of protection systems: the capability system and the access-control-list system.

The capability system

The simplest generalization, conceptually, is the capability system, suggested by Dennis and Van Horn [CACM 9, 3 (1966), 143-155], and first implemented on an M.I.T. PDP-1 computer [Ackerman and Plummer, Proc. ACM 1st Symposium on Operating Systems Principles (October, 1967)].* There are many different detailed implementations for capability systems; we illustrate with a specific example. Recall that we introduced the privileged state bit to control who may load values into the protection descriptor registers. An alternative method of maintaining the integrity of these registers would be as follows: allow any program to load the protection descriptor registers, but only from locations in memory which have been previously certified to contain acceptable protection descriptor values. Suppose, for example, that every location in memory were tagged with an extra bit. If the bit is off,

---

* A detailed analysis of the resulting architectural implications was made by Fabry and Yngve in 1968 in a series of progress reports of the Institute of Computing Research at the University of Chicago. The capability system is a close relative of the codeword organization of the Rice Research Computer [Iliffe and Jodeit, Computer Journal 5 (Oct., 1962), 200-209], but Dennis and Van Horn seem to be the first to have noticed the application of that organization to inter-user protection.

the word in that location is an ordinary data or instruction word. If the bit is on, it is taken to be a value suitable for loading into a protection descriptor register. The instruction which loads the protection descriptor register will operate only if its operand address leads it to a location in memory which has the tag bit on. To complete the scheme, we should provide an instruction to store the contents of a protection descriptor register in memory, and which turns the corresponding tag bit on; and arrange that all other store instructions set the tag bit off in any memory location they write into. Thus we have two kinds of objects stored in the memory: protection descriptor values and ordinary data values. There are also two sets of instructions, separate registers for manipulating the two kinds of objects, and thus effectively a wall that prevents objects that are subject to general computational manipulation from ever being used as protection descriptor values. This kind of scheme is a particular example of what is called a tagged architecture.* A memory word which contains a protection descriptor value (in our simple tagged system, one which has its tag bit on) is known as a capability. Systems which permit the user to load and store protection descriptor values in memory are called capability systems.

To see how capabilities can be used to generalize our basic sharing strategy, suppose that each processor has several (say four) protection descriptor registers, and that program A is in control of a processor, as in figure 6-6. For clarity, this and future figures omit the addressing descriptors of the segmented memory. The first two protection descriptor registers have already been loaded with values permitting access to two segments: program A, and a segment we have labeled "Catalog for Doe". This latter segment, in our example, contains two

---

* The Burroughs B5700 and its ancestors, and the Rice University Computer [Iliffe and Jodeit, Computer Journal 5 (Oct., 1962), 200-209] are examples of architectures which use multi-bit tags to separately identify instructions, descriptors, and several different types of data. All examples of tagged architecture seem to trace back to suggestions made by J. Iliffe.

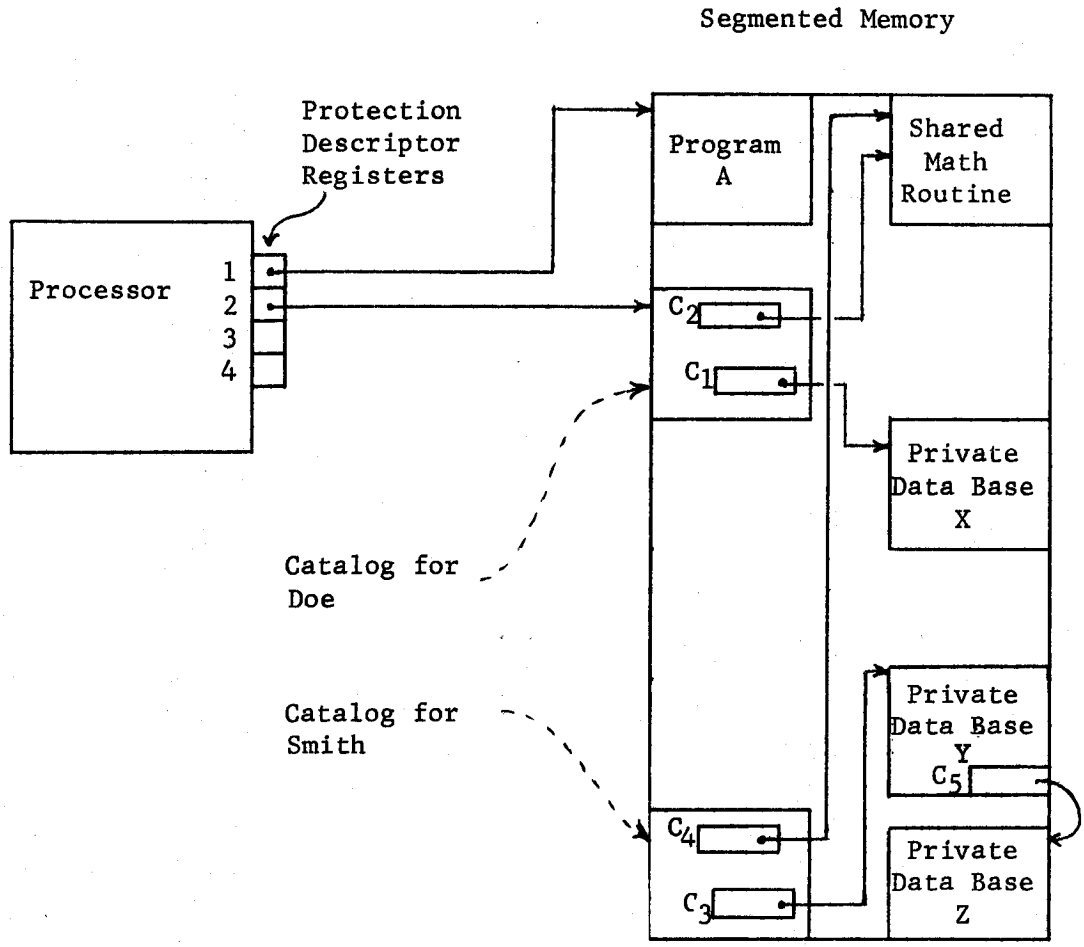


Figure 6-6 -- A simple capability system. Program A is in control of the processor. Note that there is no way for the processor to address Smith's catalog or data base Y. On the other hand, data base X could be accessed by loading capability C_1 into a protection descriptor register. Capability C_1 is loadable because it is stored in a segment which can be reached from a capability already loaded in protection descriptor register number two. Note also that the former function of the privileged state bit has been accomplished by protecting the capabilities. The privileged state bit also has other uses, and will be reintroduced later.

locations which have tags indicating that they are capabilities, C_1 and C_2. Program A may direct the processor to load the capability at location C_2 into one of the protection descriptor registers, and then the processor may address the shared math routine. Similarly, either program A or the shared math routine may the direct the loading of the capability at location C_1 into a protection descriptor register, after which the processor may address the segment labeled "Private Data Base X". By a similar chain of reasoning, another process starting with a capability for the segment labeled "Catalog for Smith" can address both the shared math routine and the segment "Private Data Base Y.

By appropriate advance initialization, we can now arrange for any desired static pattern of sharing of segments. For example, for each user, we can provide one segment for use as a catalog, and place in that catalog a capability for every segment he is authorized to address. Each capability includes read, write, and execute permission bits, so that some users may receive capabilities that permit reading and writing some segment, while others receive capabilities permitting only reading from that same segment. The catalog segment might actually contain pairs: a character-string name for some segment, and the associated capability that permits addressing that segment. A user would create a new segment by calling the supervisor; the supervisor might by convention return with some protection descriptor set to contain a capability for the new segment.* The user could then file his new segment by storing this new capability in his catalog along with a name for the segment. Thus we have an example of a primitive but usable file system to go with the basic protection structure.

---

* The construction of a capability for a newly-created object requires loading a protection descriptor register with the unique identifier of the new segment. This loading can be accomplished either by giving the supervisor program the privilege of loading protection descriptor registers from untagged locations, or else by making segment creation a hardware-supported function which includes loading the protection descriptor register.

To complete the picture, we should provide a tie to some authentication mechanism. Suppose that the system responds to an authentication request by creating a new process and starting it off in a supervisor program that initially has a capability for a user identification table, as in figure 6-7. If a user identifies himself as "Doe" and supplies a password, the supervisor program can look up his identification in the user identification table. It can verify the password, and then load into a protection descriptor register the capability for the catalog associated with Doe's entry in the user identification table. Next, it would clear the remaining capability registers, thus destroying the capability for the user identification table, and then start running some program in Doe's directory, say program A. Program A then can extend its addressability (that is, its protection domain) to any segment for which a capability exists in Doe's catalog.

Note that by providing for authentication we have actually tied together two protection systems:

- an authentication system which controls access of users to named catalog capabilities.
- the general capability system which controls access of the holder of a catalog capability to other objects stored in the system.

The authentication system associates the newly created process with the principal who is responsible for its future activities. Once the process is started, however, the character-string identifier is no longer needed; the associated catalog capability is sufficient. The dropping of the principal identifier is possible because the full range of accessible objects for this user has already been opened up to him by virtue of his acquisition of his catalog capability.

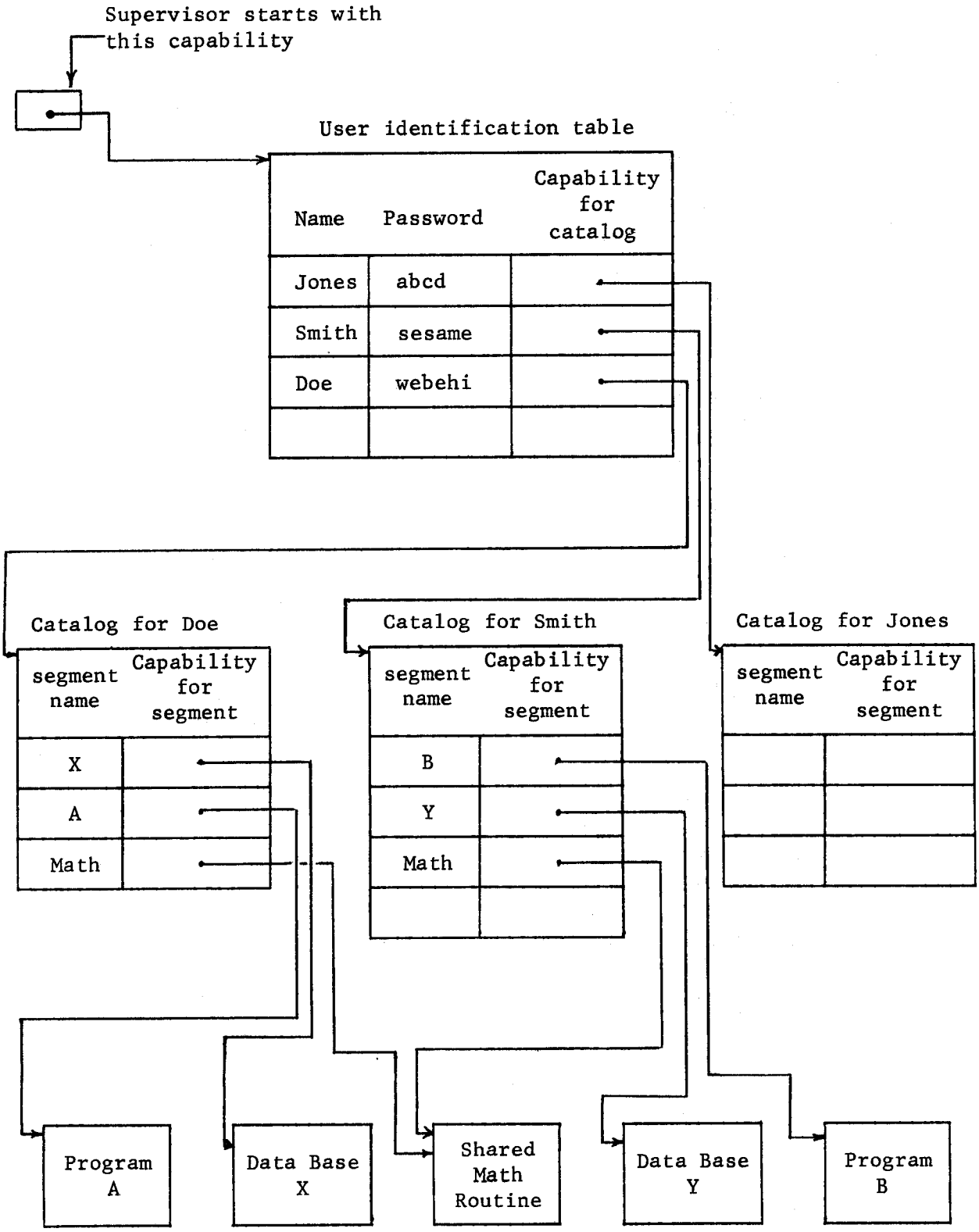


Figure 6-7 -- A capability system with provision for authentication.

With this example of a capability system, a catalog is not a special object. It is merely a segment in which some program chooses to store capabilities which are, by virtue of their tags, protected, unforgeable objects. If in figure 6-7, program A, running under Doe's control, creates a new object, it may choose to place the new capability in the segment X, in a position where it can easily be found later. In such a case, segment X has become, in effect, another catalog. To establish the full range of objects which Doe may address, it is necessary to examine not only the initial catalog segment whose capability is contained in the user identification table, but also all segments it contains capabilities for, and all segments they contain capabilities for, etc.

The scheme described so far admits any desired static arrangement of accessing authorization. However, we have not yet provided for the possibility that Doe, upon creating a new segment, might wish to authorize Smith to have access to it. We shall call this operation dynamic authorization. The dynamic authorization of sharing is a topic which must be examined quite carefully, since it exposes several subtle issues which are fundamental to sharing and of protection.

The dynamic authorization of sharing

One might initially propose that dynamic authorization could be handled very simply by arranging that Doe have a capability for Smith's catalog, so that he can store a copy of the capability for the new segment in Smith's catalog. But this approach has a defect: allowing Doe to have a capability to write into Smith's catalog would give Doe access to overwrite and destroy all of Smith's capabilities. The inverse strategy of giving Smith a capability to read Doe's catalog would give Smith access to all of Doe's segments. A more "secure" approach to the problem is needed.

If the possibility of sharing had been acticipated, the form of the anticipation could be that both Doe and Smith initially have a capability allowing reading and writing a communication segment, whose only use is to pass messages and capabilities between Doe and Smith. Doe's program then deposits the capability for his newly created object in the communication segment for Smith, and Smith's program can pick it up and use it or catalog it, at Smith's convenience. But that description oversimplifies one step. Both Doe's and Smith's programs somehow had to track down the capability for the common communication segment; how did they know what to look for? Consider the case of the sender, Doe's program, first. Presumably it looked in some catalog for the name "Smith" and found the capability for the communication segment that way. But how did Doe's program know to look for the name "Smith"? The character-string name may have been embedded in the program by Doe or he may have typed it in to his program as it was running, but either way one thing is crucial: that there has been a secure path from Doe, who is authorizing the passing of the capability, to the program, which is carrying it out. Next, we should ask, where did Doe find out the character string name "Smith" so that he could type it in or embed it in his program? Presumably, via some path outside the computer, he learned Smith's name. Perhaps Smith shouted it down the hall to him.* The method of communication is not important, but the fact of the communication is: for dynamic authorization of sharing within a computer, there must be some previous communication from the recipient to the sender, external to the computer system. Further, this reverse external communication path must be sufficiently secure that the sender is certain of the system-catalogued name of the intended recipient. That name might be, for example, the identifier of the recipient's principal within the computer system. If so, the sender can be sure that only programs run under the responsibility of that principal will have access to his new object.

---

* Imagery due to Butler Lampson.

An analogous chain of reasoning applies to Smith's program as the recipient of the capability for the new object. Smith must learn from Doe some piece of information sufficient that he can instruct his program to look in the correct communication segment for the capability which Doe is sending. Again, if Doe's principal identifier is the name used in Smith's catalog of communication segments, Smith can be certain that only some program run under Doe's responsibility could possibly have sent the capability. In summary, here is the complete protocol for dynamically authorizing sharing of a new object:

1. Sender learns receiver's principal identifier, via a communication path outside the system.
2. Receiver learns sender's principal identifier, via a communication path outside the system.
3. Sender transmits receiver's principal identifier to some program running inside the system under the sender's responsibility.
4. Sender's program uses receiver's principal identifier to insure that only processes operating under the receiver's responsibility will be able to obtain the capability being transmitted.
5. Receiver transmits sender's principal identifier to some program running inside the system under the receiver's responsibility.
6. Receiver's program uses the sender's principal identifier to insure that only a process operating under the sender's responsibility could have sent the capability being received.

Although this analysis may seem somewhat trivial, it is important because in a capability system the role of the principal identifier and the concept of responsibility do not correspond to any obvious observable mechanisms, yet they must be present somewhere in order to carry out the intentions of the outside users. The protocol described above always applies, even though parts of it may be implicit or hidden. As will be seen later, an equivalent protocol

also applies in access-control-list systems.

Our analysis of the dynamics of authorizing sharing has been in terms of a private communication segment between every pair of users, a strategy which would lead, with N users, to N(N-1)/2 communication segments. To avoid this square-law growth, one might prefer to use some scheme which dynamically constructs the communication paths also, such as having special hardware or a protected subsystem which implements a single "mailbox segment" for each user in which he receives messages and capabilities sent by all other users. In this case the mechanism which implements the mailbox segments must be a protected mechanism, since it must infallably determine the principal identifier of the sender of a message, and label the message with that identifier, so the receiver can reliably carry out step six of the protocol. Similarly, as the sender's agent, it must be able to associate the recipient's principal identifier with the recipient's mailbox, so that the sender's responsibility in step four of the protocol is correctly carried out.

More problems of dynamics

The capability system has as its chief virtues its inherent efficiency and simplicity. Efficiency comes from the ease of testing the validity of a proposed access: if the accessor can present a capability the request is valid. The simplicity comes from the natural correspondence between the mechanical properties of capabilities and the semantic properties of addressing variables. The semantics for dynamically changing addressability which are part of modern languages such as PL/I and ALGOL 68 fit naturally into a capability-based framework by using capabilities as address (pointer) variables. Straightforward additions to the capability system allow it to gracefully implement languages with dynamic type extension [Wulf, et al., CACM 17, 6 (1974), 337-345]. On the other hand, there are several potential problems with the capability system

as we have sketched it so far. The first problem is that if Doe has a change of heart--he suddenly realizes that there is confidential information in the segment for which he permitted access to Smith--there is no way that he can disable the copy of the capability which Smith now has stored away in some unknown location. Unless we provide additional control, his only recourse is to destroy the original segment, an action which may be disruptive to other, still trusted, users who also have copies of the capability. Thus, revocation of access is a problem.

A second, related difficulty is that Smith may now make further copies of the capability and distribute them to other users, without the permission, or even the knowledge, of Doe, the original owner. We have not provided for any control of propagation.

Finally, the only possible way in which Doe could make a list of all users who currently can reach his segments would be by searching every segment in the system for copies of the necessary capability. That search would be only the beginning, since there may be many paths by which users could reach those capability copies; every such path must be found and documented, a task requiring both an unreasonable amount of computation and also a complete bypassing of the protection mechanisms. Thus, review of access is a problem.*

To help counter these problems, constraints on the use of capabilities have been proposed or implemented in some systems. For example, a bit added to

---

* A fourth problem, not directly related to protection, is called the lost-object problem by Neumann et al., [IRIA Workshop on Protection (1974)]. If all copies of some capability are overwritten, the object which that capability describes would become inaccessible to everyone, but the fact of its inaccessibility is hard to discover, and recovery of the space it occupies may be hard to achieve. The simplest solution is to insist that the creator of an object be systematic in his use of capabilities, and remember to destroy the object before discarding the last capability copy. Since most computer operating systems provide for systematic resource management this simple strategy is usually adequate.

a capability (the copy bit) may be used to indicate whether or not the capability may be stored in a segment. If one user gives another user access to a capability with the copy bit off, then the second user could not make copies of the capability he has borrowed; propagation would be prevented, at the price of some loss of flexibility. (A version of this scheme was implemented in the University of California (Berkeley) CAL time sharing system.)

Alternatively, some segments (perhaps one per user) may be designated as capability-holding segments, and only those segments may be targets of the instructions which load and store descriptor registers. This scheme may reduce drastically the effort involved in auditing, and make revocation possible, since only capability-holding segments need be examined. (The Cambridge Capability System is organized approximately this way.)

A third approach is to associate a depth counter with each descriptor register. The depth counter would initially have the value, say, of one, placed there by the supervisor. Whenever a user loads a descriptor register from a place in memory, that descriptor register receives a depth count which is one greater than the depth count of the descriptor register which contained the capability which permitted the loading. Any attempt to increase a depth count beyond, say, three, would constitute an error, and the processor would fault. Thus, the depth counters limit the distance that one may follow a chain of capabilities. Again, this form of constraint reduces the effort of auditing, since one must trace chains back only a fixed number of steps to get a list of all potential accessors. (The M.I.T. CTSS used a software version of this scheme, with a depth limit of two.)

To gain more precise control of revocation, Redell [Ph. D. thesis, Berkeley (1974)] has proposed that the basic capability mechanism be extended to include the possibility of forcing a capability to go through an indirect

address before reaching the actual object of interest. This indirect address would be an independently addressable, recognizable object, and anyone with an appropriate capability for it could destroy the indirect object, thereby revoking access to anyone else who had been given a capability for that indirect object. By constructing a separate indirect object for each different user he shared an object with, the owner of the object could then maintain the ability to independently revoke access for each other user. The indirect objects would be implemented within the memory-mapping hardware (e.g., the addressing descriptors of figure 6-5) both to allow high speed bypassing tricks if frequent multiple indirections occur and also to allow the user of a capability to be oblivious to the existence of the indirection.* Redell's indirect objects are precursors of the access controllers of the access-control-list system, described in the next section. While providing a systematic revocation strategy (if their user develops a protocol for systematically using them) the indirect objects provide only slight help for the problems of propagation and auditing.

The basic trouble being encountered is that a binding of a permission to a principal is accomplished any time a capability is copied. Unless an indirect object is created for the copy, there is no provision for reversing this binding, and the ability to make a further copy (and potentially a new binding) is coupled to possession of a capability rather than independently controllable. Restrictions on the ability to copy, while helping to limit the number or kind of bindings, also hamper the simplicity, usefulness, flexibility, and uniformity of capabilities as addresses. For this reason, the most effective way of preserving the flexibility and efficiency of

---

* In the HYDRA system [Wulf et al., CACM 17, 6 (1974), 337-345] an equivalent functional effect is provided by allowing capabilities to be used as indirect addresses, and by separately controlling permission to use them that way. This strategy, in contrast to Redell's, makes the fact of indirection known to the user, and is also not as susceptible to speedup tricks.

capabilities is to use them only in the bottommost implementation layer of a computer system, where the lifetime and scope of the bindings can be controlled. The authorizations implemented by the capability system are then systematically maintained as an image of some higher level authorization description, usually some kind of an access-control-list system, which provides for direct and continuous control of all permission bindings.

The access-control-list system

The usual strategy to provide for reversibility of bindings is to control the time they occur--typically by delaying them until the last possible moment. The access-control-list system provides exactly such a delay by inserting an extra permission-checking step at the latest possible point: as each memory access is made. Where the capability system was basically a ticket-oriented strategy, the access-control-list system is a list-oriented strategy. Again, there are many possible mechanizations, and we must choose one for illustration. For ease of discussion, we will describe a mechanism implemented completely in hardware (perhaps by micro-programming) although access-control-list systems have historically been implemented partly with interpretive software, driving an underlying hardware capability system.

Return first to the system of figure 6-5, which identified protection descriptors as a processor mechanism and addressing descriptors as a memory mechanism. Suppose that the memory mechanism is further augmented as follows: whenever a user requests a segment to be created, the memory system will actually allocate two linked storage areas. One of the storage areas will be used to store the data of the segment, as usual, while the second will be treated as a special kind of object which we will call an access controller. An access controller contains two pieces of information: an addressing descriptor for the associated segment and an

access-control-list, as in figure 6-8. An addressing descriptor for the access controller itself is assigned a unique identifier and placed in the map used by the memory system to locate objects. Thus the access controller is to be used as a kind of indirect address, as in figure 6-8. The processor, in order to access a segment, must thus supply the unique identifier of that segment's access controller. However, there is no longer any need for these unique identifiers to be protected, so the former protection descriptor registers can be replaced with unprotected pointer registers, which can be loaded from any addressable location with arbitrary bit patterns. Of course, only bit patterns corresponding to the unique identifier of some segment's access controller will work. A data reference by the processor then proceeds in the following steps, keyed to figure 6-9:

1. The program wishes to generate a write reference to the segment described by pointer register three with offset k.
2. The unique identifier found in pointer register three is used to address access controller AC_1.
3. The access-control-list in AC_1 is searched to see if this user's principal identifier is recorded there.
4. If the principal identifier is found, the permission bits associated with that entry of the access-control-list are examined to see if writing is permitted.
5. If writing is permitted, the addressing descriptor of segment X, stored in AC_1, and the original offset k, are used to generate a write request inside the memory system.

We need one more mechanism to make this system usable. The set of processor registers must be augmented with a new, protected register which can contain the identifier of the principal currently responsible for the

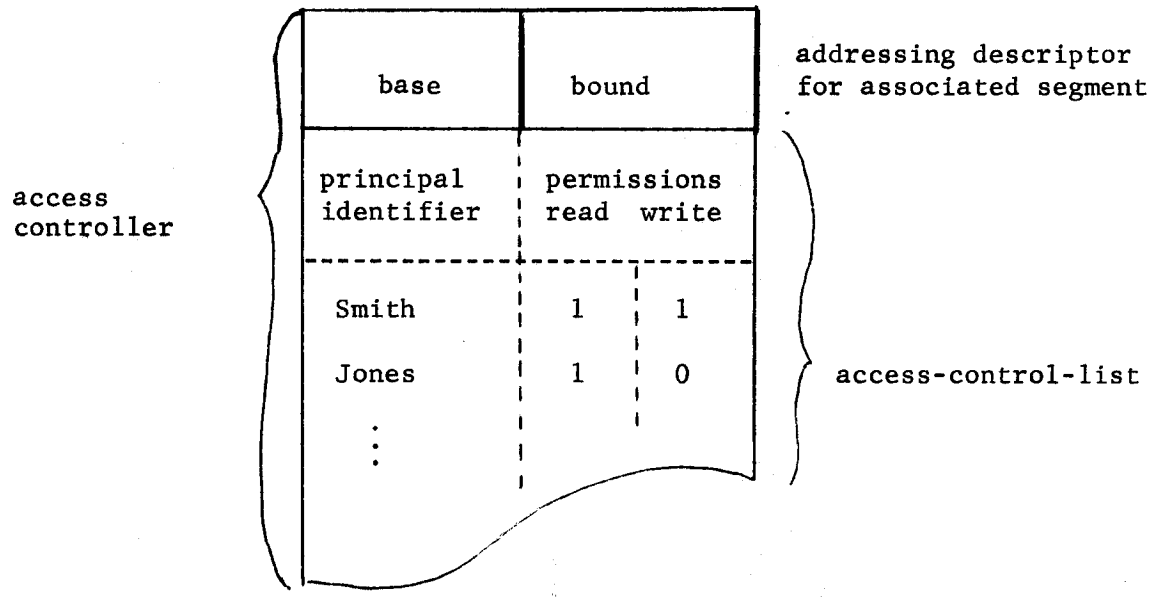


Figure 6-8 -- Conceptual model of an access controller. When a user attempts to refer to the segment associated with the access controller, the processor looks up his principal identifier in the access-control-list part. If found, the permissions associated with that entry of the access-control-list, together with the addressing descriptor, are used to complete the access.

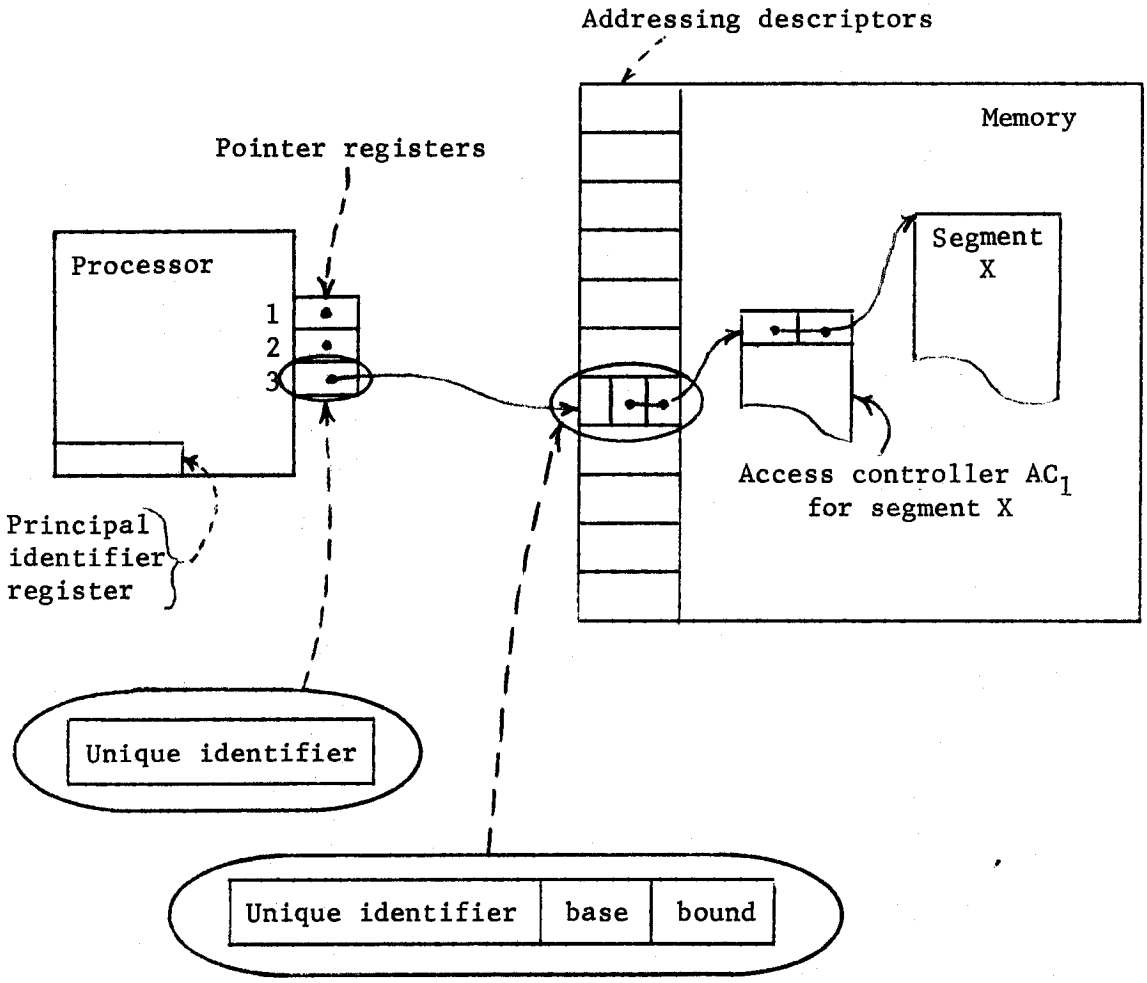


Figure 6-9 -- A revision of figure 6-5, with the addition of an access controller as an indirect address to be used on all references by the processor to the memory. Since the access controller contains permission bits, they no longer need appear in the processor registers, which have been renamed "pointer registers". Note that the privileged state bit of the processor has been replaced with a principal identifier register.

activity of the process, as shown in figure 6-9. (Without that change, step 3 would have been unimplementable.)

For example, we may have an organization like that of figure 6-10, which implements essentially the same pattern of sharing as did the capability system of figure 6-6. The crucial difference between these two figures is that in figure 6-10, all references to data are made indirectly via access controllers. We may note that the overall effect differs in several ways from that of the pure capability system described before;

1. The decision to allow access to segment X has known, auditable consequences: Doe cannot make a copy of the addressing descriptor of segment X since he does not have direct access to it. Thus, propagation of access has been eliminated: the pointer to X's access controller itself may be freely copied about and passed to anyone, but every use of the pointer must be via the access controller, which will prevent access by unauthorized principals.
2. The access-control-list directly implements the fourth step of the dynamic sharing protocol: verifying that the user of the capability is authorized by the creator of the object. In the capability system, that verification

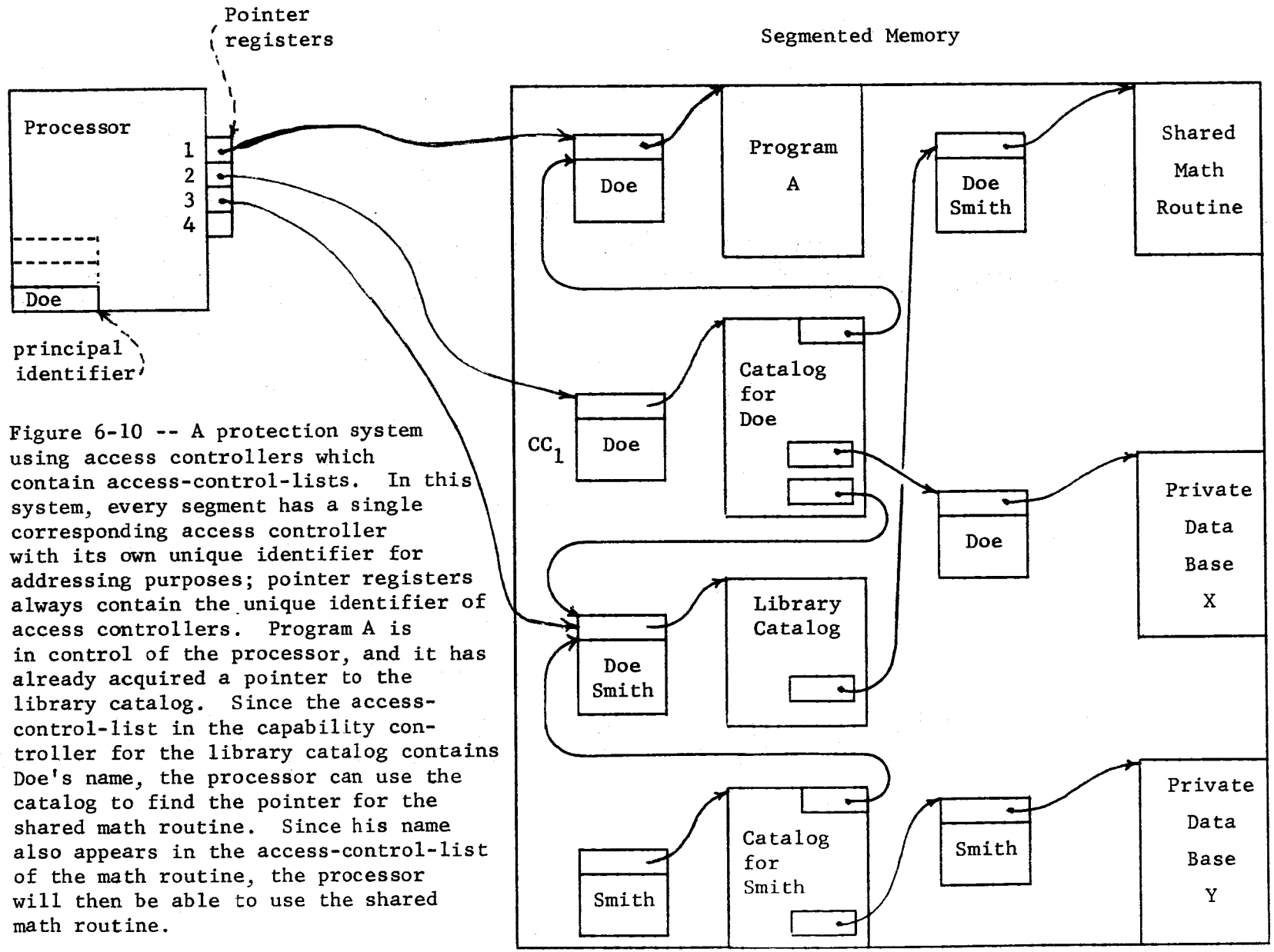


Figure 6-10 -- A protection system using access controllers which contain access-control-lists. In this system, every segment has a single corresponding access controller with its own unique identifier for addressing purposes; pointer registers always contain the unique identifier of access controllers. Program A is in control of the processor, and it has already acquired a pointer to the library catalog. Since the access-control-list in the capability controller for the library catalog contains Doe's name, the processor can use the catalog to find the pointer for the shared math routine. Since his name also appears in the access-control-list of the math routine, the processor will then be able to use the shared math routine.

5. The tight association between data organization and authorization has been broken. For example, although a catalog may be considered to "belong" to a particular user, the segments appearing in that catalog can have different access-control-lists. It follows that the grouping of segments for naming, searching, and archiving purposes can be independent of any desired grouping for protection purposes. Thus, in figure 6-10, a library catalog has been introduced.

It is also apparent that implementation, especially direct hardware implementation, of the access-control-list system could be quite a massive undertaking. We will later consider some strategies to simplify implementation with minimum compromise of functions, but first it will be helpful to introduce one more functional property: protection groups.

Protection groups

One final concept will round out our discussion of access-control-list systems. There are often cases where it would be inconvenient to explicitly list by name every individual who is to have access to a particular file, either because the list would be impossibly long, or else because the group of users which is to have access changes very frequently. To handle this situation, most access-control-list systems implement some form of protection groups, which are principals which may be used by more than one user. If the name of a protection group appears in an access-control-list, the intention is that all users who are members of that protection group be permitted access to that segment.

Methods of implementation of protection groups vary over a wide range of possibilities. A simple way to add them to the model of figures 6-9 and 6-10 is to extend the "principal-holding" register of the process so that it can hold two (or more) principal identifiers at once; one for a personal principal

identifier and one for each protection group that the user is a member of. Figure 6-10 shows this extension in dashed lines. In addition, we upgrade the access-control-list checker so that it searches for a match between any of the principal identifiers on the one hand and any entries of the access-control-list on the other.* Finally, there must also be a systematic way of controlling who is allowed to use those principals which represent protection group identifiers. We might imagine that for each protection group there is a person responsible for determining who shall be members of that protection group. That person maintains a segment containing a protection group list, that is, a list of the personal principal identifiers of all users authorized to use the protection group's principal identifier. This segment can be protected with an access controller, just as any other segment. When a user logs in, he can specify the list of principal identifiers he proposes to use. His right to use his personal principal identifier is authenticated, for example, by a password. His right to use the remaining principal identifiers can then be authenticated by the login procedure by looking up the now-authenticated personal identifier on each named protection group list. If everything checks, a process can safely be created and started with the specified list of principal identifiers.

Some implementation considerations

The model of a complete protection system which we have developed in figure 6-10 is one of many possible architectures, most of which have essentially identical functional properties; our choices among alternatives have

---

* If there is more than one match, and the multiple access-control-list entries specify different access permissions, some resolution strategy is needed. For example, the inclusive or of the individually specified access permissions might be granted.

been guided more by pedagogical considerations than by practical implementation issues. There are at least four key areas in which a direct implementation of figure 6-10 might encounter practical problems:

1. As proposed, every reference to an object in memory requires several steps: reference to a pointer register, indirect reference through an access controller including search of an access-control-list and, finally, access to the object itself via addressing descriptors. Not only are these steps serial, but several memory references are required, so high memory bandwidth would be needed.
2. An access-control-list search with multiple principal identifiers is likely to require a complex mechanism, or be slow, or both.
3. Allocation of space for access-control-lists which can change in length can be a formidable implementation problem.
4. Allocation and loading of pointer registers may become a burden.

The first of these problems can be minimized by providing, for each pointer register, a "shadow" register which is invisible to the process, as in figure 6-11. Whenever a pointer register containing the unique identifier of an access controller is first used, the shadow register is loaded with a copy of the addressing descriptor for the segment protected by the access controller. Subsequent references via that pointer register can proceed directly using the shadow register rather than indirectly through the access controller. One implication is a minor change in the revocability properties of an access-control-list: changing an access-control-list does not affect already loaded shadow registers of running processors. (One could restore complete revocability by clearing all shadow registers of all processors and restarting any current

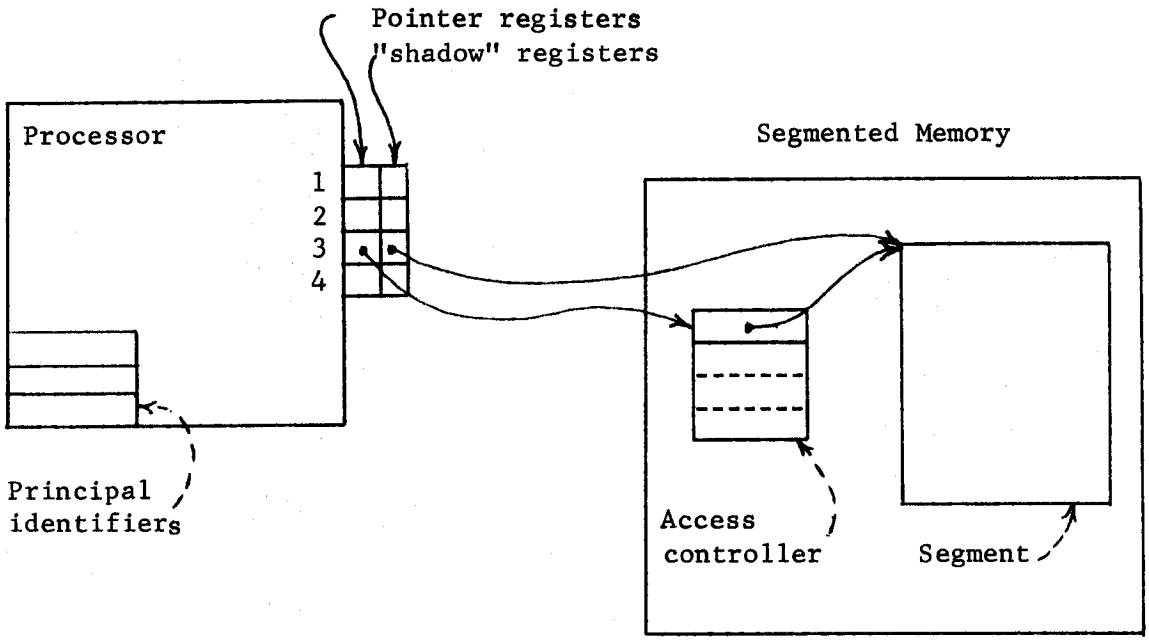


Figure 6-11 -- Use of "shadow" registers to speed up an access-control-list system. When a pointer register is loaded with a unique identifier, the shadow register is simultaneously loaded from the access controller to which the unique identifier refers. Storing of a pointer register means storing of the unique identifier only; the "shadow" register is never stored.

access-control-list searches. The next attempted use of a cleared shadow register would automatically trigger its reloading and thus a new access-control list check.)*

The second and third problems, allocation and search of access-control-lists, appear to require more of a compromise of functional properties. One might, for example, constrain all access-control-lists to contain, say, exactly five entries, to simplify the space allocation problem. One popular implementation allows only three entries on each access-control-list: the first is filled in with the personal principal identifier of the user who created the object being protected, the second with the principal identifier of the (single) protection group to which he belongs, and the third with the principal identifier of a universal protection group of which all users are members. The individual access permissions for these three entries are specified by the program creating the segment.**

Finally, for the access-control-list system to be practical, a fairly large number of pointer registers are needed, enough to address all or most of the objects needed by a single process during its lifetime. If the number of pointer registers were too small, they would need to be reloaded frequently, and since each reloading implies searching an access-control-list, costs would mount. One way to provide a large number of pointer registers is to place them not in the processor, but in a segment of memory, and provide only a single processor register which points to the segment. A separate pointer segment would be used for each process (or at least for each principal identifier).

---

* Note that the revocability problem is actually a familiar one: the shadow registers are an underlying hardware capability system which is being used to speed up the intrinsically slower access-control-list system. As mentioned several times before, all practical protection systems seem to include a capability system at the bottom.

** This strategy is implemented in software in TENEX [Bobrow et al., CACM 15, 3 (1972), 135-143] and UNIX [Ritchie and Thompson, CACM 17, 7 (1974) 365-375]. This idea seems to have originated in the University of California SDS-940 Time-Sharing System.

Authority to change access-control-lists

The access-control-list organization focuses one issue: control of who may modify the access-control-lists in access controllers. In the capability system, the corresponding consideration is diffuse: any program having a capability may make a copy and put that copy in a place where other programs, running in other processes, can make use (or further copies) of it. The access-control-list system was devised to provide more precise control of authority, so some mechanism for exercising that control is needed. The goal of any such mechanism is to provide, internal to the computer system, an authority structure which models the authority structure of the organization which uses the computer. Two different authority-controlling policies, with subtly different modelling abilities, have been implemented or proposed in existing systems: we name these two self control, and hierarchical control.

The simplest scheme is the one we shall name self control. With this scheme, we extend our earlier concept of access permission bits to include not just permission to read, write, and execute, but also permission to modify the access-control-list which contains the permission bits. Thus, in figure 6-12, we have a slightly more elaborate access controller, which by itself controls who may make modification to it. Suppose that the creation of a new segment is accompanied by the creation of an access controller which contains one initial entry in its access-control-list: an entry giving all permissions to the principal identifier associated with the creating process. The creator receives a pointer for the access controller he has just created, and can then adjust its access-control-list to contain any desired list of principal identifiers and permissions. Adjustment of the access-control-list requires a special "store" instruction (or supervisor entry, in a software implementation) which interprets its address as direct, rather than

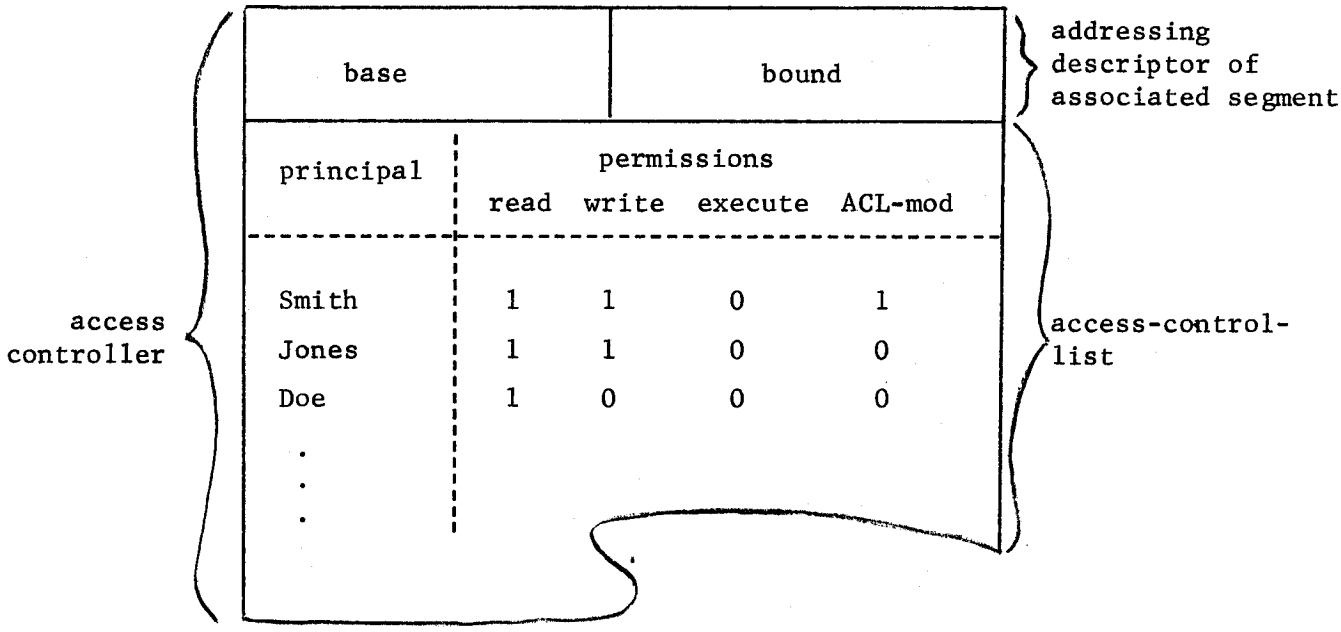


Figure 6-12 -- The access controller extended for self-contained control over modification of its access-control-list. In this example, user Smith has three permissions: to read and to write into the associated segment, and to make modifications to the access-control-list of this access controller. Jones cannot modify the access-control-list, even though he can read and write the segment described by this access controller.

indirect, but still performs the access-control-list checks before performing the store. This special instruction must also restrict the range of addresses it allows so as to prevent modifying the addressing descriptor stored in the access controller.

Probably the chief objection to the self-control approach is that it is so absolute: there is no provision for graceful changes of authority not anticipated by the creator of an access-control-list. For example, if in a commercial time-sharing system, a key member of a company's financial department is taken ill, there may be no way for his manager to authorize a co-worker to temporarily access a stored budget file unless the absent user had the foresight to set his access-control-lists just right. (Worse yet would be the possibility of accidentally creating an object for which its access controller permits access to no one--a kind of black hole.) To answer these objections, the hierarchical control scheme is sometimes used.

To obtain a hierarchical control scheme, we insist that whenever a new object is created, the creator specify some previously existing access controller which is to regulate future changes to the access-control-list in the access controller for the new object. The representation of an access controller must also be expanded to contain the addressing descriptor of the access controller which regulates it. In addition, the interpretation of the permission bit named "ACL-mod" is changed to apply to those access controllers which are hierarchically immediately below the access controller containing the permission bit. Thus, as in figure 6-13, all of the access controllers of the system will be arranged in a hierarchy, or tree structure, branching from the first access controller in the system, whose creation must be handled as a special case, since there is no previously existing access controller to regulate it. The hierarchical arrangement is now the pattern of access control, since a user with permission to modify

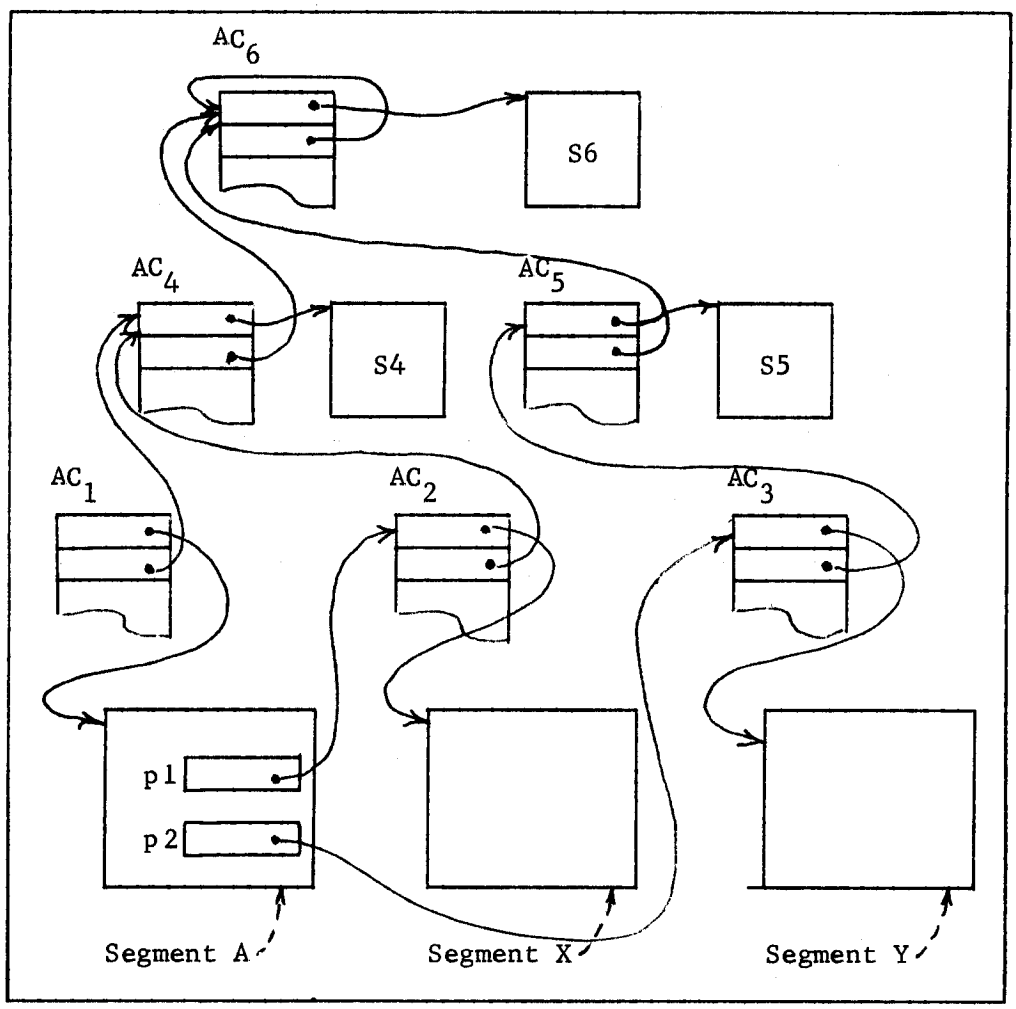


Figure 6-13 -- Hierarchical control of authority to modify access-control-lists. Authority to access segments A, X, and Y is controlled by access controllers AC_1, AC_2, and AC_3, respectively. Authority to modify AC_1 and AC_2 is in turn controlled by AC_4, while authority to modify AC_3 is controlled by AC_5. Authority to modify AC_4 and AC_5 is controlled by AC_6, which is the first access controller in the system. In this example, the authority to modify AC_6 is special-cased to be similar to the self-control scheme. Note that segments S4, S5, and S6 may be degenerate; AC_4, AC_5, and AC_6 may exist solely to control the authority to modify other access controllers. The meaning of the backpointer, say from AC_1 to AC_4, is that if a user attempts to modify the access-control-list of AC_1, the backpointer is followed, leading to AC_4. Only if the user's principal identifier is found in AC_4 (with appropriate permission) is the modification to AC_1 permitted. Segments A, X, and Y are arranged in an independent hierarchy of their own, with A superior to X and Y, by virtue of the pointer values p1 and p2 found in segment A. Note that p1 and p2 refer to their access controllers by unique identifiers.

access-control-lists may add his own principal identifier, with access-modifying permission to lower level access controllers, giving himself ability to change access-control-lists still further down the hierarchy. Thus access-modifying permission at any one node of the hierarchy permits the holder to grant himself access to anything in the entire subtree based on that node.

The contained hierarchical control scheme might be used in a time-sharing system as follows: the first access controller created is given an access-control-list naming one user, a system administrator. The system administrator creates several access controllers, for example one for each department in his company, and grants access-modifying permission in each to a department administrator. The department administrator then can create additional access controllers in a tree below the one for his department, perhaps for subdepartments or individual computer users in his department. These individual users can develop any pattern of sharing they wish, through the use of access-control-lists in access controllers, for the segments they create. In an emergency, however, the department administrator can intervene and modify any access-control-list in his department. Similarly, the system administrator can intervene in the case that a department administrator makes a mistake, or is not available.*

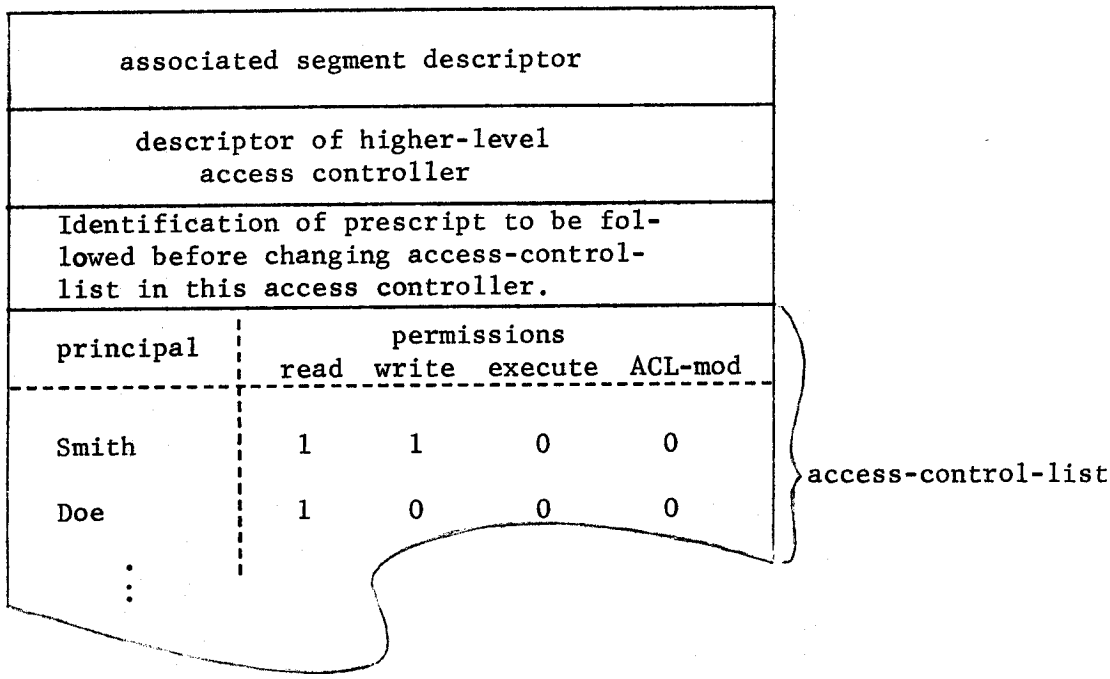
The contained hierarchical system is subject to the objection in our example that the system administrator and department administrators are too

---

* A variation which is occasionally encountered is the use of the segments controlled by access controllers higher in the hierarchical authority structure as catalogs for the segments below. This variation, if carried to the extreme, maps together the authority control hierarchy and the cataloguing hierarchy; some mechanical simplifications can be made, but trying to make dual use of a single hierarchy may lead to cataloguing strategies inappropriate for the data bases, or else to pressures to distort the desired authority structure. The Multics system [Daley and Neumann, AFIPS Conf. Proc. 27, Vol I, (1965), pp. 213-229], for example, uses this variation.

powerful; any hierarchical arrangement inevitably leads to concentration of authority at the higher levels of the hierarchy. A hierarchical arrangement of authority actually corresponds fairly well to the way many organizations operate, but the contained control method of modelling the hierarchy has one severe drawback: the use and possible abuse of higher-level authority is completely unchecked. In most societal organizations, higher-level authority exists, but there are also checks on it. Thus, for example, a savings bank manager may be able to authorize a withdrawal despite a lost passbook, but only after advertising its loss in the newspaper. A creditor may directly remove money from a debtor's bank account, but only with a court order. A manager may open an employee's locked file cabinet, but (in some organizations) only after temporarily obtaining the key from a security office, an action which leaves a record in the form of a logbook entry. A policeman may search your house, but only after obtaining a warrant. In each case, the authority to perform the operation exists, but the use of the authority is inseparably coupled with checks and balances which are designed to prevent abuse of the authority. In brief, the hierarchical control scheme provides for exercise of authority, but, as sketched so far, has no provision for preventing abuse of that authority.

One abuse-preventing strategy which has been suggested in various forms [Rotenberg, M.I.T. Ph.D. Thesis (1973); Daley and Neumann, AFIPS Conf. Proc. 27, Vol I, (1965), 213-229); Hsiao, ********** ; Hoffman, U. of C. (Berkeley) Ph.D. Thesis (1970)] is to add a field to an access controller, as in figure 6-14, which we may call the prescript field. Whenever an attempt is made to modify an access-control-list (either by a special store instruction or by a supervisor call, depending on the implementation) the access-modifying permission of the higher-level access controller regulating the access-control-list is checked as always. If the permission exists, the prescript field of the access-control-list which is about to be modified is then examined, and some action, depending on the value found, is automatically triggered. Figure 6-14



prescript identification	action triggered
0	no action
1	identifier of principal making change is logged
2	change is delayed 24 hours ("cooling-off" period)
3	change is delayed until some other principal attempts the same change ("buddy" system)
4	change is delayed until signal is received from some specific (system-designated) principal (court order)

Figure 6-14 -- The access controller, extended to provide prescripts which are intended to inhibit abuses of authority to modify access-control-lists. Some examples of possible prescripts are suggested in the table.

suggests some possible actions which might be triggered by the prescript value, and some external policies which can be modelled with the prescript scheme. The notion of a prescript, while apparently essential to a protection system which is intended to model typical real authority structures, has not been very well developed in existing or proposed computer systems. The particular prescript mechanism we have used for illustration of the concept can easily model only a small range of policies. More complex policies may be implementable by use of protected subsystems, a general escape mechanism described briefly in a later section.

Discretionary and non-discretionary controls

Our discussion of authorization and authority structures has so far rested on an unstated assumption: that the principal which creates a file or other object in a computer system has unquestioned authority to authorize access to it by other principals. Thus in the description of the self-control scheme, for example, it was suggested that a newly created object begins its existence with one entry in its access-control-list, giving all permissions to its creator.

We may characterize this control pattern as discretionary,* implying that the individual user may, at his own discretion, completely determine who is authorized to access the objects he creates. There are a variety of situations in which discretionary control may not be acceptable, and must be limited or prohibited. For example, the manager of a department developing a new product line may want to "compartmentalize" his department's use of the company computer system so as to insure that, even within the company, only those employees with "need-to-know" have access to information about the new product. The Vice President for Marketing may wish to compartmentalize all use of the company computer for calculating product prices, since pricing policy may be similarly sensitive. Neither manager

---

* A term suggested by Schell [**********].

may consider it acceptable that any individual employee within his department can abridge the compartmentalization decision merely by changing an access control list on an object he creates. Thus, the manager has a need to limit the use of discretionary controls by his employees. Any limits he imposes on authorization are controls which are out of the hands of his employees, and thus are viewed by them as non-discretionary. Non-discretionary controls may need to be imposed either in addition to or instead of discretionary controls. For example, the department manager may be prepared to allow his employees to adjust their access-control-lists any way they wish within the constraint that no one outside the department is ever given access. In that case, both non-discretionary and discretionary controls apply. Similar constraints are imposed in military security applications, in which not only are isolated compartments required, but also the concept of nested sensitivity levels (e.g., top secret, secret, and confidential) must be modeled in the authorization-granting mechanics of the computer system.

The key reason for interest in non-discretionary controls is not so much the threat of malicious insubordination as it is a need to safely use as tools complex and sophisticated programs created by suppliers who are not under the manager's control. Thus individual employees may use an APL interpreter, or a fast file sorting program supplied by a contract software house. They may run these borrowed programs as though they were their own, under their own principal identifier. If so, they are susceptible to a form of subversion known as the "Trojan Horse" attack [Branstad, Computer 6, 1 (1973), 43-47], in which the borrowed program, in addition to its advertised function, uses the authority of the borrower to modify access-control-lists, make illicit copies of files, or otherwise change the user's authorization intents. One way to prevent this kind of attack would be to forbid the use of borrowed programs, but for most organizations the requirement that

all programs be locally written would be an unbearable economic burden. The alternative is to provide non-discretionary controls which, since they cannot be changed by the user, cannot be changed by the programs he borrows either.

Complete elimination of discretionary controls is actually quite easy to accomplish. For example, in the self-control scheme, one could change the default initial value of the access-control-list of all newly created objects to be something specified by the user's manager. If this default initial value did not include permission for the user himself to modify the access-control-list, the user would have no discretionary control at all, and the manager would have complete control. A similar modification to the hierarchical control system can also be designed.

It is a harder task to arrange for the coexistence of discretionary and non-discretionary controls. Non-discretionary controls, may, for example, be implemented with a second access-control-list system operating in parallel with the first discretionary, control system, but using a different authority control pattern. Access to an object would be permitted only if both access-control-list systems agreed. Such an approach, using a full-bore access-control-list for non-discretionary controls may be more elaborate then necessary. Most designs that have so far appeared have taken advantage of a perceived property of non-discretionary controls: that they are usually relatively simple, such as "divide the activities of this system into six totally isolated compartments". It is then practical to provide a simplified access-control-list system to operate in parallel with the discretionary control machinery. For example one would add to an access controller a field indicating the compartment or compartments from which that object contains information (the access-control-list), and would add to a process state a second principal identifier indicating the compartments

which the responsible principal is authorized to access. The processor would then require a match between these two fields (the non-discretionary control) in addition to a match between original principal identifier and the regular access-control-list (the discretionary control). Any new object created by a process would automatically receive in the compartment field of its access controller the compartment designations of the process state.

An interesting requirement for a non-discretionary control system which implements isolated compartments arises in the case in which a principal may be authorized to access two or more compartments simultaneously, and some data objects may be labeled as being simultaneously in two or more compartments (e.g., pricing data for a new product may be labeled as requiring access to the "pricing policy" compartment as well as the "new product line" compartment.) In such a case it would seem reasonable that before permitting reading of data from an object the control mechanics should require that the set of compartments of the object being referenced be a subset of the compartments to which the accessor is authorized. However, a more stringent interpretation is required for permission to write, if confinement of "Trojan Horse" programs is to be accomplished. Confinement requires that the process be constrained to write only into objects that have a compartment set identical to that of the process itself. If such a restriction were not enforced, a "Trojan Horse" program could, upon reading data labeled for both the "pricing policy" and "new product line" compartments, make a copy of part of it in a segment labeled only "pricing policy", thereby compromising the "new product line" compartment boundary. A similar set of restrictions on writing can be expressed for sensitivity levels; a complete and systematic analysis in the military security context was developed by Weissman [AFIPS Conf. Proc. 35 (1969) 119-133]. Weissman suggested that the problem be solved by automatically labeling any object written with the compartment labels needed to permit

writing, a strategy he named the "high-water-mark". Lipner [**********] has suggested an alternative strategy of declaring attempts to write into objects without the necessary compartment labels to be errors which cause these programs to stop. In either case, a system which correctly implements this restriction may be said to have the confinement property, a term used in mathematical modelling of security systems.*

Protecting objects other than segments

So far, it has been useful to frame our discussion of protection in terms of protecting segments, which are basically arbitrary-sized units of memory with no internal structure. Capabilities and access-control-lists are just as well adapted to protecting other kinds of objects, also. In figure 6-9, access controllers themselves were treated as system-implemented objects, and in figure 6-13 were protected by other access controllers. There are many other kinds of objects which are provided by the hardware and software of computer systems for which protection is appropriate. To protect an object other than a segment, one must first establish what are the kinds of operations which can be performed on the object, and then work out an appropriate set of permissions to perform those operations. For a data segment, the separately controllable operations we have used in our examples are those of reading, writing, and executing the contents of the segment.

For an example of a different kind of system-implemented object, suppose that the processor is augmented with instructions that manipulate the contents of a segment as a first-in, first-out queue. These instructions might interpret

---

* It should be noted that complete confinement of a program in a shared system is a very difficult or perhaps impossible task to accomplish, since the program may be able to signal to other users by subtle strategies other than writing into shared segments. For example, the program may intentionally vary its paging rate in a way observable by users outside the compartment, or it may unexpectedly stop, causing its user to go back to the original author for help, thereby revealing the fact that it stopped. Lampson [CACM 16, 10 (1973),613-615] and Rotenberg [Ph.D. thesis, M.I.T. Project MAC-TR-115 (1973)] have explored this problem in some depth.

the first few words of the segment as pointers or counters, and the remainder as a storage area for items placed in the queue. One might provide two special instructions, "enqueue" and "dequeue", which add to and remove from the queue, respectively. Typically, both of these operations would need to both read and write various parts of the segment being used as a queue.

As described so far, the enqueue and dequeue instructions would indiscriminately treat any segment as a queue, given only that the process issuing the instruction had a capability permitting reading and writing the segment. One could not set up a segment such that some users could only enqueue messages, but not be able to dequeue--or even directly read--messages left by others. Such a distinction between queues and other segments can be achieved by introducing the concept of <u>type</u> in the protection system.

Consider, for example, the capability system, as in figure 6-6. Suppose we add to a capability an extra field, which we will name the <u>type</u> field. This field will have the value 1 if the object described by the capability is an ordinary segment, and the value 2 if the object is to be considered a queue. The protection descriptor registers are also expanded to contain a type field. We also add to the processor the knowledge of which types are suitable as operands for each instruction. Thus, the special instructions for manipulating queues require that the operand capability have type field 2, while all other instructions require an operand capability with type field 1. Further, the interpretation of the permission bits can be different for the queue type and the segment type. For the queue type, one might use the first permission bit to control use of the enqueue instruction and the second permission bit for the dequeue instruction. Finally, we should also expand the "create" operation to permit specification of the type of object which should be created.

Clearly, one could extend the notion of type beyond segments and queues; any hardware interpreted data structure could be similarly distin-

guished and protected from misuse. The concept of type-extension is not restricted to capability systems; in an access-control-list system one could place the type field in the access controller and require that the processor present to the memory, along with each operand address, the required type and permission bits for the operation being performed. Table 6-I lists some typical system-implemented objects and the kinds of operations one might selectively permit. This table could be extended to include other objects which are basically interpreted data structures, such as accounts or catalogues.

Finally, one may wish to extend dynamically the range of objects protected by the protection system. Such a goal might be achieved by making the type field large enough to contain an additional unique identifier, and allowing for software interpretation of the access to typed objects. This observation brings us to the subject of user-specified controls on sharing, and the implementation of protected objects and protected subsystems. We shall not attempt to examine this topic in depth, but rather only enough to learn what problems are encountered.

Protected objects and subsystems

Both the capability system and the access-control-list system allow controlled sharing among users of the objects implemented by the system. Several common patterns of use, such as reading, writing, or running as a program, can be independently controlled. While a great improvement over "all-or-nothing" sharing, this sort of controlled sharing has two important limitations. The first limitation is that only those access restrictions provided by the standard system facilities can be enforced. It is easy to imagine many cases where the standard controls are not sufficient. For example, an instructor who maintains the grade records for a subject in a segment on an interactive system may wish to allow each student to read his own grades to verify correct recording of each assignment, but not the grades of other students, and to allow each student to generate the distribution of all grades for each

object	separately permittable operations
data segment	read data write data load capability store capability
access controller	read access-control-list modify names appearing on an access-control-list modify permissions in access-control-list entries destroy object protected by this capability controller
FIFO message queue	enqueue a message dequeue a message examine queue contents without dequeueing
input/output device	read data write data issue device-control commands
removable recording medium (e.g., magnetic tape reel)	read data over write data write data in new area

Table 6-I -- Typical system-provided objects which may be protected by a capability system or an access-control-list system.

assignment. Implementing such controls within systems providing controlled sharing of the sort discussed in the last few sections would be awkward, requiring at least the creation of a separate segment for each student and for the distributions. If, in addition, the instructor wishes his assistant to enter new grades, but wants to guarantee that each grade entered cannot be changed later without the instructor's specific approval, we have a situation that is beyond the ability of the mechanisms so far described for controlling sharing.

The second limitation concerns users who borrow programs constructed by other users. Execution of a borrowed program in the borrower's process can present a real danger to the borrower, for the borrowed program can exercise all the capabilities in the protection domain of the borrower. If the borrowed program malfunctions, or is malicious, it can damage or release information of the borrower. Thus, a user must have a certain amount of faith in the provider of a program, before he executes the program in his own process.

The key to removing these limitations is the notion of a protected subsystem. A protected subsystem is a collection of program and data segments that are encapsulated so that other executing programs cannot read or write the program and data segments, and cannot disrupt the intended operation of the component programs, but can invoke the programs by calling designated entry points. The encapsulated data segments are the <u>protected objects</u>. To remove the first limitation, programs in a protected subsystem can act as caretakers for the protected objects and interpretively enforce arbitrarily complex controls on access to them. Programs outside the protected subsystem are allowed to manipulate the protected objects only by invoking the caretaker programs. Algorithms in these programs may judge the propriety of the requested access based on information provided by the system describing the circumstances of invocation, and may even record each access request in some way in some protected objects. For example, the protected subsystem shown in figure 6-15 implements the grade keeping system

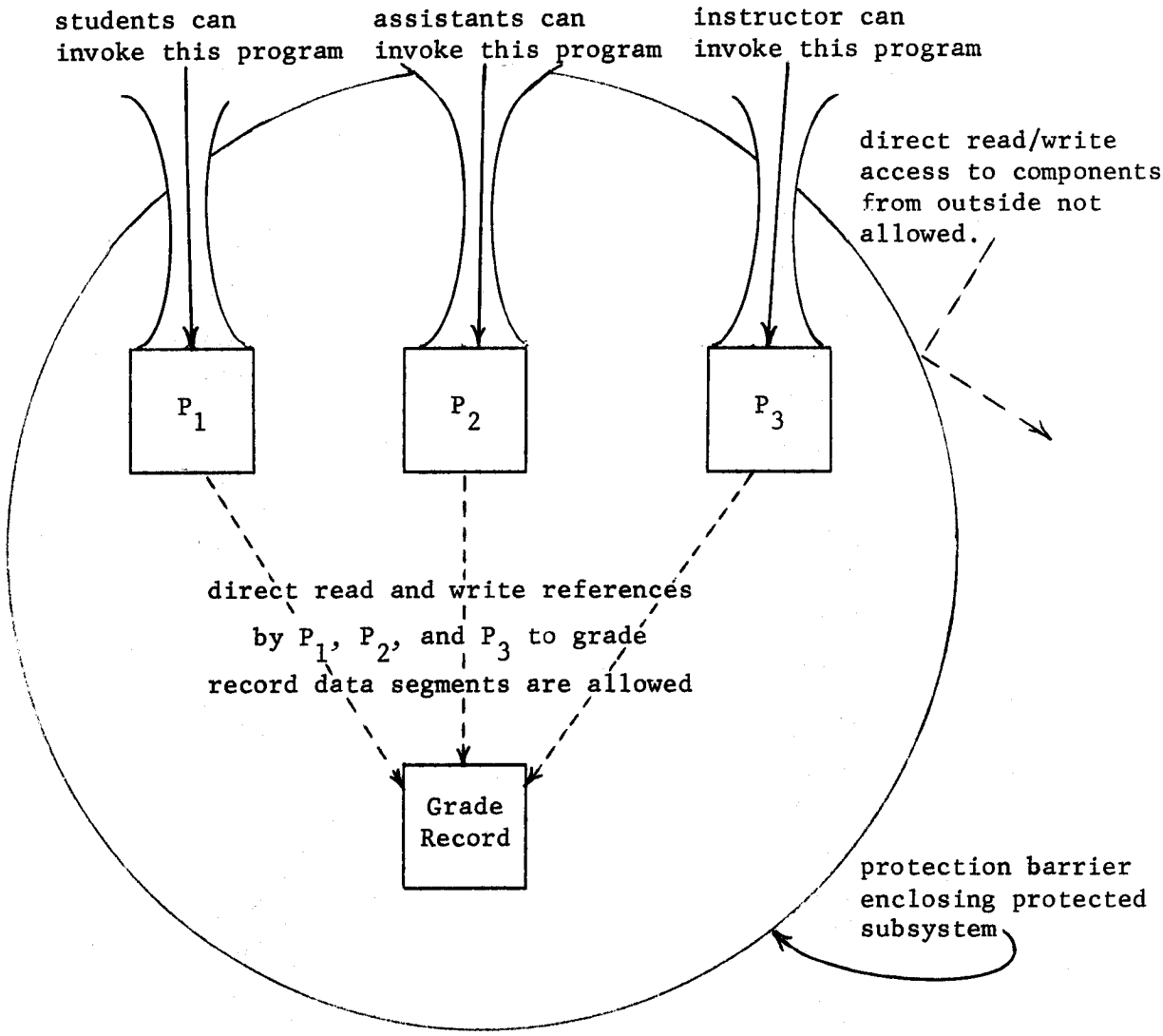


Figure 6-15 -- A protected subsystem to implement the grade keeping system described in the text. P_1, which can be invoked by all students in the subject, is programmed to return the caller's grade for a particular assignment or the distribution of all grades for an assignment. P_2, which can be invoked by the teaching assistants for the subject, is programmed to allow the addition of new grades to the record but to prevent changing a grade once it is entered. P_3, which can be invoked only by the instructor, is programmed to read or write on request any data in the grade record.

discussed above. Clearly, any access constraints which can be specified in an algorithm can be implemented in this fashion. Giving users the ability to construct protected subsystems out of their own program and data segments allows users to provide arbitrary controls on sharing.

If programs inside protected subsystems also can invoke programs outside (perhaps in another protected subsystem) without compromising the security of the former, then we achieve the ability to plug together multiple protected subsystems to perform a computation. We also find a way around the second limitation of simple controlled sharing, the borrowed program problem. The access environment in which the programs of a user normally execute on his behalf could be set up as a protected subsystem. Then the user could arrange for programs borrowed from other users to execute outside of this home protected subsystem. In this way the borrowed programs could be invoked without giving them access to all the programs and data of the borrower. If the borrowed program is malicious or malfunctions then the damage it can do is thereby limited. The lending user could also encapsulate the lent program complex in a protected subsystem of its own and thus insulate it from the programs of the borrower. *

The notion of protected subsystems, then, provides mutual protection for multiple program complexes cooperating in the same computation, and removes the two limitations of facilities providing simple controlled sharing. It is clear from the description of protected subsystems that each must operate in its own protection domain. Implementing protected subsystems requires providing mechanisms that allow the association of more than one protection domain with a computation and providing means for changing from one protection domain to another as control passes from one protected sub-

---

* Note that encapsulation of a borrowed program in a protected subsystem is done with a different goal than confinement of a borrowed program within a compartment. Encapsulation may be used to limit the access a borrowed program has to the borrower's data. Confinement is intended to allow a borrowed program to have access to data, but insure that the program cannot release the information.

system to another. The design must insure that one protected subsystem cannot interfere in any way with the correct operation of another involved in the same computation.

We note in passing that the supervisor in most computer systems is a good example of a protected subsystem. If general facilities are provided for supporting user-constructed protected subsystems, then these mechanisms can be applied to protect the supervisor from user programs as well. The resulting uniformity of mechanism is consistent with the design principle of simplicity.

In order to implement protected subsystems, then, there must be a way of associating multiple protection domains with a single computation. One way would be to use a separate process, each with its own protection domain, for each protected subsystem, a notion discussed by Lampson [Proc. 5th Princiton Conf. Info. Science and Systems (1971),437-443]. A computation involving multiple protected subsystems thus would require multiple cooperating processes. The invocation of one protected subsystem by another, and any response in the reverse direction, would be achieved using the interprocess communication facilities of the system [Hanson, CACM 13, 4 (1970), 238-250]. A multiprocess implementation, while conceptually straightforward, tends to be awkward and inefficient in practice. Furthermore, it tends to obscure important features of the required mechanisms. Unless there is an inherent need for the protected subsystems in a computation to execute in parallel, a single process implementation seems more natural. Such an implementation would require the association of multiple protection domains with a single process, a strategy explored in detail by Schroeder [Ph.D. thesis, M.I.T. (1972)], Rotenberg [Ph.D. thesis, M.I.T. (1974)], Jones [Ph.D. thesis, Carnegie-Mellon University (1974)], and Needham [AFIPS Conf. Proc. 41 (1972), 571-578]. In this case, communication among protected subsystems could be via interprocedure call and return operations.

The actual mechanics of implementation of protected objects and subsystems are beyond the scope of this tutorial, since there is not yet any widespread agreement on which mechanisms are fundamental, but it is useful to quickly sketch the considerations which those mechanisms must handle.

- the principle of "separation of privilege" is basic to the idea that the internal structure of some data objects is accessible to process A, but only when the process is executing in program B. If, for example, the protection system requires possession of two capabilities before it allows access to the internal contents of some objects, then the program responsible for maintenance of the objects can hold one of the capabilities while the user of the program can catalog the other. Morris [CACM 16,1 (1973), 15-21] has proposed an elegant semantics for separation of privilege in which the second capability is known as a seal. In terms of the earlier discussion of types, the type field of a protected object contains a seal which is unique to the protected subsystem; access to the internal structure of an object can be achieved only by presenting the original seal capability as well as the capability for the object itself.
- The switching of protection domains within a single process must be intimately entangled with the mechanisms that provide for dynamic activation records and static (own) variable storage, since both the activation records and static storage of one protection domain must be distinct from that of another.
- The passing of argument values between domains must be carefully controlled, to insure that the called domain will be able to access its arguments without violating its own protection intentions. Schroeder [M.I.T. Ph.D. Thesis (1972)] explored argument passing in depth from

the access-control-list point of view, while Jones [Carnegie-Mellon Ph.D. Thesis (1973)] explored the same topic in the capability framework.

The reader interested in learning about the mechanics of protected objects and subsystems in detail is referred to the literature mentioned above and in the suggestions for further reading. The area is in a state of "advanced development", in which several ideas have been tried out experimentally, but there is not yet much agreement on which mechanisms are fundamental. For this reason, the subject is best explored by case study.

III: The State of the Art and Current Research Directions

The state of the art

Until quite recently, the protection of computer-stored information has been given relatively low priority both by major computer manufacturers and by a majority of their customers. Although research time-sharing systems using base-and-bound registers appeared as early as 1960 [ref. **********], and Burroughs marketed a descriptor-based system in 1961, those early features were directed more toward preventing accidents than toward providing foolproof protection. Thus in the design of the IBM System 360, which appeared in 1964 [Amdahl, et al., IBM J. of R. & D. 8, 2 (1964), 87-101], the only protection mechanisms consisted of a privileged state and a protection key scheme which prevented writing in those blocks of memory allocated to other users. The early IBM software used these mechanisms only to the minimum extent necessary to allow accident-free multiprogramming. Not until 1970 did fetch protect (the ability to prevent one user from reading primary memory allocated to another user) become a standard feature of the IBM architecture [ref **********]. As of 1974, descriptor-based architectures, which open the door to the more sophisticated protection mechanisms described in section II, have become common, but not universal, in commercially marketed systems and in most manufacturer's plans for forthcoming product lines. Examples of commercially available systems which are descriptor-based are the IBM System 370 models which support virtual memory [ref. **********] the Univac (formerly RCA) system 7 [Oppenheimer and Weizer, **********], the Honeywell 6180 [Corbató et al., AFIPS Conf. Proc. 40 (1972), 571-583], the Control Data Corporation Star-100 (not yet delivered) [Holland, et al., IEEE Comp. Conf. (1971), 55-56], the Burroughs B5700/6700 [Hauck,

et al., AFIPS Conf. Proc. 32 (1968), 245-251], the Hitachi 8800 [ref. **** ******]. and the Digital Equipment Corporation PDP-11 model 45 [ref. ***** *****]. On the other hand, architectural provision for controlled sharing of information, whether by software or hardware, is still the exception rather than the rule.

In the decade between 1964 and 1974, several protection architectures were explored as research and development projects, usually starting with a computer that provided only a privileged mode, adding minor hardware features, and interpreting with software the desired protection architecture. Among these were the Compatible Time-Sharing System of the Massachusetts Institute of Technology, which implemented in 1961 user authentication, compartments, and, in 1964, shared files with permission lists [CTSS Programmers' Guide, 1st ed. (1963) and 2nd ed. (1965), M.I.T. Press]. The Advanced Development Prototype (ADEPT) System of the System Development Corporation, in 1967, implemented in software on an IBM system 360, a model of the U.S. military security system, complete with clearance levels, compartments, need-to-know, and centralized authority control [Weissman, AFIPS Conf. Proc. 35 (1969) 119-133]. At about the same time the IBM Cambridge Scientific Center released an operating system, named CP/67 and later marketed under the name VM/370, that used descriptor-based protection hardware to implement virtual system 360 computers, using a single system 360 model 67 [Meyer and Seawright, IBM Sys. J. 9 (1970), 199-218]. In 1969, the University of California (at Berkeley) CAL system implemented a software-interpreted capability system on a Control Data 6400 computer. Also in 1969, the Multics system, a joint project of the Massachusetts Institute of Technology and Honeywell, implemented in software and hardware a complete descriptor-based access-control-list system with hierarchical control of authentication on a Honeywell 645 computer system [Saltzer, CACM 17, 7 (1974), 388-402, Graham, CACM 11, 5 (1968), 365-369].

Based on the plans for Multics, the Hitachi Central Research Laboratory implemented a simplified descriptor-based system with hardware implemented ordered domains (rings of protection) on the HITAC 5020E computer in 1968 [ref ***** *****]. In 1973, a hardware version of the rings of protection idea was implemented for Multics in the Honeywell 6180 computer system [Schroeder and Saltzer, CACM 15, 3 (1972) 157-170]. At about the same time, the Plessey Corporation announced a telephone switching computer system, the Plessey 250, which is based on a capability architecture. Today, there are several research organizations experimenting with more elaborate protection architectures, such as capabilities and user-defined protected objects, taking advantage of the flexibility available from microprogramming and from rapidly dropping hardware costs. Some of these projects were described in a series of sessions at the 1974 National Computer Conference [AFIPS Conf. Proc. 43 (1974), 973-999].

Current research directions

There are several different areas in which research on protecting information remains to be done. In addition to continued experimenting with different protection architectures to try to discover simpler, yet effective, implementations of protection goals, there are at least five major areas receiving attention today:

- Certification of the correctness of a protection system design and implementation.
- Invulnerability to single faults.
- Constraints on use of information after release.
- Encipherment of information with secret keys.
- Improved authentication mechanisms.

These five areas are discussed in turn.

A research problem which is attracting much attention today is that of certifying the correctness of the design and implementation of

hardware and software protection mechanisms. There are actually several subproblems in this area:

1. One must have a precise model of the protection goals of a system, so that the design and implementation can be measured against that model. When the goal is that independent users should be completely isolated, the model is straightforward and the mechanisms of the virtual machine are relatively easy to match with the model. However, when controlled sharing of information is desired, the model is much less clear and the attempt to clarify it generates many unsuspected questions of policy. Even attempts to model the well-documented military security system have led to surprisingly complex formulations and have exposed formidable implementation problems [Weissman, AFIPS Conf. Proc. 35 (1969) 119-133, Bell and LaPadula, MITRE Corp. MTR-2547, Vol 1, (1973)].
2. Given a precise model of the protection goals of a system, and a working implementation of that system, the next challenge is to somehow verify that the presented implementation actually supplies the claimed functions. Since protection functions are usually a kind of negative specification, testing by sample cases provides almost no information. One proposed approach is using proofs of correctness to establish formally that a system is correctly implemented. Work in this area consists primarily of attempts to extend methods of proving assertions about programs to cover the constructs typically encountered in operating systems.
3. Most current day systems present the user with a rather intricate interface for specifying his protection needs. The result is that the user has trouble figuring out how to make the specification, and in verifying that he has requested the right thing. What is needed are user interfaces which more closely match the mental models people have of information protection.

4. In most operating systems, there is an unreasonably large quantity of "system" software which runs without protection contraints. The reasons are many: hoped-for higher efficiency, historical accident, misunderstood design, and inadequate hardware support. The usual result is that the essential mechanisms which implement protection are thoroughly tangled with a much larger body of mechanisms, making certification an impossibly complex task. In any case, there has not yet been established a minimum set of protected supervisor functions--a protected kernel--for a full-scale modern operating system.

These four subproblems taken together are all part of the general research area of certifying correctness of protection system design and implementation.

An area of vulnerability of most modern operating systems is their reaction in the face of hardware failures. Failures which cause the system to misbehave are usually easily detected and, with experience, automatic recovery can be provided. Far more serious are failures which result in an undetected disabling of the protection mechanisms. Since routine use of the system may not include attempts to access things which should not be accessible, failures in access-checking circuitry may go unnoticed indefinitely. There is a challenging, but probably solvable, problem involved in guaranteeing invulnerability of the protection mechanisms in the face of all single hardware failures. Molho [AFIPS Conf. Proc. 36 (1970), 135-141] explored this topic in the IBM System 360 model 50 computer, and made several suggestions for improvement. Fabry [Comm. ACM 16, 11 (1973), 659-668] has described an experimental system in which all operating system decisions which could affect protection are repeated by independent hardware and software.

Another area of research is on constraining the use to which information may be put, even after release to an executing program. In part I,

we described such constraints as a fourth level of desired function. For example, one might wish to "tag" a file with a notation that any process reading that file is to be forever after restricted from printing output on remote terminals located outside a headquarters building. For this restriction to be complete, it should propagate with all results created by the process, and into other files it writes into. Information use restrictions such as these are common in legal agreements (as in the agreement between a taxpayer and a tax return preparing service) and the unsolved problem is to see if there are corresponding mechanisms for computer systems which could help enforce (or detect violations of) such agreements. Rotenberg explored this topic in depth in his Ph.D. thesis (M.I.T., 1973) and proposed a privacy restriction processor to aid enforcement.

A potentially powerful technique for protecting information is to encipher it, using a key known only to authorized accessors of the information. (Thus encipherment is basically a ticket-oriented system.) One problem with encipherment strategies is developing techniques for communicating keys to authorized users. If this communication is done internal to the computer system, then schemes for protecting the keys must be devised. Strategies for securing multinode computer communication networks using encipherment are a topic of current research; Branstad has summarized the state of the art [AIAA paper 73-427, (1973)]. Another research problem is development of encipherment techniques (sometimes called privacy transforamtions) for random access to data. Most well-understood enciphering techniques operate sequentially on long bit streams (as found in point-to-point communications, for example.) Techniques for enciphering and deciphering small, randomly selected groups of bits such as a single word or byte of a file have been proposed, but the effort required to cryptanalyze such encipherments (that is, the work

factor) is not easily determined, and is still a subject for research. A block enciphering system based on a scheme suggested by Feistel was developed at the IBM T. J. Watson Research Laboratory by Smith, Notz, and Osseck [Proc. ACM 25th Nat. Conf., Vol. II (1972), 282-297].

Research in encipherment encounters a practice of military classification of most work. Thus, since World War II only three papers with significant contributions have appeared in the open literature [Shannon, BSTJ 28, 4 (1949), 656-715; Baran, RAND Corp. report RM-3765-PR Vol. 9 (1964); and Feistel, IBM Research Report RC-2427 (1970)]; most other papers have only updated, reexplained, or rearranged concepts published many years earlier.

Finally, spurred by need for better credit and check-cashing authentication, there is considerable research and development effort going into better authentication mechanisms. Many of these strategies are based on attempts to measure some combination of personal attributes, for example, the dynamics of a handwritten signature or the rhythm of keyboard typing. Others are directed toward developing difficult-to-duplicate machine-readable identification cards.

Concluding remarks

In reviewing the extent to which protection mechanisms are systematically understood (which is not a large extent) and the current state of the art, one cannot but help draw a parallel between current protection inventions and the first mass-produced computers of the 1950's. At that time, by virtue of experience and strongly developed intuition, there was a confidence that the architectures being delivered as products were sufficiently complete to be useful, and it turned out that they were. Yet, it was rapidly established that matching a problem statement to the architecture (e.g., programming) was a major effort and one whose magnitude was quite sensitive to the exact architecture chosen. In a parallel way, the matching of a set of protection goals to a particular protection architecture by setting the bits and locations of access-control-lists or capabilities or by

devising protected subsystems is a matter of programming the architecture. Following the parallel, it is not surprising that users of the current first crop of protection systems have found them relatively clumsy to program and not especially well matched to their image of the problem to be solved, even though they be logically complete. As in the case of all programming systems it will be necessary for protection systems to be used and analyzed, and for those users to propose different, possibly orthogonal, views of the necessary and sufficient semantics to clearly support information protection.

Acknowledgements

(to be supplied in a later draft)

Suggestions for further reading

The following short bibliography has been selected to direct the reader to the most useful, up-to-date, and significant materials currently available. Many of these readings have been collected and reprinted in the book by L. J. Hoffman, reference 6.35. The four bibliographies listed as 6.31 - 6.35 provide access to a vast collection of related publications.

Privacy and the impact of computers:

6.1 Westin, A.F., Privacy and Freedom, Atheneum, New York, 1967.

6.2 Miller, A., "The Assault on Privacy, University of Michigan Press, 1971, (Signet Paperback W4934, 1972).

6.3 Beardsley, C.W., "Is your computer insecure?" IEEE Spectrum 9, 1 (January, 1972), pp. 67-78.

6.4 Parker, D.B., Nycum, S., and Oura, S.S., "Computer Abuse," Stanford Research Institute, November, 1973, final report of SRI Project ISU 2501.

Case studies of protection systems:

6.5 Saltzer, J.H., "Protection and the Control of Information Sharing in Multics," Comm. ACM 17, 7 (July, 1974), pp. 388-402.

6.6 Gray, J., Lampson, B., Lindsay, B., and Sturgis, H., "The Control Structure of an Operating System," (Unpublished description of the University of California (Berkeley) CAL operating system, 1972).

6.7 Needham, R.M., "Protection systems and protection implementations," AFIPS Conf. Proc. 41, Vol. I, (1972 FJCC), pp. 571-578.

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

6.9 Molho, L.M., "Hardware aspects of secure computing," AFIPS Conf. Proc. 36, (1970 SJCC), pp. 135-141.

6.10 Schroeder, M.D., and Saltzer, J.H., "A Hardware Architecture for implementing Protection Rings," Comm. ACM 15, 3 (March, 1972), pp. 157-170.

6.11 Sturgis, H.E., "A Postmortem for a Time Sharing System," Ph.D. thesis, University of California at Berkeley, 1973. (Also available as Xerox Palo Alto Research Center Technical Report CSL 74-1.)

6.12 Fabry, R.S., "Dynamic Verification of Operating System Decisions," Comm. ACM 16, 11, (November, 1973), pp. 659-668.

Protected objects and protected subsystems.

6.13 Schroeder, M.D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," Ph.D. Thesis, M.I.T. Department of Electrical Engineering, September, 1972. Also available as Project MAC Technical Report, TR-104.

6.14 Jones, Anita K., "Protection in Programmed Systems," Carnegie-Mellon University, Department of Computer Science, Ph.D. Thesis, June, 1973.

6.15 Rotenberg, L., "Making Computers Keep Secrets," Ph.D. Thesis, M.I.T. Department of Electrical Engineering, June, 1973. Also available as Project MAC Technical Report, TR-115, January, 1974.

6.16 Morris, J.H., Jr., "Protection in Programming Languages," Comm. ACM 16, 1 (January, 1973), pp. 15-21.

6.17 Fabry, R.S., "The Case for Capability-based Computers," Comm. ACM 17, 7 (July, 1974), pp. 403-412.

6.18 Lampson, B.W., "Protection," Proc. 5th Princeton Conference on Information Sciences and Systems," (March, 1971), pp. 437-443.

6.19 Redell, D.D., "Naming and Protection in Extendible Operating Systems," Ph.D. thesis, University of California at Berkeley, September, 1974.

Protection by encipherment:

6.20 Smith, J.L., Notz, W.A., and Osseck, P.R., "An Experimental Application for Cryptography to a Remotely Accessed Data System," Proc. ACM 25th National Conf., August, 1972, pp. 282-297.

6.21 Girsdansky, M.B., "Cryptology, The Computer, and Data Privacy," Computers and Automation, April, 1972, pp. 12-19.

6.22 Feistel, H., "Cryptographic coding for databank privacy," IBM Research Report RC 2827, March 18, 1970.

6.23 Mellen, G.E., "Cryptology, computers, and common sense," AFIPS Conf. Proc. 42, (1973 NCC), June, 1973, pp. 569-579.

6.24 Kahn, D., The Codebreakers, Macmillan, New York, 1967.

6.25 Branstad, D.K., "Security Aspects of Computer Networks," AIAA Paper No. 73-427, AIAA Computer Network Systems Conference, Huntsville, Alabama, April 16-18, 1973.

Applications to military security; non-discretionary controls.

6.26 Ware, W., et al., Security Controls for Computer Systems, Rand Corporation Technical Report R-609, 1970. (Classified confidential).

6.27 Anderson, James P., Computer Security Technology Planning Study, United States Air Force Electronics Systems Division Technical Report ESD-TR-73-51, October, 1972), Vol. I and II. (unclassified).

6.28 Schiller, W.L., "Design of a Security Kernel for the PDP-11/45," MITRE Technical Report MTR-2709, June 30, 1973.

Comprehensive discussions of all aspects of computer security:

6.29 Martin, James, Security, Accuracy, and Privacy in Computer Systems, Prentice-Hall, 1973, Englewood Cliffs, New Jersey.

6.30 Anderson, James, "
Advances in Computers 12, Academic Press, 1973, pp. 1-

Bibliographies and Collections on protection and privacy:

6.31 Bergart, J.G., Denicoff, M., and Hsiao, D.K., "An Annotated and Cross-Referenced Bibliography on Computer Security and Access Control in Computer Systems," Ohio State University, Computer and Information Science Research Center Report OSU-CISRC-TR072-12, November, 1972.

6.32 Anderson, R.E., and Fagerlund, E., "Privacy and the Computer: An Annotated Bibliography," Computing Reviews 13, 11, (November, 1972), pp. 551-559.

6.33 Reed, S.K., and Gray, M.M., "Controlled Accessibility Bibliography," NBS Technical Note 780, National Bureau of Standards, June, 1973.

6.34 Scherf, J.A., "Computer and Data Security: A Comprehensive Annotated Bibliography," Massachusetts Institute of Technology, Project MAC Technical Report TR-122, January, 1974.

6.35 Hoffman, L.J., editor, Security and Privacy in Computer Systems, Melville Publishing Co., 1973, Los Angeles.