About this paper

The Protection of Information in Computer Systems

JEROME H. SALTZER, SENIOR MEMBER, IEEE, AND
MICHAEL D. SCHROEDER, MEMBER, IEEE

Invited Paper

Abstract - **This tutorial paper explores the mechanics of protecting computer-stored information from unauthorized use or modification. It concentrates on those architectural structures--whether hardware or software--that are necessary to support information protection. The paper develops in three main sections. Section I describes desired functions, design principles, and examples of elementary protection and authentication mechanisms. Any reader familiar with computers should find the first section to be reasonably accessible. Section II requires some familiarity with descriptor-based computer architecture. It examines in depth the principles of modern protection architectures and the relation between capability systems and access control list systems, and ends with a brief analysis of protected subsystems and protected objects. The reader who is dismayed by either the prerequisites or the level of detail in the second section may wish to skip to Section III, which reviews the state of the art and current research projects and provides suggestions for further reading.**

Glossary

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

Access
: The ability to make use of information stored in a computer system. Used frequently as a verb, to the horror of grammarians.

Access control list
: A list of principals that are authorized to have access to some object.

Authenticate
: To verify the identity of a person (or other agent external to the protection system) making a request.

Authorize
: To grant a principal access to certain information.

Capability
: In a computer system, an unforgeable ticket, which when presented can be taken as incontestable proof that the presenter is authorized to have access to the object named in the ticket.

Certify
: To check the accuracy, correctness, and completeness of a security or protection mechanism.

Complete isolation
: A protection system that separates principals into compartments between which no flow of

information or control is possible.

Confinement
: Allowing a borrowed program to have access to data, while ensuring that the program cannot release the information.

Descriptor
: A protected value which is (or leads to) the physical address of some protected object.

Discretionary
: (In contrast with *nondiscretionary*.) Controls on access to an object that may be changed by the creator of the object.

Domain
: The set of objects that currently may be directly accessed by a principal.

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.

Grant
: To authorize (*q. v.*).

Hierarchical control
: Referring to ability to change authorization, a scheme in which the record of each authorization is controlled by another authorization, resulting in a hierarchical tree of authorizations.

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

Password
: A secret character string used to authenticate the claimed identity of an individual.

Permission
: A particular form of allowed access, e.g., permission to READ as contrasted with permission to WRITE.

Prescript
: A rule that must be followed before access to an object is permitted, thereby introducing an opportunity for human judgment about the need for access, so that abuse of the access is discouraged.

Principal
: The entity in a computer system to which authorizations are granted; thus the unit of accountability in a computer system.

Privacy
: The ability of an individual (or organization) to decide whether, when, and to whom personal (or organizational) information is released.

Propagation
: When a principal, having been authorized access to some object, in turn authorizes access to another principal.

Protected object
: A data structure whose existence is known, but whose internal organization is not accessible, except by invoking the protected subsystem (*q.v.*) that manages it.

Protected subsystem
: A collection of procedures and data objects that is encapsulated in a domain of its own so 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 domain entry points.

Protection

1) Security (*q.v.*).

2) Used more narrowly to denote mechanisms and techniques that control the access of executing programs to stored information.

Protection group
: A principal that may be used by several different individuals.

Revoke
: To take away previously authorized access from some principal.

Security
: With respect to information processing systems, used to denote mechanisms and techniques that control who may use or modify the computer or the information stored in it.

Self control
: Referring to ability to change authorization, a scheme in which each authorization contains within it the specification of which principals may change it.

Ticket-oriented
: Used to describe a protection system in which each principal maintains a list of unforgeable bit patterns, called tickets, one for each object the principal is authorized to have access.

User
: Used imprecisely to refer to the individual who is accountable for some identifiable set of activities in a computer system.

I. BASIC PRINCIPLES OF INFORMATION PROTECTION

A. Considerations Surrounding the Study of Protection

1) General Observations: As computers become better understood and more economical, every day brings new applications. Many of these new applications involve both storing information and simultaneous use by several individuals. 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 ensure 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 be sure that a request for a passenger list from a law enforcement agency is reviewed by the correct level of management.

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 online warehouse inventory management system that generates reports about the current status of the inventory. These reports not only represent corporate information that must be protected from release outside the company, but also may indicate 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 in common controlled sharing of information among multiple users. All, therefore, require some plan to ensure that the computer system helps implement the correct authority structure. Of course, in some applications 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 adequately protects the stored information. Although there are situations in which the computer need provide no aids to ensure protection of information, often it is appropriate to have the computer enforce a desired authority structure.

The words "privacy," "security," and "protection" are frequently used in connection with information-storing systems. Not all authors use these terms in the same way. This paper uses definitions commonly encountered in computer science literature.

The term "privacy" denotes a socially defined ability of an individual (or organization) to determine whether, when, and to whom personal (or organizational) information is to be released.

This paper will not be explicitly concerned with privacy, but instead with the mechanisms used to help achieve it.[1]

The term "security" describes techniques that control who may use or modify the computer or the *information contained in it.*[2]

Security specialists (e.g., Anderson [6]) 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 information use and from those patterns can infer some information content. It also includes unauthorized use of a proprietary program.

2) Unauthorized information modification: an unauthorized person is able to make changes in stored information--a form of sabotage. Note that this kind of violation does not require that the intruder see the information he has changed.

3) Unauthorized denial of use: an intruder can prevent an authorized user from referring to or modifying information, even though the intruder may not be able to refer to or modify the information. Causing a system "crash," disrupting a scheduling algorithm, or firing a bullet into a computer are examples of denial of use. This is another form of sabotage.

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 who controls the information, possibly even contrary to the constraints supposedly enforced by the system. The biggest complication 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.

Examples of security techniques sometimes applied to computer systems are the following:

1. labeling files with lists of authorized users,
2. verifying the identity of a prospective user by demanding a password,
3. shielding the computer to prevent interception and subsequent interpretation of electromagnetic radiation,
4. enciphering information sent over telephone lines,
5. locking the room containing the computer,
6. controlling who is allowed to make changes to the computer system (both its hardware and software),
7. using redundant circuits or programmed cross-checks that maintain security in the face of hardware or software failures,
8. certifying that the hardware and software are actually implemented as intended.

It is apparent that a wide range of considerations 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 security techniques that control the access of executing programs to stored information.[3] An example of a protection technique is labeling of computer-stored files with lists of authorized users. Similarly, the term *authentication* is used for those security techniques that verify the identity of a person (or other external agent) making a request of a computer system. An example of an authentication technique is demanding a password. This paper concentrates on protection and authentication mechanisms, with only occasional reference to the other equally necessary security mechanisms. One should recognize that concentration on protection and authentication mechanisms provides a narrow view of information security, and that a narrow view is dangerous. The objective of a secure system is to prevent all unauthorized use of information, a negative kind of requirement. It is hard to prove that this negative requirement has been achieved, for one must demonstrate that every possible threat has been anticipated. Thus an expansive view of the problem is most appropriate to help ensure that no gaps appear in the strategy. In contrast, a narrow concentration on protection mechanisms, especially those logically impossible to defeat, may lead to false confidence in the system as a whole.[4]

2) Functional Levels of Information Protection: Many different designs have been proposed and mechanisms implemented for protecting information in computer systems. One reason for differences among protection schemes is their different functional properties--the kinds of access control that can be expressed naturally and enforced. It is convenient to divide protection schemes according to their functional properties. A rough categorization is the following.

a) Unprotected systems: Some systems have no provision for preventing a determined user from having access 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 1975, many of the most widely used, commercially available batch data processing systems fall into this category--for example, the Disk Operating System for the IBM System 370 [9]. Our definition of protection, which excludes features usable only for mistake prevention, is important here since it is common for unprotected systems to contain a variety of mistake-prevention features. These may provide just enough control that any breach of control is likely to be the result of a deliberate act rather than an accident. Nevertheless, it would be a mistake to claim that such systems provide any security.[5]

b) All-or-nothing systems: These are systems that provide isolation of users, sometimes moderated by total sharing of some pieces of information. If only isolation is provided, the user of such a system might just as well be using his own private computer, as far as protection and sharing of information are concerned. More commonly, such systems also have public libraries to which every user may have access. In some cases the public library mechanism 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 timesharing systems provide a protection scheme with this level of function. Examples include the Dartmouth Time-Sharing System (DTSS) [10] and IBM's VM/370 system [11]. There are innumerable others.

c) Controlled sharing: Significantly more complex machinery is required to control explicitly 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 straightforward, actual implementation is surprisingly intricate, and only a few complete examples exist. These include M.l.T.'s Compatible Time-Sharing System (CTSS) [12], Digital Equipment Corporation's DECsystem/10 [13], System Development Corporation's Advanced Development Prototype (ADEPT) System [14], and Bolt, Beranek, and Newman's TENEX [15][6]

d) User-programmed sharing controls: A user may want to restrict access to a file in a way not provided in the standard facilities for controlling sharing. For example, he may wish to permit 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 of the data in a file. Maybe he wishes to require that a file be modified only if two users agree. For such cases, and a myriad of others, 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 only the programs of the subsystem have direct access to the data (that is, the protected objects). Access to those programs is limited to calling specified entry points. Thus the programs of the subsystem completely control the operations performed on the data. By constructing a protected subsystem, a user can develop any programmable form of access control to the objects he creates. Only a few of the most advanced system designs have tried to permit user-specified protected subsystems. These include Honeywell's Multics [16], the University of California's CAL system [17], Bell Laboratories' UNIX system [18], the Berkeley Computer Corporation BCC-500 [19], and two systems currently under construction: the CAP system of Cambridge University [20], and the HYDRA system of Carnegie-Mellon University [21]. Exploring alternative mechanisms for implementing protected subsystems is a current research topic. A specialized use of protected subsystems is the implementation of protection controls based on data content. For example, in a file of salaries, one may wish to permit access to all salaries under $15 000. Another example is permitting access to certain statistical aggregations of data but not to any individual data item. This area of protection raises questions about the possibility of discerning information by statistical tests and by examining indexes, without ever having direct access to the data itself. Protection based on content is the subject of a variety of recent or current research projects [22]-[25] and will not be explored in this tutorial.

e) Putting strings on information: The foregoing three levels have been concerned with establishing conditions for the release of information to an executing program. The fourth level of capability is to maintain some control over the user of the information even *after* it has been released. Such control is desired, for example, in releasing income information to a tax advisor; constraints should prevent him from passing the information on to a firm which prepares mailing lists. The printed labels on classified military information declaring 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 that implement such strings on information are rare and the mechanisms are incomplete. For example, the ADEPT system [14] keeps track of the classification level of all input data used to create a file; all output data are automatically labeled with the highest classification encountered during execution.

There is a consideration that cuts across all levels of functional capability: the *dynamics of use*. This term refers to how one establishes and changes the specification of who may access what. At any of the levels it is relatively easy to envision (and design) systems that statically express a particular protection intent. But the need to change access authorization dynamically and the need for such changes to be requested by executing programs introduces much complexity into protection systems. For a given functional level, most existing protection systems differ primarily in the way they handle protection dynamics. To gain some insight into the complexity introduced by program-directed changes to access authorization, consider the question "Is there any way that O'Hara could access file X?" One should check to see not only if O'Hara has access to file X, but also whether or not O'Hara may change the specification of file X's accessibility. The next step is to see if O'Hara can change the specification of who may change the specification of file X's accessibility, etc. Another problem of dynamics arises when the owner revokes a user's access to a file while that file is being used. Letting the previously authorized user continue 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 authorization may severely disrupt the user. It should be apparent that provisions for the dynamics of use are at least as important as those for static specification of protection intent.

In many cases, it is not necessary to meet the protection needs of the person responsible for the information stored in the computer entirely through computer-aided enforcement. External mechanisms such as contracts, ignorance, or barbed wire fences may provide some of the required functional capability. This discussion, however, is focused on the internal mechanisms.

3) Design Principles: Whatever the level of functionality provided, the usefulness of a set of protection mechanisms depends upon the ability of a system to prevent security violations. In practice, producing a system at any level of functionality (except level one) that actually does prevent all such unauthorized acts has proved to be extremely difficult. Sophisticated users of most systems are aware of at least one way to crash the system, denying other users authorized access to stored information. Penetration exercises involving a large number of different general-purpose systems all have shown that users can construct programs that can obtain unauthorized access to information stored within. Even in systems designed and implemented with security as an important objective, design and implementation flaws provide paths that circumvent the intended access constraints. Design and construction techniques that systematically exclude flaws are the topic of much research activity, but no complete method applicable to the construction of large general-purpose systems exists yet. This difficulty is related to the negative quality of the requirement to prevent *all* unauthorized actions.

In the absence of such methodical techniques, experience has provided some useful principles that can guide the design and contribute to an implementation without security flaws. Here are eight examples of design principles that apply particularly to protection mechanisms.[7]

a) Economy of mechanism: Keep the design as simple and small as possible. This well-known principle applies to any aspect of a system, but it deserves emphasis for protection mechanisms for this reason: design and implementation errors that result in unwanted access paths will not be noticed during normal use (since normal use usually does not include attempts to exercise improper access paths). As a result, techniques such as line-by-line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential.

b) Fail-safe defaults: Base access decisions on permission rather than exclusion. This principle, suggested by E. Glaser in 1965,[8] 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, presents the wrong psychological base for secure system design. A conservative design must be based on arguments why objects should be accessible, rather than why they should not. In a large system some objects will be inadequately considered, so a default of lack of permission is safer. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use. This principle applies both to the outward appearance of the protection mechanism and to its underlying implementation.

c) Complete mediation: Every access to every object must be checked for authority. 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 implies that a foolproof method of identifying the source of every request must be devised. It also requires that proposals to gain performance by remembering the result of an authority check be examined skeptically. If a change in authority occurs, such remembered results must be systematically updated.

d) Open design: The design should not be secret [27]. 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 decoupling of protection mechanisms from 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.[9] Finally, it is simply not realistic to attempt to maintain secrecy for any system which receives wide distribution.

e) Separation of privilege: Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that 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. It is also at work in the defense system that fires a nuclear weapon only if two different people both give the correct command. In a computer system, separated keys apply to any situation in which two or more conditions must be met before access should be permitted. For example, systems providing user-extendible protected data types usually depend on separation of privilege for their implementation.

f) Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism 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.

g) Least common mechanism: Minimize the amount of mechanism common to more than one user and depended on by all users [28]. Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security. Further, any mechanism serving all users must be certified to the satisfaction of every user, a job presumably harder than satisfying only one or a few users. For example, given the choice of implementing a new function as a supervisor procedure shared by all users or as a library procedure that can be handled as though it were the user’s own, choose the latter course. Then, if one or a few users are not satisfied with the level of certification of the function, they can provide a substitute or not use it at all. Either way, they can avoid being harmed by a mistake in it.

h) Psychological acceptability: It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user’s mental image of his protection goals matches 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.

Analysts of traditional physical security systems have suggested two further design principles which, unfortunately, apply only imperfectly to computer systems.

a) Work factor: Compare the cost of circumventing the mechanism with the resources of a potential attacker. The cost of circumventing, commonly known as the "work factor," in some cases can 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 terminal, 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. The trouble with the work factor principle is that many computer protection mechanisms are *not* susceptible to direct work factor calculation, since defeating them by systematic attack may be logically impossible. Defeat can be accomplished only by indirect strategies, such as waiting for an accidental hardware failure or searching for an error in implementation. Reliable estimates of the length of such a wait or search are very difficult to make.

b) Compromise recording: It is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss. For example, if a tactical plan is known to have been compromised, it may be possible to construct a different one, rendering the compromised version worthless. An unbreakable padlock on a flimsy file cabinet is an example of such a mechanism. Although the information stored inside may be easy to obtain, the cabinet will inevitably be damaged in the process and the next legitimate user will detect the loss. For another example, many computer systems record the date and time of the most recent use of each file. If this record is tamperproof and reported to the owner, it may help discover unauthorized use. In computer systems, this approach is used rarely, since it is difficult to guarantee discovery once security is broken. Physical damage usually is not involved, and logical damage (and internally stored records of tampering) can be undone by a clever attacker.[10]

As is apparent, these principles do not represent absolute rules--they 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.

4) Summary of Considerations Surrounding Protection: Briefly, then, we may outline our discussion to this point. The application of computers to information handling problems produces a need for a variety of security mechanisms. We are focusing on one aspect, computer protection mechanisms--the mechanisms that control access to information by executing programs. At least four levels of functional goals for a protection system can be identified: all-or-nothing systems, controlled sharing, user-programmed sharing controls, and putting strings on information. But at all levels, the provisions for dynamic changes to authorization for access are a severe complication.

Since no one knows how to build a system without flaws, the alternative is to rely on eight design principles, which tend to reduce both the number and the seriousness of any flaws: Economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability.

Finally, some protection designs can be evaluated by comparing the resources of a potential attacker with the work factor required to defeat the system, and compromise recording may be a useful strategy.

B. Technical Underpinnings

1) The Development Plan: At this point we begin a development of the technical basis of information protection in modern computer systems. There are two ways to approach the subject: from the top down, emphasizing the abstract concepts involved, or from the bottom up, identifying insights by, studying example systems. We shall follow the bottom-up approach, introducing a series of models of systems as they are, (or could be) built in real life.

The reader should understand that on this point the authors' judgment differs from that of some of their colleagues. The top-down approach can be very satisfactory when a subject is coherent and self-contained, but for a topic still containing *ad hoc* strategies and competing world views, the bottom-up approach seems safer.

Our first model is of a multiuser system that completely isolates its users from one another. We shall then see how the logically perfect walls of that system can be lowered in a controlled way to allow limited sharing of information between users. Section II of this paper generalizes the mechanics of sharing using two different models: the capability system and the access control list system. It then extends these two models to handle the dynamic situation in which authorizations can change under control of the programs running inside the system. Further extensions to the models control the dynamics. The final model (only superficially explored) is of protected objects and protected subsystems, which allow arbitrary modes of sharing that are unanticipated by the system designer. These models are not intended so much to explain the particular systems as they are to explain the underlying concepts of information protection.

Our emphasis throughout the development is on direct access to information (for example, using LOAD and STORE instructions) rather than acquiring information indirectly (as when calling a data base management system to request the average value of a set of numbers supposedly not directly accessible). Control of such access is the function of the protected subsystems developed near the end of the paper. Herein lies perhaps the chief defect of the bottom-up approach, since conceptually there seems to be no reason to distinguish direct and indirect access, yet the detailed mechanics are typically quite different. The beginnings of a top-down approach based on a message model that avoids distinguishing between direct and indirect information access may be found in a paper by Lampson [30].

2) The Essentials of Information Protection: For purposes of discussing protection, the information stored in a computer system is not a single object. When one is considering direct access, the information is divided into mutually exclusive partitions, as specified by its various creators. 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 collections of information in the partitions are the fundamental objects to be protected.

Conceptually, then, it is necessary to build an impenetrable wall around each distinct object that warrants separate protection, construct a door in the wall through which access can be obtained, and post 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 authority check is usually implemented by having the guard demand a match between something he knows and something the prospective user possesses. Both protection and authentication mechanisms can be viewed in terms of this general model.

Before extending this model, we pause to consider two concrete examples, the multiplexing of a single 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. Later we will extend our model of guards and walls in the discussion of shared information.

3) An Isolated Virtual Machine: A typical computer consists of a processor, a linearly addressed memory system, and some collection of input/output devices associated with the processor. It is relatively easy to use a single computer to simulate several, each of which is completely unaware of the existence of the others, except that each runs more slowly than usual. Such a simulation is of interest, since during the intervals when one of the simulated (commonly called *virtual*) processors is waiting for an input or output operation to finish, another virtual processor 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 virtual processor to be unaware of the existence of the others, it is essential that some isolation mechanism be provided. One such mechanism is a special hardware register called a *descriptor register*, as in Fig. 1. In this figure, all memory references by the processor are checked by an extra piece of hardware that is interposed in the path to the memory. The descriptor register controls exactly which part of memory is accessible. The descriptor register contains two components: a *base* value and a *bound* value. The base is the lowest numbered address the program may use, and the bound is the number of locations beyond the base that may be used.[11] We will call the value in the descriptor register a *descriptor*, as it describes an object (in this case, one program) stored in memory. The program controlling the processor has full access to everything in the base-bound range, by virtue of possession of its one descriptor. As we go on, we shall embellish the concept of a descriptor: it is central to most implementations of protection and of sharing of information.[12]

So far, we have not 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. The instruction that 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. It is easier to control who may use the descriptor, and a common scheme is to introduce an additional bit in the processor state. This bit is called the *privileged state* bit.[13] 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 changed. One program (named the *supervisor*--program S in Fig. 1) runs with the privileged state bit ON, and controls the simulation of the virtual processors for the other programs. All that is needed to make the scheme complete is to ensure that the privileged state bit cannot be changed by the user programs except, perhaps, by an instruction that simultaneously transfers control to the supervisor program at a planned entry location. (In most implementations, the descriptor register is not used in the privileged state.)

One might expect the supervisor program to maintain a table of values of descriptors, one for each virtual processor. 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 processor--is accountable for the activity of the real processor. For protection to be complete, a virtual processor must not be able to change arbitrarily 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 processors. We have here an example of a common strategy and sometime cause of confusion: the protection mechanisms not only protect one user from another, *they may also protect their own implementation*. We shall encounter this strategy again.

So far, this virtual processor implementation contains three protection mechanisms that we can associate with our abstractions. For the first, the information being protected is the distinct programs of Fig. 1. The guard is represented by the extra piece of hardware that enforces the descriptor restriction. The impenetrable wall with a door is the hardware that forces all references to memory through the descriptor mechanism. The authority check on a request to access memory is very simple. The requesting virtual processor is identified by the base and bound values in the descriptor register, and the guard checks that the memory location to which access is requested lies within the indicated area of memory.

The second mechanism protects the contents of the descriptor register. The wall, door, and guard are implemented in hardware, as with the first mechanism. An executing program requesting to load the descriptor register is identified by the privileged state bit. If this bit is OFF, indicating that the requester is a user program, then the guard does not allow the register to be loaded. If this bit is ON, indicating that the requester is the supervisor program, then the guard does allow it.

The third mechanism protects the privileged state bit. It allows an executing program identified by the privileged state bit being OFF (a user program) to perform the single operation "turn privileged state bit ON and transfer to the supervisor program." An executing program identified by the privileged state bit being ON is allowed to turn the bit OFF. This third mechanism is an embryonic form of the sophisticated protection mechanisms required to implement protected subsystems. The supervisor program is an example of a protected subsystem, of which more will be said later.

The supervisor program is part of all three protection mechanisms, for it is responsible for maintaining the integrity of the identifications manifest in the descriptor register and the privileged state bit. If the supervisor does not do its job correctly, virtual processors could become labeled with the wrong base and bound values, or user programs could become labeled with a privileged state bit that is ON, The supervisor protects itself from the user programs with the same isolation hardware that separates users, an example of the "economy of mechanism" design principle.

With an appropriately sophisticated and careful supervisor program, we now have an example of a system that completely isolates its users from one another. Similarly isolated permanent storage can be added to such a system by attaching some longterm 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 to implement its descriptor scheme with the supervisor programs rather than hardware, but the principle is the same. Data streams to input or output devices can be controlled similarly. The combination of a virtual processor, a memory area, some data streams, and an isolated region of long-term storage is known as a virtual machine.[14]

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 and its long-term storage area with a single terminal. Then, for example, padlocks can control access to the terminal. If, on the other hand, a more flexible system is desired, the supervisor program must be prepared to associate any terminal with any virtual machine and, as a result, must be able to verify the identity of the user at a terminal. Schemes for performing this authentication are the subject of our next example.

4) Authentication Mechanisms: Our second example is of an authentication mechanism: a system that verifies a user's claimed identity. The mechanics of this authentication mechanism differ from those of the protection mechanisms for implementing virtual machines 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 and the intended virtual machine. Such systems follow our abstract model of a guard who demands a match between something he knows and something the requester possesses. The objects being protected by the authentication mechanism are the virtual machines. In this case, however, the requester is a computer system user rather than an executing program, and because of the lack of physical control over the user and the communication system, the security of the computer system must depend on either the secrecy or the unforgeability of the user's 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 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 that must be carefully guarded by the system. In some systems, all passwords are passed through a hard-to-invert transformation[15] before being stored, an idea suggested by R. Needham [37, p. 129]. 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 transform itself is well known), 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 "hardness of inversion" is difficult to measure. 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. In one recent study of some 300 self-chosen passwords on a typical time-sharing system, more than 50 percent were found to be short enough to guess by exhaustion, derived from the owner's name, or something closely associated with the owner, such as his telephone number or birth date. For this reason, some systems have programs that generate random sequences of letters for use as passwords. They may even require that all passwords be system-generated and changed frequently. 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. One solution to this problem is to provide a generator of "pronounceable" random passwords based on digraph or higher order frequency statistics [26] to make memorization easier.

A second significant defect is that the password must be exposed to be used. 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 relatively difficult-to-fabricate object. The terminal has an input device that examines the object and transmits its unique identifying code to the computer system, which treats the code as a password that 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-fabricate 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 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 conventional 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 penetrate a password system, for example, is to intercept all communications to and from the terminal and direct them to another computer--one that 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. The same attack can be used on the unforgeable object system as well.

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 [38], 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 *encryption* 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 that a recipient of such an enciphered signal may comprehend it, he must have a deciphering circuit 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 on 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 [39]. The user, at a terminal, begins bypassing the enciphering equipment. He types his name. This name passes, unenciphered, through the communication system to the computer. The computer looks up the name, just as with the password system. Associated with each name, 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 used for authentication--the transformation key--has not been transmitted through the communication system. If communication fails (because the user is unauthorized, the system has been replaced by a masquerader, or an error occurred), each party to the transaction has immediate warning of a problem.[16]

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 [40] 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.

5) Shared Information: The virtual machines of the earlier section were totally independent, as far as information accessibility was concerned. Each user might just as well have his own private computer system. 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 users is useful, or even essential. For example, there may be a library of commonly used, reliable programs. Some users may create new programs that other users would like to use. Users may wish to be able to update a common data base, such as a file of airline seat reservations or a collection of programs that implement a biomedical statistics system. In all these cases, virtual machines are inadequate, because of the total isolation of their users from one another. Before extending the virtual machine example any further, let us return to our abstract discussion of guards and walls.

Implementations of protection mechanisms that permit sharing fall into the two general categories described by Wilkes [37]

a) "List-oriented" implementations, in which the guard holds a list of identifiers of authorized users, and the user carries a unique unforgeable identifier that must appear on the guard's list for access 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.

b) "Ticket-oriented" implementations, in which the guard holds the description of a single identifier, and each user has a collection of unforgeable identifiers, or tickets,[17] corresponding to the objects to which he has been authorized access. A locked door that 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, defined as giving a user access to some object, is different in these two schemes. In a list-oriented system, a user is authorized to use an object by having his name placed 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 means that some kind of associative search must accompany 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 only need compare the presented ticket with his own expectation before allowing the physical memory access. Because associative matching tends to be either slower or more costly than simple comparison, 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 tickets around from one user to another. As a rule, most real systems contain both kinds of sharing implementations--a list-oriented system at the human interface and a ticket-oriented system in the underlying hardware implementation. This kind of arrangement is accomplished by providing, at the higher level, a list-oriented guard[18] whose only purpose is to hand out temporary tickets which the lower level (ticket-oriented) guards will honor. Some added complexity arises from the need to keep authorizations, as represented in the two systems, synchronized with each other. Computer protection systems differ mostly in the extent to which the architecture of the underlying ticket-oriented system is visible to the user.

Finally, let us consider the degenerate cases of list- and ticket-oriented systems. In a list-oriented system, if each guard’s list of authorized users can contain only one entry, we have a "complete isolation" kind of protection system, in which no sharing of information among users can take place. Similarly, in a ticket-oriented system, if there can be only one ticket for each object in the system, we again have a "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. These observations are important in examining real systems, which usually consist 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 understand the relationship of a user to these transactions. We are concerned with protection of information from programs that are executing. The user is the individual who assumes accountability for the actions of an executing program. Inside the computer system, a program is executed by a virtual processor, so one or more virtual processors can be identified with the activities directed by the user.[19]

In a list-oriented system it is the guard’s business to know whose virtual processor is attempting to make an access. The virtual processor has been marked with an unforgeable label identifying the user accountable for its actions, and the guard inspects this label when making access decisions. In a ticket-oriented system, however, the guard cares only that a virtual processor present the appropriate unforgeable ticket when attempting an access. The connection to an accountable user is more diffuse, since the guard does not know or care how the virtual processor acquired the tickets. In either case, we conclude that in addition to the information inside the impenetrable wall, there are two other things that must be protected: the guard’s authorization information, and the association between a user and the unforgeable label or set of tickets associated with his virtual processors.

Since an association with some user is essential for establishing accountability for the actions of a virtual processor, it is useful to introduce an abstraction for that accountability--the *principal*. A principal is, by definition, the entity accountable for the activities of a virtual processor.[20] In the situations discussed so far, the principal corresponds to the user outside the system. However, there are situations in which a one-to-one correspondence of individuals with principals is not adequate. For example, a user may be accountable for some very valuable information and authorized to use it. On the other hand, on some occasion he may wish 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 that 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 corresponding to the same user.

Similarly, one can envision a data base that is to be modified only if a committee agrees. Thus there might be an authorized principal that cannot be used by any single individual; all of the committee members must agree upon its use simultaneously.

Because the principal represents accountability, we shall see later (in the section on dynamic authorization of sharing) that authorizing access is done in terms of principals. That is, if one wishes a friend to have access to some file, the authorization is done by naming a principal only that friend can use.

For each principal we may identify all the objects in the system which the principal has been authorized to use. We will name that set of objects the *domain* of that principal.

Summarizing, then, a principal is the unforgeable identifier attached to a virtual processor 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, establishes the user's right to use that principal. The authentication mechanism itself may be either list- or ticket-oriented or of the complete isolation type. Then a computation is begun in which all the virtual processors of the computation are labeled with the identifier of that principal, which is considered accountable for all further actions of these virtual processors. The authentication mechanism has allowed the virtual processor to enter the domain of that principal. That description makes apparent the importance of the authentication mechanism. Clearly, one must carefully control the conditions under which a virtual processor enters a domain.

Finally, we should note that in a ticket-oriented system there is no mechanical need to associate an unforgeable identifier with a virtual processor, since the tickets themselves are presumed unforgeable. Nevertheless, a collection of tickets can be considered to be a domain, and therefore correspond to some principal, even though there may be no obvious identifier for that principal. Thus accountability in ticket-oriented systems can be difficult to pinpoint.

Now we shall return to our example system and extend it to include sharing. 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, although workable, has several defects. Most obvious, the multiple copies require multiple storage spaces. More subtly, the scheme does not respond well to changes. If a newer, better math routine is written, upgrading 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 access to a single *master copy* of the program. 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 share a single copy of the math routine by adding to the real processor a second descriptor register, as in Fig. 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 bit select the descriptor register. Thus, in Fig. 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 [42], is to add explicitly to the format of instruction words a field that 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 that can cause considerable confusion. The reason why descriptors are of interest for sharing becomes apparent by comparing parts a and b of Fig. 2. When program A is in control, it can have access only to itself and the math routine; similarly, when program

B is in control, it can have access only to itself and the math routine. Since neither program has the power to change the descriptor register, 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 Fig. 3, which is Fig. 2 redrawn with two virtual processors, one executing program A and the other 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 Fig. 3. Every virtual processor of the system may be viewed as having its own real processor, capable of access to the memory in parallel with that of every other virtual processor. There may be an underlying processor multiplexing facility that distributes a few real processors among the many virtual processors, but such a multiplexing facility is essentially unrelated to protection. Recall that a virtual 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 virtual processors. The processor multiplexing facility must be prepared to switch the entire state of the real processor from one virtual processor to another, including the values of the protection descriptor registers.

Although the basic mechanism to permit information sharing is now in place, a remarkable variety of implications that follow from its introduction require further mechanisms. These implications include the following.

1) If virtual processor P_1 can overwrite the shared math routine, then it could disrupt the work of virtual processor P_2.

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 more than one program or data base is 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 area of memory is a procedure, then to avoid the possibility that virtual processor P_1 will maliciously overwrite it, we can restrict the methods of access. Virtual processor P_1 needs to retrieve instructions from the area of the shared procedure, and may need to read out the values of constants embedded in the program, but it has no need 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 [32] . For example, we may add two bits, one controlling permission to read and the other permission to write in the storage area defined by each descriptor, as in Fig. 4. In virtual processor P_1 of Fig. 3, descriptor 1 would have both permissions granted, while descriptor 2 would permit only reading of data and execution of instructions.[21] An alternative 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 processors that had a descriptor. Although identical access for all users of the shared math routine of Figs. 1-2-3 might be acceptable, a data base could not be set up with several users having permission to read but a few also having permission to write.

The second implication of a shared procedure, mentioned before, is that the shared procedure must be careful about where it stores temporary results, since it may be used simultaneously by several virtual processors. 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.[22] The math routine will find that it can read but not write into itself, but that it can both read and write into the area of program A. Thus program A might allocate an area of its own address range for the math routine to use as temporary storage.[23]

As for the third implication, the need for expansion, we could generalize our example to permit several distinct shared items merely by increasing the number of descriptor registers and informing the supervisor which shared objects should be addressable by each virtual processor. However, there are two substantially different forms of this generalization--*capability systems* and *access control list systems*. In terms of the earlier discussion, capability systems are ticket-oriented, while access control list systems are list-oriented. Most real systems use a combination of these two forms, the capability system for speed and an access control list system for the human interface. Before we can pursue these generalizations, and the fourth implication, authorization, more groundwork must be laid.

In Section II, the development of protection continues with a series of successively more sophisticated models. The initial model, of a capability system, explores the use of encapsulated but copyable descriptors as tickets to provide a flexible authorization scheme. In this context we establish the general rule that communication external to the computer must precede dynamic authorization of sharing. The limitations of copyable descriptors--primarily lack of accountability for their use--lead to analysis of revocation and the observation that revocation requires indirection. That observation in turn leads to the model of access control lists embedded in indirect objects so as to provide detailed control of authorization.

The use of access control lists leads to a discussion of controlling changes to authorizations, there being at least two models of control methods which differ in their susceptibility to abuse. Additional control of authorization changes is needed when releasing sensitive data to a borrowed program, and this additional control implies a nonintuitive constraint on where data may be written by the borrowed program. Finally, Section II explores the concept of implementing arbitrary abstractions, such as extended types of objects, as programs in separate domains.

II. DESCRIPTOR-BASED PROTECTION SYSTEMS

A. Separation of Addressing and Protection[24]

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

Conceptually, we may achieve 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 Fig. 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 that indicates which part of the segment is to be read or written. All users of the memory system would use the same 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 that operates as a clock (counting, say, microseconds) and is large enough never to overflow in the lifetime of the memory system. The value of the clock register at the time a segment is created can be used as that segment's unique identifier.[25] As long as the memory system remembers anything, the time base of the clock register must not be changed.

The processor of Fig. 5 contains, as part of its state, protection descriptors similar to those of Figs. 1 and 2, with the addition of permissions, as in Fig. 4. All references by the processor are constrained to be to segments described by these protection descriptors. The protection descriptor itself no longer contains a base and bound; instead the descriptor contains the unique segment identifier that 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 Fig. 2. Note in Fig. 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 it has no *protection* descriptors for those two segments. It is useful to distinguish between the system address space, consisting of all the segments in the memory system, and the processor address space, consisting of those segments for which protection descriptors exist. Since the addressing descriptors are part of the memory system, which is shared by all processors, the system address space is universal. Any single processor address space, on the other hand, is defined by the particular protection descriptors associated with the processor and therefore is local. If the supervisor switches control of a real processor from one virtual processor to another, it would first reload the protection descriptors; the processor address space thus is different for different users, while the system address space remains the same for all users.

With the addressing function separated architecturally from the protection function, we may now examine the two generalized forms of protection systems: the capability system and the access control list system.

B. The Capability System

1) The Concept of Capabilities: The simplest generalization is the capability system suggested by Dennis and Van Horn [41], and first partially implemented on an M.l.T. PDP-1 computer [48].[26] 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. Another way to maintain the integrity of these registers would be to allow any program to load the protection descriptor registers, but only from locations in memory that previously have been 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, the word in that location is an ordinary data or instruction word. If the bit is ON, the word is taken to contain a value suitable for loading into a protection descriptor register. The instruction that loads the protection descriptor register will operate only if its operand address leads it to a location in memory that has the tag bit ON. To complete the scheme, we should provide an instruction that stores the contents of a protection descriptor register in memory and turns the corresponding tag bit ON, and we must arrange that all other store instructions set the tag bit OFF in any memory location they write into. This gives us 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, effectively, a wall that prevents values 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.[27]

This particular tagged architecture is known as a *capability system,* one that lets the user place protection descriptor values in memory addresses that are convenient to him. A memory word that contains a protection descriptor value (in our simple tagged system, one that has its tag bit ON) is known as a *capability*.

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 Fig. 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." In our example, this latter segment contains two locations with tags indicating that they are capabilities, Cl and C2. Program A may direct the processor to load the capability at location C2 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 direct the loading of the capability at location Cl 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 processor 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."

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 use. Each capability contains separate read and write 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 actually might 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 by convention might set some protection descriptor to contain a capability for the new segment.[28] 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 filing system to go with the basic protection structure.[29]

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 virtual processor and starting it executing in a supervisor program that initially has a capability for a user identification table, as in Fig. 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 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, destroying the capability for the user identification table, and start running some program in Doe’s directory, say program A. Program A can extend its addressability to any segment for which a capability exists in Doe’s catalog. Formally, after verifying the claimed identity of the user, the authentication system has allowed the virtual processor to enter Doe’s domain, starting in procedure A.

By providing for authentication we have actually tied together two protection systems: 1) an authentication system that controls access of users to named catalog capabilities, and 2) the general capability system that controls access of the holder of a catalog capability to other objects stored in the system.

The authentication system associates the newly created virtual processor with the principal accountable for its future activities. Once the virtual processor is started, however, the character-string identifier "Doe" is no longer used; the associated catalog capability is sufficient. The replacement of the character-string form 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. The catalog capability becomes, in effect, the principal identifier. On the other hand, some loss of accountability has occurred. It is no longer quite so easy, by examining the registers of a running virtual processor, to establish who is accountable for its activity. This lack of accountability will have to be repaired in order to allow the virtual processor to negotiate the acquisition of new capabilities.

With this example of a capability system, a catalog is not a special object. It is merely any segment in which any program chooses to store capabilities that are, by virtue of their tags, protected unforgeable objects. If in Fig. 7, program A, running under Doe’s control, creates a new object, it may choose to place the new capability in 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 that 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. It could be used in an application for which a simple, rarely changed, authorization pattern is useful. For example, a company data base management system might have a relatively static authorization pattern, which changes only when major revisions are made to the style of maintaining the data base. We have not yet provided, however, for the possibility that Doe, upon creating a new segment, might wish to authorize access to it for Smith. Such a need would probably arise if the computer system is used for the creation and editing of interoffice memoranda and letters or for constructing programs. We shall call this operation *dynamic authorization*. The dynamic authorization of sharing is a topic that must be examined quite carefully, since it exposes several subtle issues that are fundamental to sharing and protection.

2) The Dynamic Authorization of Sharing: One might propose to handle dynamic authorization very simply by arranging that Doe have a capability to write into Smith's catalog. Then Doe could 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 enable Doe 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. To develop this approach, we will consider a clumsy strategy with square-law growth, and then refine it.

If the possibility of sharing had been anticipated, both Doe and Smith might initially have had a capability allowing reading and writing a communication segment used only to pass messages and capabilities between Doe and Smith. Doe's program 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 have to locate the capability for the common communication segment. How do they know what to look for? Consider the case of the sender, Doe's program, first. Presumably it looks in some trusted catalog for the name "Smith" and finds the capability for the communication segment next to Smith's name. But how does Doe's program know to look for the name "Smith"? The character-string name may be embedded in the program by Doe or he may type it into his program as it runs, but either way one thing is crucial--that there be 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 does Doe find out the character-string name "Smith" so that he could type it in or embed it in his program? Presumably, he learns Smith's name via some path *outside the computer*. Perhaps Smith shouts it down the hall to him.[30] 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-cataloged name of the intended recipient. That name is, by definition, the identifier of the recipient's principal within the computer system. Thus the sender can be sure that only programs run under the accountability of that principal will have access to his new object.

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, Doe's principal identifier should be the name used in Smith's catalog of communication segments, so Smith can be certain that only some program run under Doe's accountability could possibly have sent the capability. In summary, here is a complete protocol for dynamically authorizing sharing of a new object.

Sender's part:

1. Sender learns receiver’s principal identifier via a communication path outside the system.
2. Sender transmits receiver’s principal identifier to some program running inside the system under the accountability of the sender.
3. Sender’s program uses receiver’s principal identifier to ensure that only virtual processors operating under the accountability of the receiver will be able to obtain the capability being transmitted.

Receiver’s part:

1. Receiver learns sender’s principal identifier, via a communication path outside the system.
2. Receiver transmits sender’s principal identifier to some program running inside the system under the accountability of the receiver.
3. Receiver’s program uses the sender’s principal identifier to ensure that only a virtual processor operating under the accountability of the sender could have sent the capability being received.

This protocol provides protection for the authorization changing mechanism (copying of a capability) by requiring an authority check (comparison of a principal identifier found inside the system with authorization information transmitted from outside). Although the analysis may seem somewhat strained, it is important because it always applies, even though parts of it may be implicit or hidden. We have described the protocol in terms of a capability system, but the same protocol also applies in access control list systems.

Our analysis of the dynamics of authorizing sharing has been in terms of private communication segments between every pair of users, a strategy which would lead, with N users, to some N^2 communication segments. To avoid this square-law growth, one might prefer to use some scheme that dynamically constructs the communication paths also, such as having special hardware or a protected subsystem that implements a single "mailbox segment" for each user to receive messages and capabilities sent by all other users. Of course, the mechanism that implements the mailbox segments must be a protected, reliable mechanism, since it must infallibly determine the principal identifier of the sender of a message and label the message with that identifier, so the receiver can reliably carry out his step 3) of the protocol. Similarly, as the sender’s agency, it must be able to associate the recipient’s principal identifier with the recipient’s mailbox, so that the sender’s intent in his step 3) of the protocol is carried out correctly.

3) Revocation and Control of Propagation: The capability system has as its chief virtues its inherent efficiency, simplicity, and flexibility. 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 that are part of such modern languages 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 gracefully to implement languages with dynamic-type extension [21] . Flexibility comes from the defining property of a capability system: the user may decide which of his addresses are to contain capabilities. The user can develop a data structure with an arbitrary pattern of access authorizations to his liking.

On the other hand, there are several potential problems with the capability system as we have sketched it so far. If Doe has a change of heart--he suddenly realizes that there is confidential information in the segment he permitted Smith to read--there is no way that he can disable the copy of the capability that 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 users, still trusted, who also have copies of the capability. Thus *revocation* of access is a problem.

A second, related property of a capability system is that Smith may now make copies of the capability and distribute them to other users, without the permission or even the knowledge of Doe. While in some cases, the ability of a recipient to pass access authorization along is exactly what the original grantor intended, in others it is not. 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 segment 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, a task that may involve a fair amount of computation and that also completely bypasses the protection mechanisms. Thus *review* of access is a problem.[31]

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 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 lost flexibility.

Alternatively, some segments (perhaps one per user) may be designated as *capability-holding* segments, and only those segments may be targets of the instructions that 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 CAP system [20] and the Plessey 250 [53] are organized in approximately this way, and the Burroughs B5000 family restricts descriptor storage to the virtual processor stack and a single table of outbound references [47].) In systems that make a programmer-visible distinction between short-term processor-addressable memory (addressed by LOAD and STORE instructions) and long-term storage (addressed by GET and PUT subroutines), it is possible to restrict capabilities so that they may be stored only in processor-addressable memory. This restriction not only reduces the effort required for auditing, but also limits the lifetime of a capability to that of a virtual processor. When the system shuts down, the only memory of the system is in long-term storage and all capabilities vanish. Of course, the next time the system starts up, newly created virtual processors need some way (such as appeal to an access control list system, described in the next subsection) to acquire the capabilities they need.

A third approach is to associate a *depth counter* with each protection descriptor register. The depth counter initially would have the value, say, of one, placed there by the supervisor. Whenever a program loads a descriptor register from a place in memory, that descriptor register receives a depth count that is one greater than the depth count of the descriptor register that contained the capability that permitted the loading. Any attempt to increase a depth count beyond, say, three, would constitute an error, and the processor would fault. In this way, the depth counters limit the length of the chain by which a capability may propagate. 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 [54] has proposed that the basic capability mechanism be extended to include the possibility of forcing a capability to specify its target indirectly through a second location before reaching the actual object of interest. This second location would be an independently addressable recognizable object, and anyone with an appropriate capability for it could destroy the indirect object, revoking access to anyone else who had been given a capability for that indirect object. By constructing a separate indirect object for each different principal he shared an object with, the owner of the object could maintain the ability to revoke access independently for each principal. The indirect objects would be implemented within the memory-mapping hardware (e.g., the addressing descriptors of Fig. 5) both to allow high-speed bypassing if frequent multiple indirections occur and also to allow the user of a capability to be ignorant of the existence of the indirection.[32] Redell's indirect objects are closely related to the *access controllers* of the access control list system, described in the next subsection. 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 an authorization--a kind of binding--takes place any time a capability is copied. Unless an indirect object is created for the copy, there is no provision for reversing this binding. The ability to make a further copy (and potentially a new authorization) is coupled to possession of a capability and is not independently controllable. Restrictions on the ability to copy, while helping to limit the number or kind of authorizations, also hamper the simplicity, flexibility, and uniformity of capabilities as addresses. In particular, capabilities are especially useful as a way of communicating exactly the necessary arguments from one procedure to another. In this way, they encourage wide use of procedures, a cornerstone of good programming practice. Restrictions on copyability, then, inhibit their usefulness in the context of procedure calls, and that runs counter to the goal of providing base-level facilities that encourage good programming practice. This dilemma seems to present an opportunity for research. At the present level of understanding, the most effective way of preserving some of the useful properties of capabilities is to limit their free copyability to the bottom most 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.[33]

C. The Access Control List System

1) Access Controllers: The usual strategy for providing reversibility of bindings is to control when 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 authorization check at the latest possible point. 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 microprogramming), although, historically, access control list systems have been implemented partly with interpretive software. Our initial model will impose the extra authorization check on *every* memory reference, an approach that is unlikely in practice but simpler to describe. Later we will show how to couple an access control list system to a capability system, a more typical realization that reduces the number of extra checks.

The system of Fig. 5 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 that a segment 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, and 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 Fig. 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. The access controller is to be used as a kind of indirect address, as in Fig. 9. In order to access a segment, the processor must supply the unique identifier of that segment's access controller. Since the access controller is protected, however, there is no longer any need for these unique identifiers to be protected. The former protection descriptor registers can be replaced with unprotected *pointer registers*, which can be loaded from any addressable location with arbitrary bit patterns. (In terms of IBM System 370 and Honeywell Multics, the pointer registers contain *segment numbers* from a universal address space. The segment numbers lead to the segment addressing descriptors stored in the access controller.) Of course, only bit patterns corresponding to the unique identifier of some segment's access controller will work. A data reference by the processor proceeds in the following steps, keyed to Fig. 9.

1. The program encounters an instruction that would write in the segment described by pointer register 3 at offset k.
2. The processor uses the unique identifier found in pointer register 3 to address access controller AC_1. The processor at the same time presents to the memory system the user's principal identifier, a request to write, and the offset k.
3. The memory system searches the access control list in AC_1 to see if this user's principal identifier is recorded there.
4. If the principal identifier is found, the memory system examines the permission bits associated with that entry of the access control list 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 work. The set of processor registers must be augmented with a new protected register that can contain the identifier of the principal currently accountable for the activity of the virtual processor, as shown in Fig. 9. (Without that change, one could not implement the second and third steps.)

For example, we may have an organization like that of Fig. 10, which implements essentially the same pattern of sharing as did the capability system of Fig. 6. The crucial difference between these two figures is that, in Fig. 10, all references to data are made indirectly via access controllers. Overall, the organization differs in several ways from 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, eliminating propagation of direct access. The pointer to X's access controller itself may be freely copied and passed to anyone, but every use of the pointer must be via the access controller, which prevents access by unauthorized principals.[34]
2. The access control list directly implements the sender's third step of the dynamic sharing protocol--verifying that the requester is authorized to use the object. In the capability system, verification was done once to decide if the first capability copy should be made; after that, further copying was unrestricted. The access control list, on the other hand, is consulted on every access.

3. Revocation of access has become manageable. A change to an access control list removing a name immediately preludes all future attempts by that user to use that segment.
4. The question of "who may access this segment?" apparently is answered directly by examining the access control list in the access controller for the segment. The qualifier "apparently" applies because we have not yet postulated any mechanism for controlling who may modify access control lists.
5. All unnecessary 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 Fig. 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 an 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.

2) Protection Groups: Cases often arise where it would be inconvenient to list by name every individual who is to have access to a particular segment, either because the list would be awkwardly long or because the list would change frequently. To handle this situation, most access control list systems implement factoring into *protection groups*, which are principals that may be used by more than one user. If the name of a protection group appears in an access control list, all users who are members of that protection group are to be permitted access to that segment.

Methods of implementation of protection groups vary widely. A simple way to add them to the model of Figs. 9 and 10 is to extend the "principal holding" register of the processor so that it can hold two (or more) principal identifiers at once, one for a personal principal identifier and one for each protection group of which the user is a member. Fig. 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 and any entries of the access control list.[35] Finally, who is allowed to use those principals that represent protection group identifiers must also be controlled systematically.

We might imagine that for each protection group there is 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 list is an example of an access control list that is protecting an object--a principal identifier other than a segment.) When a user logs in, he can specify the set 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 looking up the now-authenticated personal identifier on each named protection group list. If everything checks, a virtual processor can safely be created and started with the specified list of principal identifiers.[36]

3) Implementation Considerations: The model of a complete protection system as developed in Fig. 10 is one of many possible architectures, most of which have essentially identical functional properties; our choices among alternatives have been guided more by pedagogical considerations than by practical implementation issues. There are at least three key areas in which a direct implementation of Fig. 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 fast memory access 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. (This tradeoff between performance and complexity contrasts with the capability system, in which a single comparison is always sufficient.)
3. Allocation of space for access control lists, which can change in length, can be a formidable implementation problem. (Compared to a capability system, the mechanics of changing authorization in an access control list system are inherently more cumbersome.)

The first of these problems is attacked by recognizing that the purpose of the access control list is to establish authorization rather than to mediate every detailed access. Mediation of access would be handled more efficiently by a capability system. Suppose we provide for each pointer register a "shadow" capability register that is invisible to the virtual processor, as in Fig. 11. Whenever a pointer register containing the unique identifier of an access controller is first used, the shadow register is loaded with a capability consisting of a copy of the addressing descriptor for the segment protected by the access controller, together with a copy of the appropriate set of permission bits for this principal.[37] 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 the capabilities already loaded in shadow registers of running processors. (One could restore complete revocability by clearing all shadow registers of all processors and restarting any current access control list searches. The next attempted use of a cleared shadow register would automatically trigger its reloading and a new access contra list check.) The result is a highly constrained but very fast capabitity system beneath the access control list system. The detailed checking of access control falls on the capability mechanism, which on individual memory references exactly enforces the constraints specified by the access control list system.

The second and third problems, allocation and search of access control lists, appear to require more 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.[38]

A completely different way to provide an access control list system is to implement it in interpretive software in the path to the secondary storage or file system. Primary memory protection can be accomplished with either base-and-bound registers, or more generally with a capability system in which the capabilities cannot be copied into the file system. This approach takes the access control list checking mechanisms out of the heavily used primary memory access path, and reduces the pressure to compromise its functional properties. Such a mixed strategy, while more complex, typically proves to be the most practical compromise. For example, the Multics system [55] uses software-interpreted access control lists together with hardware-interpreted tables of descriptors. Similarly, the "guard file" of the Burroughs B6700 Master Control Program is an example of an access controller implemented interpretively [57].

4) Authority to Change Access Control Lists: The access control list organization brings one issue into focus: control of who may modify the access control information. 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 virtual processors, 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 of exerting that control is needed. The goal of any such mechanism is to provide within the computer an authority structure that models the authority structure of whatever organization uses the computer. Two different authority-controlling policies, with subtly different modeling abilities, have been implemented or proposed. We name these two *self control* and *hierarchical control*.

The simplest scheme is *self control*. With this scheme, we extend our earlier concept of access permission bits to include not just permission to read and write, but also permission to modify the access control list that contains the permission bits. Thus, in Fig. 12, we have a slightly more elaborate access controller, which by itself controls who may make modifications to it. Suppose that the creation of a new segment is accompanied by the creation of an access controller that contains one initial entry in its access control list-an entry giving all permissions to the principal identifier associated with the creating virtual processor. The creator receives a pointer for the access controller he has just created, and then can adjust its access control list to contain any desired list of principal identifiers and permissions.[39]

Probably the chief objection is 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, in a commercial time-sharing system, if a key member of a company's financial department is taken ill, there may be no way for his manager to authorize temporary access to a stored budget file for a co-worker unless the absent user had the foresight to set his access control lists just right. (Worse yet would be the possibility of accidentally producing an object for which its access controller permits access to no one--another version of the garbage collection problem.) To answer these objections, the *hierarchical control* scheme is sometimes used.

To obtain a hierarchical control scheme, whenever a new object is created the creator must specify some previously existing access controller 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 some kind of pointer to the access controller that regulates it (for example, a unique identifier). In addition, the interpretation of the permission bit named "ACLmod" is changed to apply to those access controllers that hierarchically are immediately below the access controller containing the permission bit. Then, as in Fig. 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 access control lists may add his own principal identifier, with permission to modify access, to lower level access controllers, giving himself ability to change access control lists still further down the hierarchy. Permission to modify access at any one node of the hierarchy permits the holder to grant himself access to anything in the entire subtree based on that node.[40]

The hierarchical control scheme might be used in a timesharing 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 permission to modify access in each controller to the department administrator. The department administrator 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 intevene and modify any access control list in his department. Similarly, the system administrator can intervene in case a department administrator makes a mistake or is unavailable.[41]

The hierarchical system in our example is subject to the objection that the system administrator and department administrators are *too* 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 hierarchical control method of modeling the organization 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. 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 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 the search is illegal unless he first obtained a warrant. In each case, the authority to perform the operation exists, but the use of the authority is coupled with checks and balances 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 strategy that has been suggested in various forms [58], [59] is to add a field to an access controller, 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 call to a supervisor entry, 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 that is about to be modified is examined, and some action, depending on the value found, is automatically triggered. The following list suggests some possible actions that might be triggered by the prescript value, and some external policies that can be modeled with the prescript scheme.

1. No action.
2. Identifier of principal making change is logged (the "audit trail").
3. Change is delayed one day ("cooling-off" period).
4. Change is delayed until some *other* principal attempts the same change ("buddy" system).
5. Change is delayed until signal is received from some specific (system-designated) principal ("court order").

The goal of all of the policies (and the prescript mechanism in general) is to ensure that some independent judgment moderates otherwise unfettered use of authority.

The notion of a prescript, while apparently essential to a protection system 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 model easily only a small range of policies. One could, for example, arrange that a prescript be invoked on every access to some segment, rather than just on changes in the authority structure. One could implement more complex policies by use of protected subsystems, a general escape mechanism described briefly in a later section.

5) Discretionary and Nondiscretionary Controls: Our discussion of authorization and authority structures has so far rested on an unstated assumption: the principal that creates a file or other object in a computer system has unquestioned authority to authorize access to it by other principals. 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*[42] implying that the individual user may, at his own discretion, determine who is authorized to access the objects he creates. In a variety of situations, 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 to ensure that only those employees with a need to know have access to information about the new product. The manager thus desires to apply the principle of least privilege. Similarly, the marketing manager may wish to compartmentalize all use of the company computer for calculating product prices, since pricing policy may be sensitive. Either manager may consider it not 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. The manager has a need to limit the use of discretionary controls by his employees. Any limits he imposes on authorization are controls that are out of the hands of his employees, and are viewed by them as *nondiscretionary*. Similar constraints are imposed in military security applications, in which not only isolated compartments are required, but also nested *sensitivity levels* (e.g., top secret, secret, and confidential) that must be modeled in the authorization mechanics of the computer system. Nondiscretionary controls may need to be imposed 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 nondiscretionary and discretionary controls apply.

The key reason for interest in nondiscretionary controls is not so much the threat of malicious insubordination as the need to safely use complex and sophisticated programs created by suppliers who are not under the manager's control. A contract software house may provide an APL interpreter or a fast file sorting program. If the supplied program is to be useful, it must be given access to the data it is to manipulate or interpret. But unless the borrowed program has been completely audited, there is no way to be sure that it does not misuse the data (for example, by making an illicit copy) or expose the data either accidentally or intentionally. One way to prevent this kind of security violation would be to forbid the use of borrowed programs, but for most organizations the requirement that all programs be locally written (or even thoroughly audited) would be an unbearable economic burden. The alternative is *confinement* of the borrowed program, a term introduced by Lampson [61]. That is, the borrowed program should run in a domain containing the necessary data, but should be constrained so that it cannot authorize sharing of anything found or created in that domain with other domains.

Complete elimination of discretionary controls is easy to accomplish. For example, if self-controlling access controllers are being used, one could arrange that the initial value for the access control list of all newly created objects not give "ACL-mod" permission to the creating principal (under which the borrowed program is running). Then the borrowed program could not release information by copying it into an object that it creates and then adjusting the access control list on that object. If, in addition, all previously existing objects in the domain of the borrowed program do not permit that principal to modify the access control list, the borrowed program would have no discretionary control at all and the borrower would have complete control. A similar modification to the hierarchical control system can also be designed.

It is harder to arrange for the coexistence of discretionary and nondiscretionary controls. Nondiscretionary controls may be implemented, for example, 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 fully general access control list for nondiscretionary controls, may be more elaborate than necessary. The few designs that have appeared so far have taken advantage of a perceived property of some applications of nondiscretionary controls: the desired patterns usually are 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.

An interesting requirement for a nondiscretionary control system that implements isolated compartments arises whenever a principal is 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 borrowed programs are to be confined. Confinement requires that the virtual processor be constrained to write only into objects that have a compartment set identical to that of the virtual processor itself. If such a restriction were not enforced, a malicious borrowed program could, upon reading data labeled for both the "pricing policy" and the "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 [14]. He suggested that the problem be solved by automatically labeling any written object with the compartment labels needed to permit writing, a strategy he named the "high water mark." As an alternative, the strategy suggested by Bell and LaPadula [62] declared that attempts to write into objects with too few compartment labels are errors that cause the program to stop.[43] Both cases recognize that writing into objects that do not have the necessary compartment labels represents potential "declassification" of sensitive information. Declassification should occur only after human judgment has been interposed to establish that the particular information to be written is not sensitive. Developing a systematic way to interpose such human judgments is a research topic.

Complete confinement of a program in a shared system is very difficult, or perhaps impossible, to accomplish, since the program may be able to signal to other users by strategies more subtle than writing into shared segments. For example, the program may intentionally vary its paging rate in a way users outside the compartment can observe, or it may simply stop, causing its user to go back to the original author for help, thereby revealing the fact that it stopped. D. Edwards characterized this problem with the phrase "banging on the walls." Lampson [61], Rotenberg [59], and Fenton [64] have explored this problem in some depth.

D. Protecting Objects Other Than Segments

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

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 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. 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 program issuing the instruction had loaded a capability permitting reading and writing the segment. One could not set up a segment so that some users could only enqueue messages, and not be able to dequeue--or even directly read--messages left by others. Such a distinction between queues and other segments can be made by introducing the concept of *type* in the protection system.

Consider, for example, the capability system in Fig. 6. Suppose we add to a capability an extra field, which we will name the type 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 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 extend the "create" operation to permit specification of the type of object being created.

Clearly, one could extend the notion of type beyond segments and queues; any data structure could be similarly distinguished and protected from misuse. Further, input and output streams attached to interactive terminals, printers, and the like could be considered distinct types with their own repertoire of separately permitted operations. 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, an indication of the type and permission bits required for the operation being performed. Table 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 that are basically interpreted data structures, such as accounts or catalogs.

TABLE I

Typical System-Provided Protected Objects

Object	Typical Separately Permittable Operations
Data segment	READ data from the segment WRITE data into the segment Use any capability found in the segment Write a capability into the segment
Access controller	Read access control list Modify names appearing on an access control list Modify permissions in access control list entries Destroy objects protected by this access controller
FIFO message queue	Enqueue a message Dequeue a message Examine queue contents without dequeueing
Input/Output	READ data WRITE data Issue device-control commands
Remove recording medium (e.g. magnetic tape reel)	READ data WRITE over data WRITE data in new area

Finally, one may wish to extend dynamically the range of objects protected. Such a goal might be reached 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-programmed 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.

E. Protected Objects and Domains

Both the capability system and the access control list system allow controlled sharing of the objects implemented by the system. Several common patterns of use can be independently controlled, such as reading, writing, or running as a program. While it is 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 his course grade records 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 any student to examine the histogram of the class grades for each assignment. Implementing such controls within systems 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 an 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.

The mechanisms described so far cannot handle this situation because the manipulations we wish to perform on a grade or a set of grades are not fundamental operations of the base-level system. In essence, we wish to dynamically define a new type, the grade record, and provide a set of programs that interpretively implement the operations appropriate for this new type.[44]

The second limitation concerns users who borrow programs constructed by other users. Execution of a borrowed program in the borrower's domain can present a real danger to the borrower, for the borrowed program can exercise all the capabilities in the domain 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 domain.

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 is "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 *protected objects*. 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 care taker programs. Algorithms in these caretaker programs may perform any appropriate operation, possibly depending on 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 Fig. 14 implements the grade keeping system discussed above. Clearly, any access constraints that 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 a protected subsystem can invoke programs in another protected subsystem without compromising the security of the first subsystem, then we can plug together multiple protected subsystems to perform a computation. We also find a way around the borrowed program problem. The normal domain of a user is one example of a protected subsystem. The user arranges for programs borrowed from other users to execute outside of this "home" protected subsystem. In this way, the borrowed programs can be invoked without giving them access to all the programs and data of the borrower. If the borrowed program is malicious or malfunctions, the damage it can do is 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.[45]

The notion of protected subsystems, then, provides mutual protection for multiple program complexes cooperating in the same computation and removes two limitations of facilities providing simple controlled sharing. It is clear from the description of protected subsystems that each must operate in its own domain. Implementing protected subsystems requires mechanisms that allow the association of more than one domain with a computation and also requires means for changing from one protection domain to another as control passes from one protected subsystem to another. The design must ensure that one protected subsystem cannot interfere in any way with the correct operation of another subsystem involved in the same computation.

We note in passing that the supervisor in most computer systems is an 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. Thus the protection mechanisms are protecting their own implementation. The resulting uniformity is consistent with the design principle of economy of mechanism.

In order to implement protected subsystems, then, there must be a way of associating multiple domains with a single computation. One way would be to use a separate virtual processor, each with its own domain, for each protected subsystem, a notion proposed by Dennis and Van Horn [41] and discussed by Lampson [30]. A computation involving multiple protected subsystems would require multiple cooperating virtual processors. The invocation of one protected subsystem by another, and the communication of any response, would be done using the interprocessor communication facilities of the system [67]. An implementation using multiple virtual processors, though 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 reason for the protected subsystems in a computation to be expressed as asynchronous activities, a single virtual processor implementation seems more natural. Such an implementation would require the association of multiple domains with a single virtual processor, a strategy proposed by LeClerc [68], [69] and explored in detail by Lampson [19], Schroeder [70],Needham [20],Sturgis [17], Jones [71], and Rotenberg [59] . In this case, communication among protected subsystems could be via interprocedure call and return operations.

The essence of changing domains is, in access control list terms, to change principal identifiers; in capability terms it is to acquire the set of capabilities of the new domain. In both cases, it is also essential that the virtual processor begin execution at some agreed-to starting point in the new domain.

Let us consider first an access control list implementation. Suppose we extend the possible permissions on a segment, as recorded in an access controller, to include ENTER permission, and add one more field to an access controller, the *domain identifier*, which is the principal identifier of the domain to be entered. The meaning of ENTER permission on a segment is that a virtual processor having only that permission may use (the first address in) that segment only as the target of a GO TO or CALL instruction. Further, upon executing a GO TO or CALL instruction, the processor will automatically pick up the domain identifier field in the access controller and use it as the principal identifier in transactions with the memory system.

We now have a controlled domain entry facility. A user wishing to provide a protected subsystem can do so by setting the access control lists of all objects that are to be internal parts of the system to contain one of his own principal identifiers. He also adds to the access control list of the initial procedure of his subsystem ENTER permission for any other principals who are allowed to use his protected subsystem.

In a capability system, a similar addition produces protected subsystems. The permission field of a capability is extended to include ENTER permission, and when a capability is used as the target of a GO TO or a CALL instruction, control is passed to the procedure in the segment pointed to by the capability. Simultaneous with passing control to the procedure, the processor switches on the READ permission bit of the capability, thereby making available to the virtual processor a new domain--all those objects that can be reached starting from capabilities found in the procedure.

Two mechanisms introduced earlier can now be seen to be special cases of the general domain entry. In the initial discussion of the capability system, we noted that the authentication system starts a new user by allowing a virtual processor to enter that user's domain at a controlled starting point. We could use the domain entry mechanism to accomplish this result as follows. A system program is "listening" to all currently unused terminals or system ports. When a user walks up to a terminal and attempts to use it, the system program creates a new virtual processor and has that processor ENTER the domain named by the prospective user. The entry point would be to a program, perhaps supplied by the user himself, which authenticates his identity before doing any other computation. Because a protected subsystem has been used, the program that monitors the unused terminals does not have access to the data in the protected subsystem (in contrast with the system of Fig. 7), a situation in better accord with the principle of least privilege. Instead, it has an enter capability for every domain that is intended to be entered from a terminal, but that capability leads only to a program that demands authentication.

We have sketched only the bare essentials of the mechanism required to provide domain switching. The full mechanics of a practical system that implements protected objects and subsystems are beyond the scope of this tutorial, but it is useful to sketch quickly the considerations those mechanisms must handle.

1. The principle of "separation of privilege" is basic to the idea that the internal structure of some data objects is accessible to virtual processor A, but only when the virtual processor 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 hold the other. Morris [72] has described an elegant semantics for separation of privilege in which the first capability is known as a *seal*. In terms of the earlier discussion of types, the type field of a protected object contains a seal that 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. This idea apparently was suggested by H. Sturgis. The HYDRA and CAL systems illustrate two different implementations of this principle.
2. The switching of protection domains by a virtual processor should be carefully coordinated with the mechanisms that provide for dynamic activation records and static (own) variable storage, since both the activation records and the static storage of one protection domain must be distinct from that of another. (Using a multiple virtual processor implementation provides a neat automatic solution to these problems.)
3. The passing of arguments between domains must be carefully controlled to ensure that the called domain will be able to access its arguments without violating its own protection intentions. Calls by value represent no special problem, but other forms of argument reference that require access to the original argument are harder. One argument that must be especially controlled is the one that indicates how to return to the calling domain. Schroeder [70] explored argument passing in depth from the access control list point of view, while Jones [71] 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. This area is in a state of rapid development, and 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

A. Implementations of Protection Mechanisms

Until quite recently, the protection of computer-stored information has been given relatively low priority by both the major computer manufacturers and a majority of their customers. Although research time-sharing systems using base and bound registers appeared as early as 1960 and Burroughs marketed a descriptor-based system in 1961, those early features were directed more toward preventing accidents than toward providing absolute interuser protection. Thus in the design of the IBM System/360, which appeared in 1964 [73], the only protection mechanisms were a privileged state and a protection key scheme that prevented writing in those blocks of memory allocated to other users. Although the 360 appears to be the first system in which hardware protection was also applied to the I/O channels, 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 [74]. Recently, descriptor-based architectures, which can be a basis for the more sophisticated protection mechanisms described in Section II, have become common in commercially marketed systems and in most manufacturers' plans for forthcoming product lines. Examples of commercially available descriptor-based systems are the IBM System/370 models that support virtual memory, the Univac (formerly RCA) System 7, the Honeywell 6180, the Control Data Corporation Star-100, the Burroughs B5700/6700, the Hitachi 8800, the Digital Equipment Corporation PDP- 11/45, and the Plessey System 250. On the other hand, exploitation of such features for controlled sharing of information is still the exception rather than the rule. Users with a need for security find that they must improvise or use brute force techniques such as complete dedication of a system to a single task at a time [75]. The Department of Defense guide for safeguarding classified information stored in computers provides a good example of such brute force techniques [76].

In the decade between 1964 and 1974, several protection architectures were implemented 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 M.l.T.'s CTSS which, in 1961, implemented user authentication with all-or-nothing sharing and, in 1965, added shared files with permission lists [12]. In 1967, the ADEPT system of the System Development Corporation 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 [14]. At about the same time, the IBM Cambridge Scientific Center released an operating system named CP/67, later marketed under the name VM/370, that used descriptor-based hardware to implement virtual System/360 computers using a single System/360 Model 67 [11]. In 1969, the University of California (at Berkeley) CAL system implemented a software-interpreted capability system on a Control Data 6400 computer [17]. Also in 1969, the Multics system, a joint project of M.I.T. and Honeywell, implemented in software and hardware a complete descriptor-based access control list system with hierarchical control of authorization on a Honeywell 645 computer system [26], [77]. 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 [78]. In 1970, the Berkeley Computer Corporation also implemented rings of protection in the BCC 500 computer [19]. In 1973, a hardware version of the idea of rings of protection together with automatic argument address validation was implemented for Multics in the Honeywell 6180 [63]. At about the same time, the Plessey Corporation announced a telephone switching computer system, the Plessey 250 [53], based on a capability architecture.

Current experimentation with new protection architectures is represented by the CAP system being built at Cambridge University [20] and the HYDRA system being built at Carnegie-Mellon University [21] . Recent research reports by Schroeder [70], Rotenberg [59], Spier et al. [79], and Redell [54] propose new architectures that appear practical to implement.

B. Current Research Directions

Experimentation with different protection architectures has been receiving less attention recently. Instead, the trend has been to concentrate in the following five areas: 1) certification of the correctness of protection system designs and implementations, 2) invulnerability to single faults, 3) constraints on use of information after release, 4) encipherment of information with secret keys, and 5) improved authentication mechanisms. These five areas are discussed in turn below.

A research problem attracting much attention today is how to certify the correctness of the design and implementation of hardware and software protection mechanisms. There are actually several sub-problems in this area.

a) One must have a precise model of the protection goals of a system against which to measure the design and implementation. When the goal is complete isolation of independent users, the model is straightforward and the mechanisms of the virtual machine are relatively easy to match with it. When controlled sharing of information is desired, however, 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 [14], [62] .

b) Given a precise model of the protection goals of a system and a working implementation of that system, the next challenge is to verify somehow that the presented implementation actually does what it claims. Since protection functions are usually a kind of negative specification, testing by sample cases provides almost no information. One proposed approach uses proofs of correctness to establish formally that a system is implemented correctly. Most work in this area consists of attempts to extend methods of proving assertions about programs to cover the constructs typically encountered in operating systems [52] .

c) Most current systems present the user with an intricate interface for specifying his protection needs. The result is that the user has trouble figuring out how to make the specification and verifying that he requested the right thing. User interfaces that more closely match the mental models people have of information protection are needed.

d) In most operating systems, an unreasonably large quantity of "system" software runs without protection constraints. The reasons are many: fancied higher efficiency, historical accident, misunderstood design, and inadequate hardware support. The usual result is that the essential mechanisms that implement protection are thoroughly tangled with a much larger body of mechanisms, making certification impossibly complex. In any case, a minimum set of protected supervisor functions--a protected kernel--has not yet been established for a full-scale modern operating system. Groups at M.I.T. [80] and at Mitre [81], [82] are working in this area.

Most modern operating systems are vulnerable in their reaction to hardware failures. Failures that cause the system to misbehave are usually easy to detect and, with experience, candidates for automatic recovery. Far more serious are failures that result in an undetected disabling of the protection mechanisms. Since routine use of the system may not include attempts to access things that should not be accessible, failures in access-checking circuitry may go unnoticed indefinitely. There is a challenging and probably solvable research problem involved in guaranteeing that protection mechanisms are invulnerable in the face of all single hardware failures. Molho [83] explored this topic in the IBM System 360/Model 50 computer and made several suggestions for its improvement. Fabry [84] has described an experimental "complete isolation" system in which all operating system decisions that could affect protection are duplicated by independent hardware and software.

Another area of research concerns constraining the use to which information may be put after its release to an executing program. In Section 1, we described such constraints as a fifth level of desired function. For example, one might wish to "tag" a file with a notation that any program reading that file is to be restricted forever after from printing output on remote terminals located outside the headquarters building.

For this restriction to be complete, it should propagate with all results created by the program and into other files it writes. 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 problem is to identify corresponding mechanisms for computer systems that could help enforce (or detect violations of) such agreements. Rotenberg explored this topic in depth [59] 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 research problem is how to communicate the keys to authorized users. If this communication is done inside the computer system, 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 [40] . Another research problem is development of encipherment techniques (sometimes called privacy transformations) 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 finding simple and fast techniques that also require much effort to cryptanalyze (that is, with high work factors) 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 [38]. One special difficulty in this area is that research in encipherment encounters the practice of military classification. Since World War II, only three papers with significant contributions have appeared in the open literature [27], [39], [85]; other papers have only updated, reexamined, or rearranged concepts published many years earlier.

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

Work in progress is not well represented by published literature. The reader interested in further information on some of the current research projects mentioned may find useful the proceedings of two panel sessions at the 1974 National Computer Conference [86], [87], a recent workshop [88], and a survey paper [89].

C. 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 help but 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, designers had confidence that the architectures being designed were complete enough to be useful. And it turned out that they were. Even so, it was quickly established that matching a problem statement to the architecture--programming--was a major effort whose magnitude was quite sensitive to the exact architecture. In a parallel way, matching 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 mechanisms have found them relatively clumsy to program and not especially well matched to the users' image of the problem to be solved, even though the mechanisms may be sufficient. As in the case of all programming systems, it will be necessary for protection systems to be used and analyzed and for their users to propose different, better views of the necessary and sufficient semantics to support information protection.

ACKNOWLEDGMENT

R. Needham, A. Jones, J. Dennis, J. P. Anderson, B. Lindsay, L. Rotenberg, B. Lampson, D. Redell, and M. Wilkes carefully reviewed drafts of the manuscript and offered technical suggestions. In addition, the preparation of this paper was aided by discussions with many people including H. Forsdick, P. Janson, A. Huber, V. Voydock, D. Reed, and R. Fabry. L. Schroeder ruthlessly edited out surplus jargon and prose inelegance.

SUGGESTIONS FOR FURTHER READING

The following short bibliography has been selected from the reference list to direct the reader to the most useful, up-todate, and significant materials currently available. Many of these readings have been collected and reprinted by L. J. Hoffman in [90]. The five bibliographies and collections (item 8 below) provide access to a vast collection of related literature.

1. Privacy and the impact of computers [1]-[3], [91], [92].
2. Case studies of protection systems [14], [17], [20], [26], [63], [83], [84].
3. Protected objects and protected subsystems [30], [45], [54], [59], [70]-[72].
4. Protection with encipherment [38]-[40], [93], [94].
5. Military security and nondiscretionary controls [82], [95], [96].
6. Comprehensive discussions of all aspects of computer security [6] - [8].
7. Surveys of work in progress [86]-[89] .
8. Bibliographies and collections on protection and privacy [90], [97]-[100].

[Norm Hardy, 1997: OCR performs poorly on material such as that below. Mail to norm@netcom.com might occasionally elicit help in distress situations.]

References are presented in order of First citation. The sections in which each reference is cited appear in parentheses following the reference. Section SFR is Suggestions for Further Reading.

[1] A. Westin, *Privacy and Freedom*. New York: Atheneum, 1967. (I-A1, SFR)

[2] A. Miller, *The Assault on Privacy*. Ann Arbor, Mich.: Univ. of Mich. Press, 1971; also New York: Signet, 1972, Paperback W4934. (I-A1, SFR)

[3] Dept. of Health, Education, and Welfare, *Records, Computers, and the Rights of citizens*. Cambridge, Mass.: M.I.T. Press, 1973. (I-A1, SFR)

[4] R. Turn and W. Ware, "Privacy and security in computer systems," I-A1 *Amer. Scientist*, voL 63, pp. 196-203, Mar.-Apr. 1975. (I-A1)

[5] W. Ware, "Security and privacy in computer systems," in *1967 S,CC, AFIPS Cont. Proc.*, vol. 30, pp. 287-290. (I-A1)

[6] J. Anderson, "Information security in a multi-user computer environment," in *Advances in Computers*, vol. 12. New York: Academic Press, 1973, pp. 1-35. (I A1, SFR)

[7]J. Martin, *Scurity. Accuracy, and Primey in Computer Systems*. Englewood Cliffs, N.J.: Prentice-Hall, 1973. (I-A1, SFR) I

[8] R. Patrick, *Security Systems Review Manual*. Montvale, N.J.: AFIPS Press, 1974. (I-A1, SFR)

[9]G. Bender, D. Freeman, and J. Smith, "Function and design of DOS/360 and TOS/360," *IBM Syst. J.*, vol. 6, pp. 2-21, 1967. (I-A2)

[10] R. Hargraves and A. Stephenson, "Dodge considerations for an educational time-sharing system," in *1969 SJCC, AFIPS ConJ. Proc.*, vol. 34, pp. 657-664. (I-A2)

[11] R. Meyer and L. Seawright, "A virtual machine time-sharing system," *IBM Syst. J.*, vol. 9, pp. 199-218, 1970. (I-A2, I-B3, III-A)

[12] M l.T. Computation Center, *CTSS Programmer's Cuide*, 2nd ed. Cambridge, Mass.: M.I.T. Press, 1965. (I-A2, III-A)

[13] D. Stone,"PDP-IOsystem concepts end capabilities," in *PDP10 Applicanons in Science*, vol. II. Maynard, Mass: Digital Equipment Corp., undated (ca. 1970), pp. 32-55. (I-A2)

[14] C. Weissman,"Security controls in the ADEPT-SO time-sharing system " in *1969 FJCC, AFIPS Conf. Proc.*, vol. 3S, pp. 119-133. (I-A2, II-C5, III-A, III-B, SFR)

[15] D. Bobrow et al., "TENEX, a paged time sharing system for the PDP-10," *Commun. ACM*, vol. 15, pp. 135-143, Mar. 1972. (I-A2, II-C3)

[16] F. Corbato, 1. Saltzer, and C. Clingen, "Multics--The first seven years" in *1972 SJCC, AFIPS Conf. Proc.*, vol. 40, pp. 571-583. (I-A2)

[17] H. Sturgis, "A postmortem for a time sharing system," Ph.D. dissertation, Univ. of Calif., Berkeley, 1973. (Also available as Xerox Palo Alto Res. Center Tech. Rep. CSL74-1.) (I-A2, II-C2, II-E, III-A, SFR)

[18] D. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. ACM*, vol. 17, pp. 365-375, July 1974. (I-A2, II-C3)

[19] B. Lampson, "Dynamic protection structures," in 1969 *FJCC, AFIPS Conf. Proc.*, vol. 35, pp. 27-38. (I-A2, II-E, III-A)

[20] R. Needham, "Protection systems and protection implementations," in *1972 FJCC, AFIPS Conf. Proc.*, vol. 41, pt. 1, pp. 571-578. (I-A2, II-B3, II-E, III-A, SFR)

[21] W. Wulf etal., "HYDRA: The kernel of a multiprocessor operating system," *Commun. ACM*, vol. 17, pp. 337-345, June 1974. (I-A2, II-B3, III-A)

[22] R. Conway, W. Maxwell and H. Morgan, "On the implementation of security measuies in information systems," *Commun. ACM*, vol. 15, pp. 211-220, Apr. 1972. (I-A2)

[23] 1. Reed, "The application of information theory to privacy in data banks," Rand Corp., Tech. Rep. R-1282-NSF, 1973. (I-A2)

[24] D. Hsiso, D. Kerr, and F. Stahl, "Research on data secme systems," in *1974 NCC, AFIPS Conf. Proc.*, vol. 43, pp. 994-996. (I-A2)

[25] L. Hoffman and W. Miller, "Getting a petsonal dossier from a statistical data bank," *Datamation*, vol. 16, pp. 74-75, May 1970. (I-A2)

[26] J. Saltzer, "Protection and the control of information sharing in Multics," *Commun. ACM*, vol. 17, pp. 388-402, July 1974. (I-A3, 1-B4, III-A, SFR)

[27] P. Baran, "Security, secrecy, and tamper-free considerations,’ *On Distributed Communications,* no. 9, Rand Corp. Tech. Rep. RM-3765-PR, 1964. (I-A3, III-B)

[28] G. Popek, "A principle of kernel design," in *l974 NCC, AFIPS Conf. Proc.*, vol. 43, pp. 977-978. (I-A3)

[29] D. Hollingsworth, "Enhancing computer system security," Rand Corp. Paper P-S064, 1973. (I-A3)

[30] B. Lampson, "Protection," in *Proc. 5th Princeton Symp. Informanon Science and Systems* (Mar. 1971), pp. 437-443. (Reprinted in *ACM Operating Syst. Rev.*, vol. 8, pp. 18-24, Jan. 1974.) (I-B1, II-B2, II-E, SFR)

[31] E. Codd etal., "Multiprogramming Stretch: Feasibility considerations," *Commun. ACM*, vol. 2, pp. 13-17, Nov. 1959. (I-B3)

[32] W. Lonergan and P. King, "Design of the B5000 system," *Datamation*, vol. 7, pp. 28-32, May 1961. (I-B3, I-B5)

[33] G. Popek and R. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412-421, July 1974. (I-B3)

[34] R. Goldberg, "Architecture of virtual machines," in *1973 NCC, AFIPS Conf. Proc.*, vol. 42, pp. 309-318. (I-B3)

[35] G. Purdy, "A high security log-in procedure," *Commun. ACM*, vol. 17, pp. 442-445, Aug. 1974. (I-B4)

[36] A. Evans, W. Kantrowitz and E. Weiss, "A user authontication scheme not requiring secr;cy in the computer," *Commun. ACM*, vol. 17, pp. 437-442, Aug. 1974. (I-B4)

[37] M. Wilkes, *Time-Sharing Computer Systems*, 2nd ed. New York: American-Elsevier, 1972. (I-B4, I-B5, II-A)

[38] J. Smith, W. Notz, and P. Osseck, "An experimental application of cryptography to a remotely accessed data system," in *Proc. ACM 25th Nat. Conf.*, pp. 282-298, 1972. (I-B4, III-B, SFR)

[39] H. Feistel, "Cryptographic coding for data bank privacy," IBM Corp. Res. Rep. RC 2827, Mar. 1970. (I-B4, III-B, SFR)

[40] D. Branstad, "Security aspects of computer networks," in *AIAA Computer Network Systems Conf.* (Apr. 1973), Paper 73-427. (I-B4, I-B5, III-B, SFR)

[41] J. Dennis and E. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, pp. 143155, Mar. 1966. (I-B5, II-B1, II-E)

[42] J. Dennis, "Segmentation and the design of multiprogrammed computer systems," *J. ACM,* vol. 12, pp. 589-602, Oct. 1965. (I-B5, II-A)

[43] R. Daley and J. Dennis, "Virtual memory, processes, and sharing in Multics," *Commun. ACM*, vol. 11, pp. 306-312, May 1968. (I-B5)

[44] R. Watson, *Timesharing System Design Concepts.* New York: McGraw-Hill, 1970. (II-A)

[45] R. Fabry, "Capabilitybased addressing," *Commun. ACM*, voL 17, pp. 403-412, July 1974. (II-A, SFR)

[46] E. Organick, *The Multics System: An Examinanon of its Structure.* Cambridge, Mass.: M.I.T. Press, 1971. (II-A)

[47] ---, *Computer System Organization. TheB5700/B6700Serics.* New York: Academic Press, 1973. (II-A, II-B3)

[48] W. Ackerman and W. Plummer, "An implementation of a multiprocessing computer system," in

Proc. ACM Symp. Operanag System Principles (Oct. 1967), Paper D-3. (II-B1)

[49] R. Fabry, "Preliminary description of a supervisor for a machine oriented around capabilities," *Inst. Comput. Res. Quart. Rep.*, vol. 18, sec. IB, Univ. of Chicago, Aug. 1968. (II-B1)

[50] J. Iliffe and J. Jodeit, "A dynamic storage allocation scheme," *Comput. J.*, vol. 5, pp. 200-209, Oct. 1962. (II-B1)

[51] E. A. Feustel, "On the advantages of tagged architecture," *IEEE Trans. Comput.*. vol. C-22, pp. 644-656, July 1973. (II-B1)

[52] L. Robinson *et al.*, "On attaining reliable software for a secure operating system," in *Int. Conf. Reliable Software* (Apr. 1975), pp. 267-284. (II-B3, III-B)

[53] D. England, "Capability concept mechanism and structure in system 2s0," in *IRIA Int. Workshop Protection in Operating Systems* (Aug. 1974), pp. 63-82. (II-B3, III-A)

[54] D. Redell, "Naming and protection in extendible operating systems," Ph.D. dissertation, Univ. of Calif., Berkeley, 1974. (Available as M.I.T. Proj. MAC Tech. Rep. TR-140.) (II-B3, III-A, SFR)

[55] A. Bensoussan, C. Clingen, and R. Daley, "The Multics virtual memory: Concepts and design," *Commun. ACM*, vol. 15, pp. 308-318, May 1972. (II-B3, II-C3)

[56] B. W. Lampson, W. W. Lichtenberger, and M. W. Pirtle, "A user machine in a time-sharing system," *Proc.* IEEE, vol. 54, pp. 1766-1774, Dec. 1966. (II-C3)

[57] H. Bingham, "Access controls in Burroughs large systems," *Privacy and Security in Computer Systems,* Nat. Bur. Stand. Special Pub. 404, pp. 42-45, Sept. 1974. (II-C3)

[58] R. Daley and P. Neumann, "A general-purpose file system for secondary storage," in *1965 FJCC, AFIPS Conf. Proc.*, vol. 27, pt. I, pp. 213-229. (II-C4)

[59] L. Rotenberg, "Making computers keep secrets," Ph.D. dissertation, M.I.T., Cambridge, Mass., 1973. (Also available as M.I.T. Proj. MAC Tech. Rep. TR-115.) (II-C4, II-C5, II-E, III-A, III-B, SFR)

[60] K Walters et al., "Structured specification of a security kernel," in *Int. Conf. Reliable Software,* Los Angeles, Calif., pp. 285-293, Apr. 1975. (ii-C5)

[61] B. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 1 6, pp. 613-615, Oct. 1973. (II-C5)

[62] D. Bell and L. LaPadula, "Secure computer systems," Air Force Elec. Syst. Div. Rep. ESD-TR-73-278, vols. 1, 11, and III, Nov. 1973. (II-C5, III-B)

[63] M. Schroeder and J. Saltzer, "A hardware architecture for implementing protection rings," *Commun. ACM*, vol. 15, pp. 157-170, Mar. 1972. (II-C5, III-A, SFR)

[64] J. Fenton, "Memoryless subsystems," *Comput. J.*, vol. 17, pp. 143-147, May 1974. (II-C5)

[65] O. Dahl and C. Hoare, "Hierarchical program structures," in *Structured Programming*. New York: Academic Press, 1972, pp. 175-220. (II-E)

[66] D. Branstad, "Privacy and protection in operating systems,"*Computer*, vol. 6, pp. 43-46, Jan. 1973. (II-E)

[67] P. Brinch-Hansen, "The nucleus of a multiprogramming system," *Commun. ACM*, vol. 13, pp. 238-250, Apr. 1970. (II-E)

[68] J. LeClerc, "Memory structures for interactive computers," Ph.D. dissertation, Univ. of Calif., Berkeley, May 1966. (II-E)

[69] D. Evans and J. LeClerc, "Address mapping and the control of access in an interactive computer," in *1967 SJCC, AFIPS Conf. Proc.*, vol. 30, pp. 23-30. (II-E)

[70] M. Schroeder, "Cooperation of mutually suspicious subsystems in a computer utility," Ph.D. dissertation, M.I.T., Cambridge, Mass., 1972. (Also available as M.I.T. Proj. MAC Tech. Rep. TR-104.) (II-E, III-A, SFR)

[71] A. Jones, "Protection in programmed systems," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa.. 1973. (II-E, SFR)

[72] J. Morris, "Protection in programming languages," *Commun. ACM*, vol. 16, pp. 15-21, Jan. 1973. (II-E, SFR)

[73] G. Amdahl, G. Blaauw, and F. Brooks, "Architecture of the IBM System/360," *IBM J. Res Develop.*, vol 8, pp. 87-101, Apr. 1964. (III-A)

[74] IBM Corp., "System 370/Principles of operation," IBM Corp. Syst. Ref. Lib. GA22-7000-3, 1973. (III-A)

[75] R. Bisbey, II, and G. Popek, "Encapsulation: An approach to operating system security," in *Proc. ACM 1974 Annul Conf.*, pp. 666-675. (III-A)

[76] Dept. of Defense, *Manual of Techniques and Procedures for Implementing, Deactivating, Testing, and Evaluatiing Secure Resource-Sharing ADP Systems*, DOD5200.28-M, Aug. 1972. (III-A)

[77] R. Graham, "Protection in an information processing utility," *Commum ACM*, vol. 11, pp. 36s-369, May 1968. (III-A)

[78] S. Motobayashi, T. Masuda, and N. Takahashi, "The Hitac 5020 time-sharing system," in *Proc. ACM 24th Nat. Conf.*, pp. 419-429, 1969. (III-A)

[79] M. Spier, T. Hastings, and D. Cutler, "An experimental implementation of the kernel/domain architecture," *ACM Operating Syst. Rev.*, vol. 7, pp 8-21, Oct. 1973. (III-A)

[80] M.I.T. Proj. MAC, "Computer systems research," in *Project MAC Progress Reporr XI: July 1973 to June 1974*, pp. 155-183. (III-B)

[81] E. Burke, "Synthesis of a software security system," in *Proc. ACM 1974 Annu. Conf.*, pp. 648-650. (III-B)

[82] W. Schiller, "Design of a security kernel for the PDP-11/45," Air Force Elec. Syst. Div. Rep. ESD-TR-73-294, Dec. 1973. (III-B, SFR)

[83] L. Molho, "Hardware aspects of secure computing," in *1970 SJCC, AFIPS Conf. Proc.*, vol. 36, pp. 135-141. (III-B, SFR) 199

[84] R. Fabry, "Dynamic verification of operating system decisions," *Commun. ACM*, vol. 16, pp. 659-668, Nov. 1973. (III-B, SFR)

[85] C. Shannon, "Communication theory of secrecy systems," *Bell Syst. Tech. J.*, vol. 28, pp. 656-715, Oct. 1949. (III-B)

[86] S. Lipner, Chm., "A panel session--Security kernels," in *1974 NCC, AFIPS Conf. Proc.*, vol. 43, pp. 973-980. (III-B, SFR)

[87] R. Mathis, Chm., "A panel session--Research in data security--Policies and projects," in *1974 NCC, AFIPS Conf. Proc.*, vol. 43 pp. 993-999. (III-B, SFR)

[88] Institut de Recherche d'Informatique et d'Automatique (IRIA), *Int. Workshop Protectlon in Operating Systems.* Rocquencourt, France: IRIA, Aug. 1974. (III-B, SFR)

[89] J. Saltzer, "Ongoing research and development on information protection," *ACM Operating Syst. Rev.*, vol. 8, pp. 8-24, July 1974. (III-B, SFR)

[90] L. Hoffman Ed., *Security and Privacy in Compurer Systems.* Los Angeles, Calif.: Melville Pub. Co., 1973. (SFR)

[91] C. W. Beardsley, "Is your computer insecure?" *IEEE Speetrum*, vol. 9, pp. 67-78, Jan. 1972. (SFR)

[92] D. Parker, S. Nycom, and S. Oura, "Computer abuse," Stanford Res. Inst. Proj. ISU 2501, Nov. 1973. (SFR)

[93] D. Kahn, *The Codebreakers*. New York: Macmillan, 1967. (SFR)

[94] G. Mellen, "Cryptology, computers, and common sense," in *1973 NCC, AFlPS Conf. Proc.* ,vol.42, pp.569-579. (SFR)

[95] J. Anderson, "Computer security technology planning study," Air Force Elec. Syst. Div. Rep. ESD-TR-73-51, Oct. 1972. (SFR)

[96] W. Ware *et al.*, "Security controls for computer systems," Rand Corp. Tech. Rep. R-609, 1970. (Classified confidontial.) (SFR)

[97] R. Anderson and E. Fagerlund, "Privacy end the computer: An annotated bibliography," *ACM Compur. Rev.*, vol. 13, pp. 551-559, Nov. 1972. (SFR)

[98] J. Bergart, M. Denicoff, and D. Hsiao, "An annotated and cross-referenced bibliography on computer security and access control in computer systems," Ohio State Univ., Computer and Information Science Res. Center Rep. OSU-CISRC-T072-12, 1972. (SFR)

[99] S. Reed and M. Gray, "Controlled accessibility bibliography," Nat. Bur. Stand. Tech. Note 780, June 1973. (SFR)

[100] J. Scherf, "Computer and data base security: A comprehensive annotated bibliography," M.I.T. Proj. MAC Tech. Rep. TR-122, Jan. 1974. (SFR)

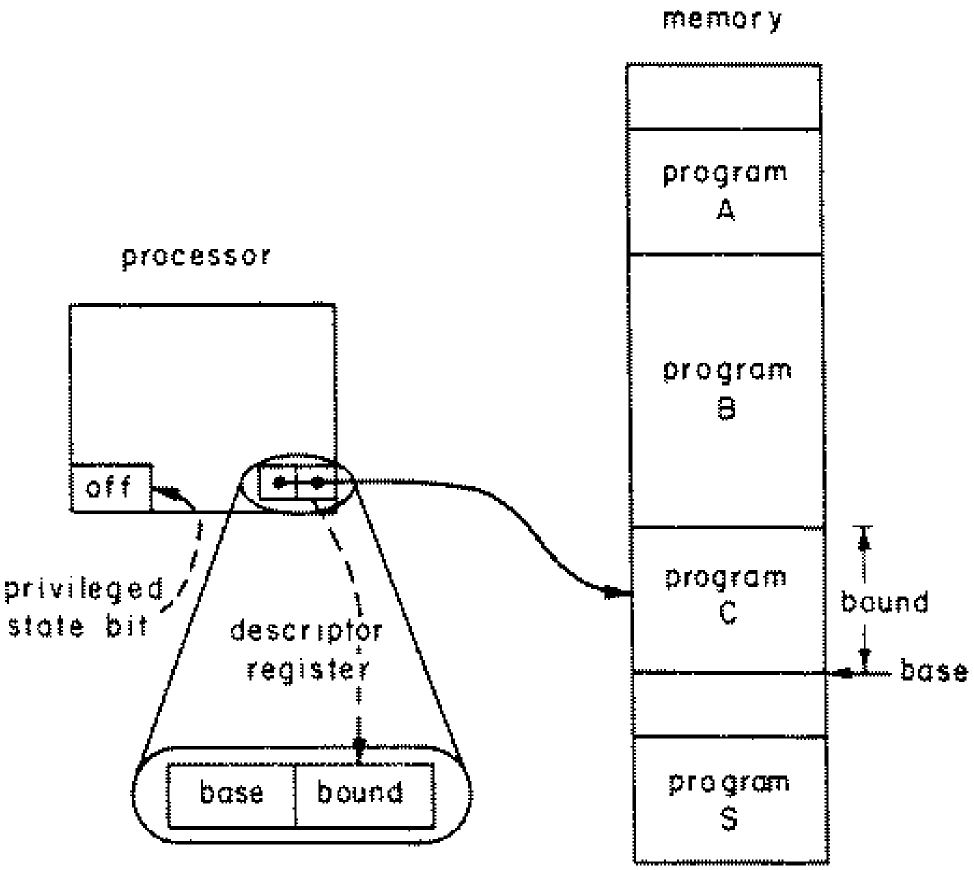


Fig. 1. Use of a descriptor register to simulate multiple virtual machines. Program C is in control of tho processor. The privileged state bit has value OFF, indicating that program C is a user program. When program S is running, the privileged state bit has value ON. In this (and later) figures, lower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

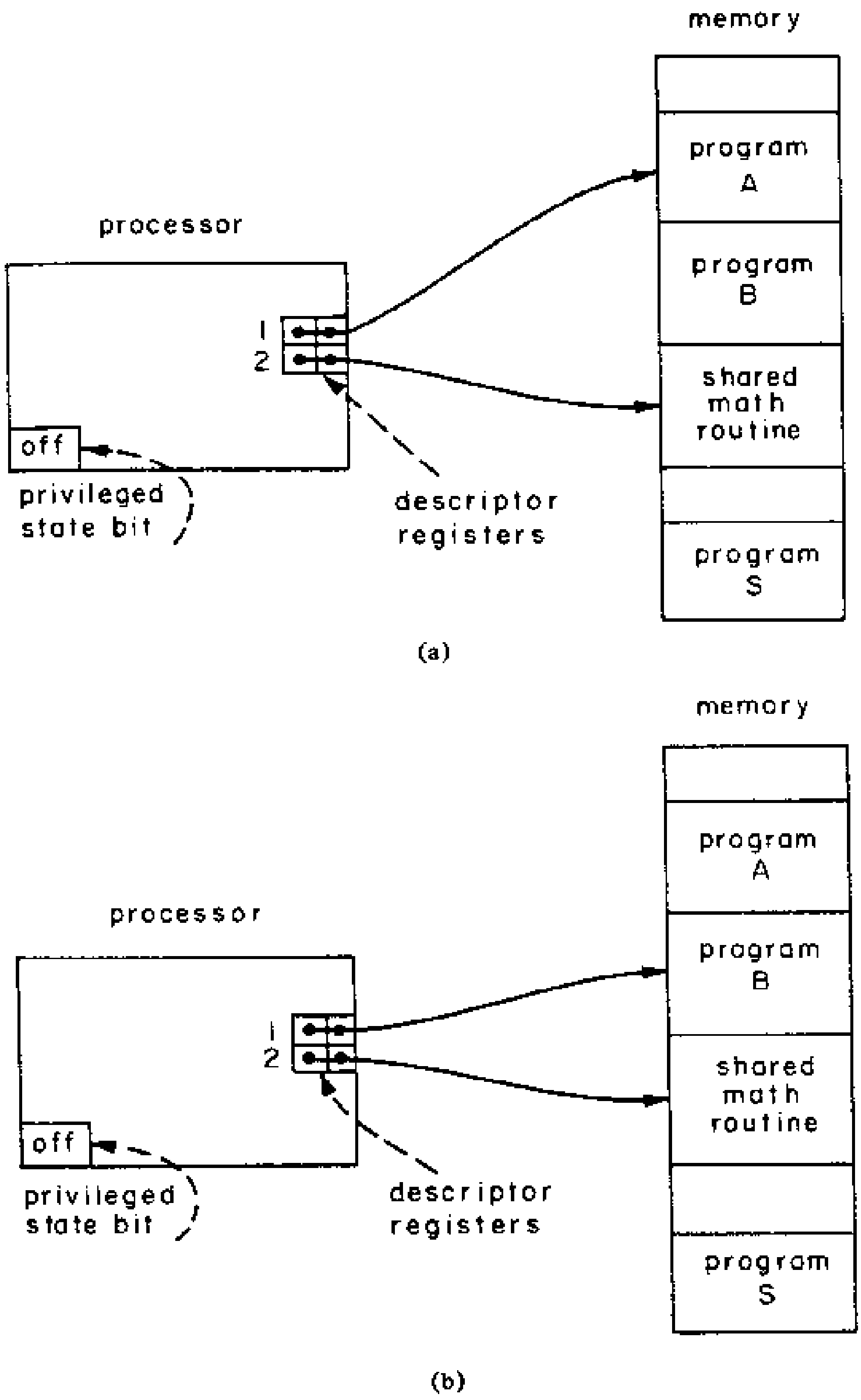


Fig. 2. Sharing of a math routine by use of two descriptor registors. (a) Program A in control of

processor. (b) Program B in control of processor.

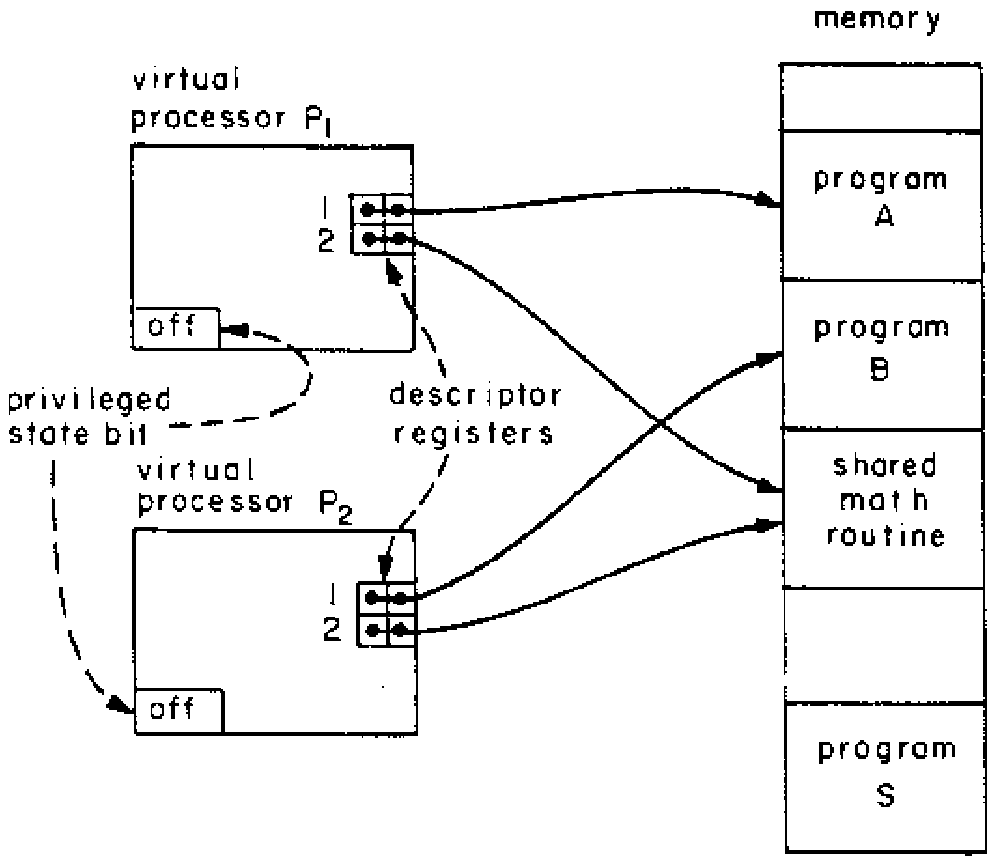


Fig. 3. Fig. 2 redrawn to show sharing of a math routine by two virtual processors simultaneously.

Fig. 4. A descriptor containing READ and WRITE permission bits.

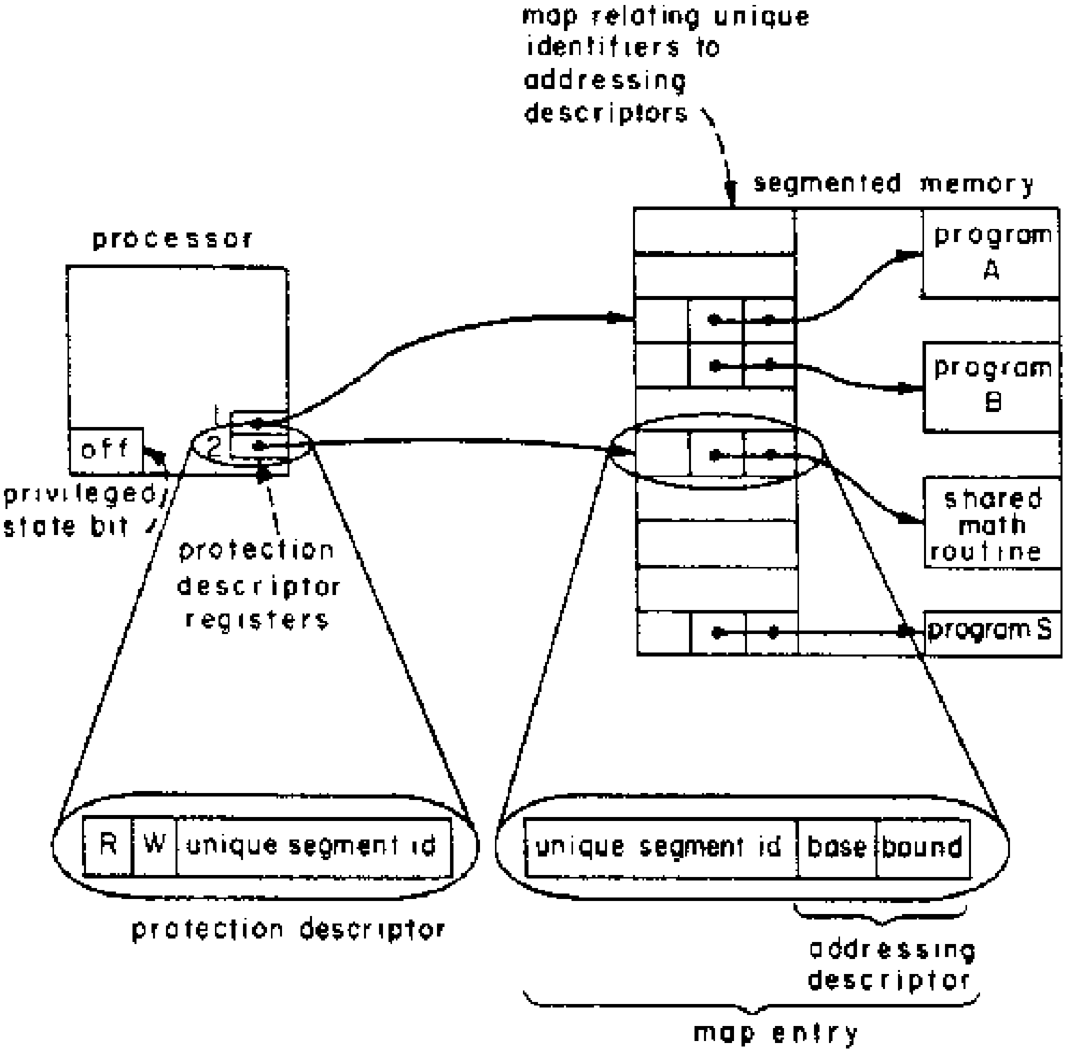


Fig. 5. An organization separating addressing from protection descriptors, using a segmented memory. The address passed from the processor to the memory consists of two parts: a unique segment identifier and an offset. Program A is in control. (Compare with Fig. 2(a).) In later figures the map containing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

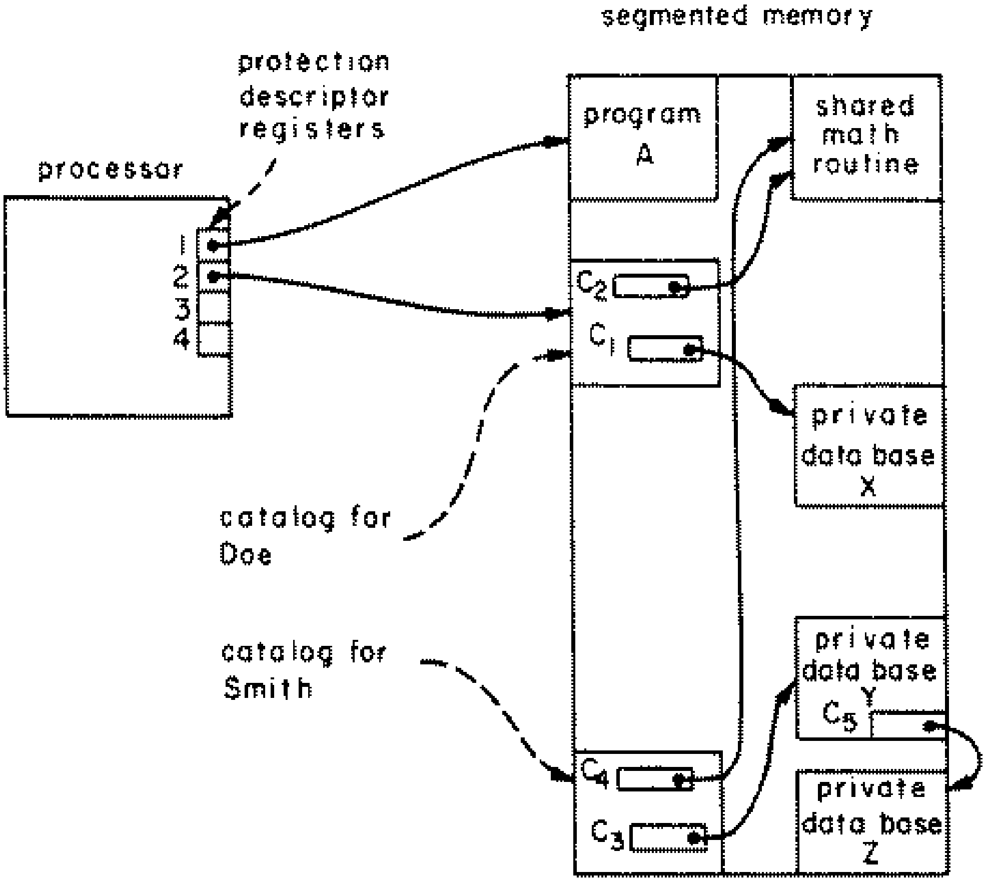


Fig. 6. A simple capability system. Program A is in control of the processor. Note that there is no way for the processor to address Smith's catalog or data base Y. On the other hand, data base X could be accessed by loading capabUity C~ into a protection descriptor register. CapabUity C' is loadable because it is stored in a segment that can be reached from a capabUity already loaded in protection descriptor register 2. Note also that the former function of the privUeged state bit has been accomplished by protecting the capabilities. The privileged state bit also has other uses and wUI be reintroduced later.

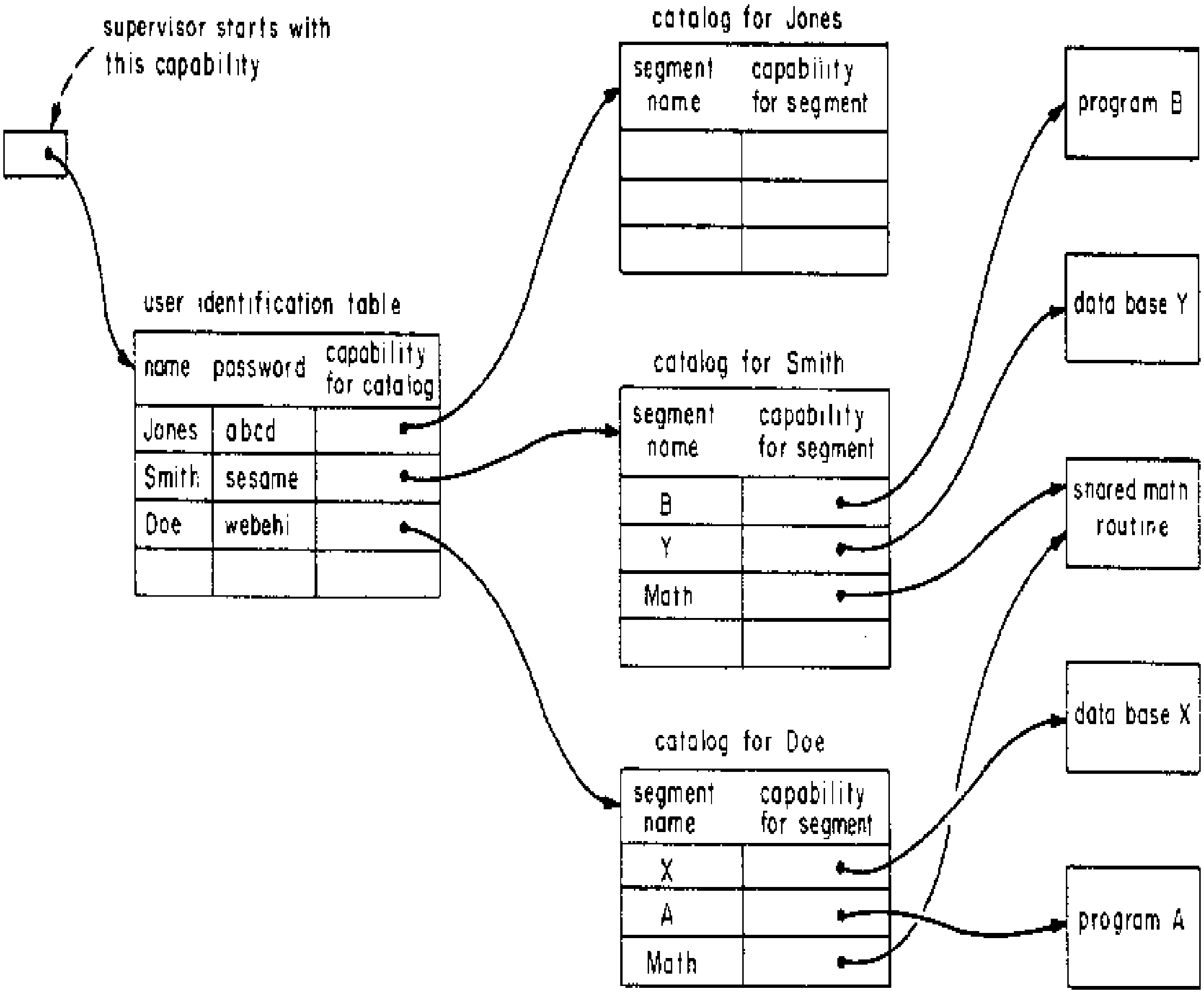


Fig. 7. A capability system with provision for authentication.

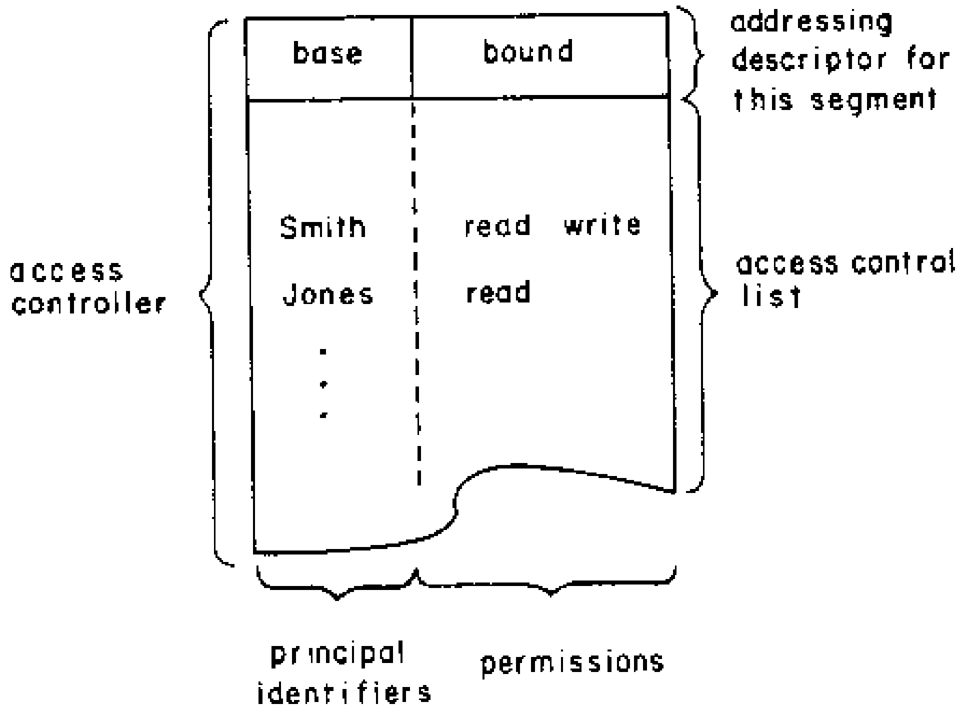


Fig. 8. Conceptual model of an access controller. When a virtual processor attempts to refer to the segment associated with the access controller, the memory system looks up the principal identifier in the access control list part. If found, the permissions associated with that entry of the access control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

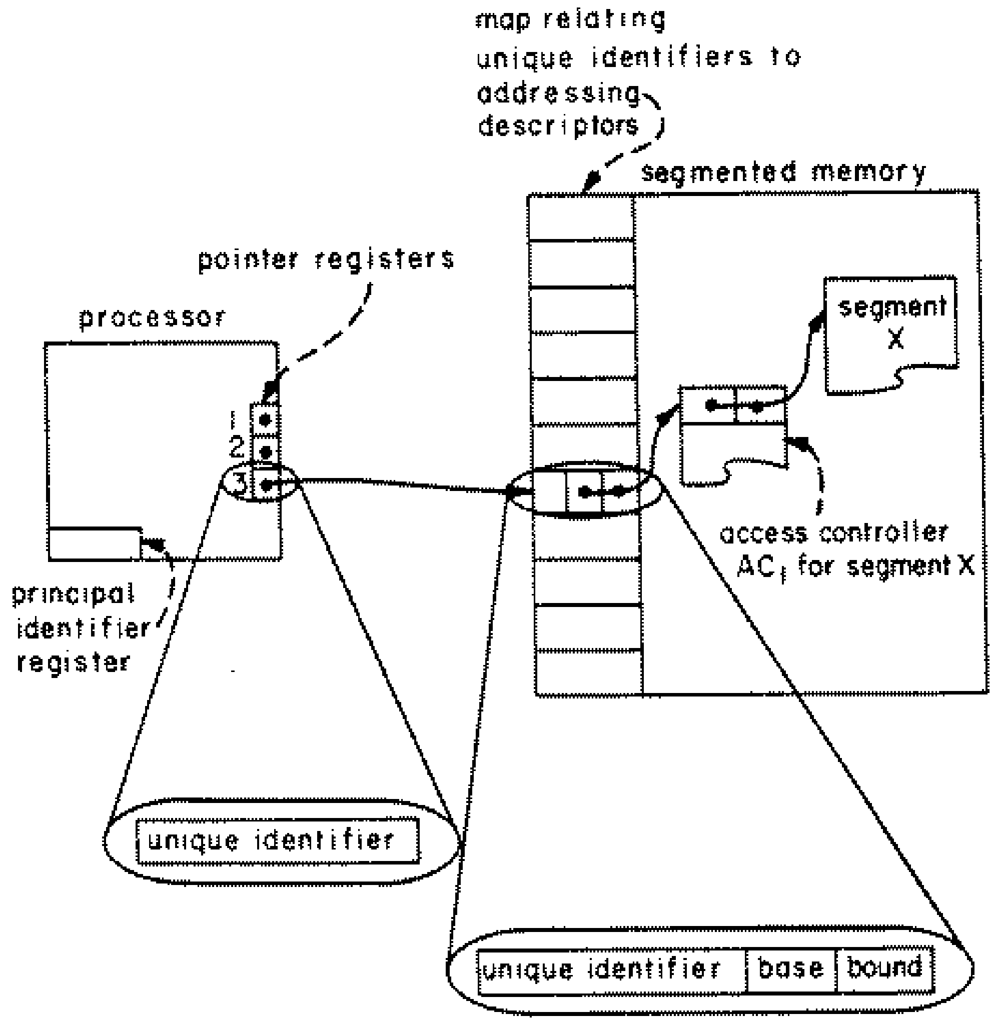


Fig. 9. A revision of Fig. 5, with the addition of an access controller as an indirect address to be used on all references by the processor to the memory. Since the access controller contains permission bits, they no longer need appear in the processor registers, which have been renamed "pointer" registers.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

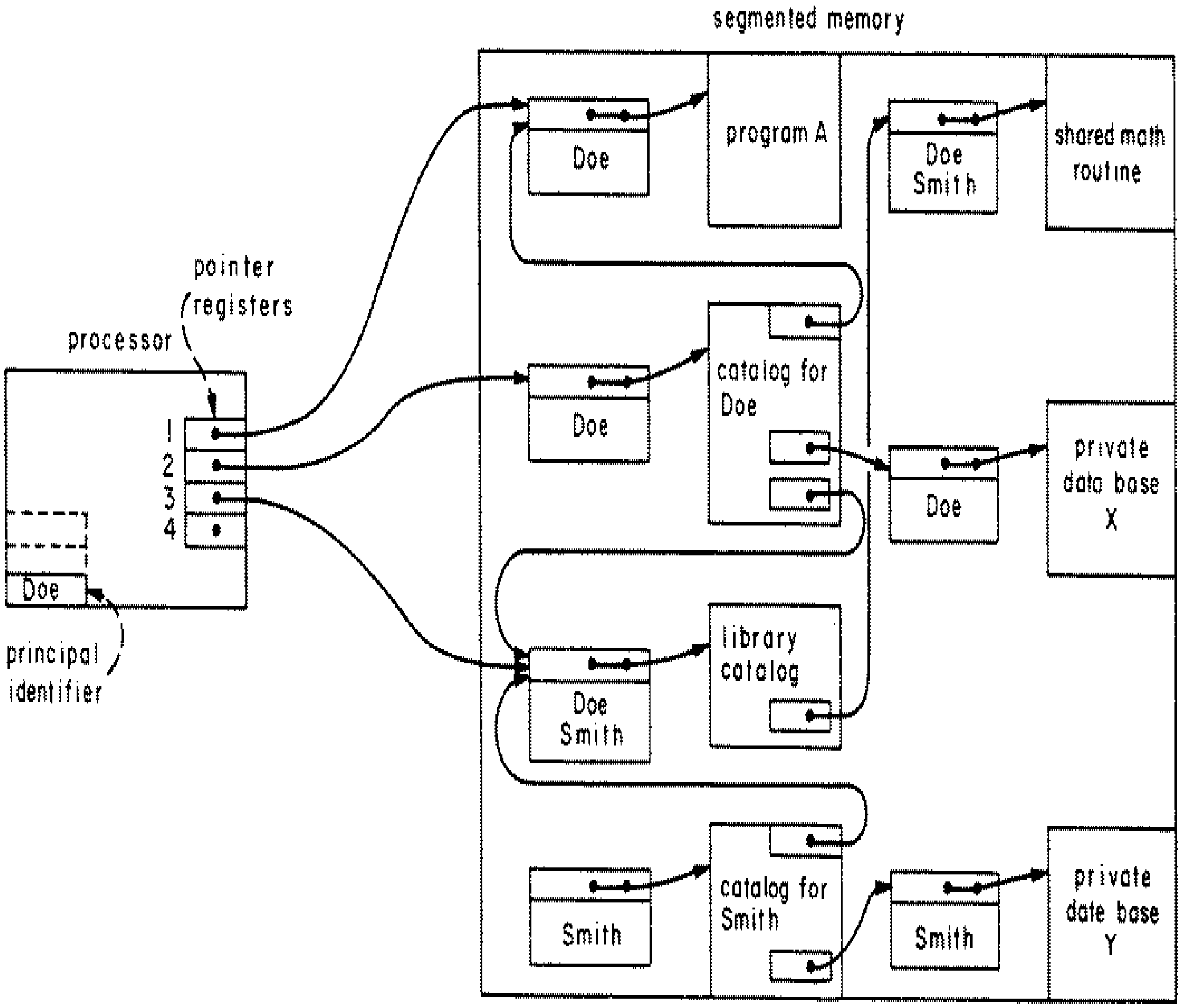


Fig. 10. A protection system using access controllers containing access control lists. In this system, every segment has a single corresponding access controller with its own unique identifier for addressing purposes; pointer registers always contain the unique identifiers of accoss controllers. Program 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 access 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 accoss control list of the math routine, the processor will then be able to use the shared math routine.

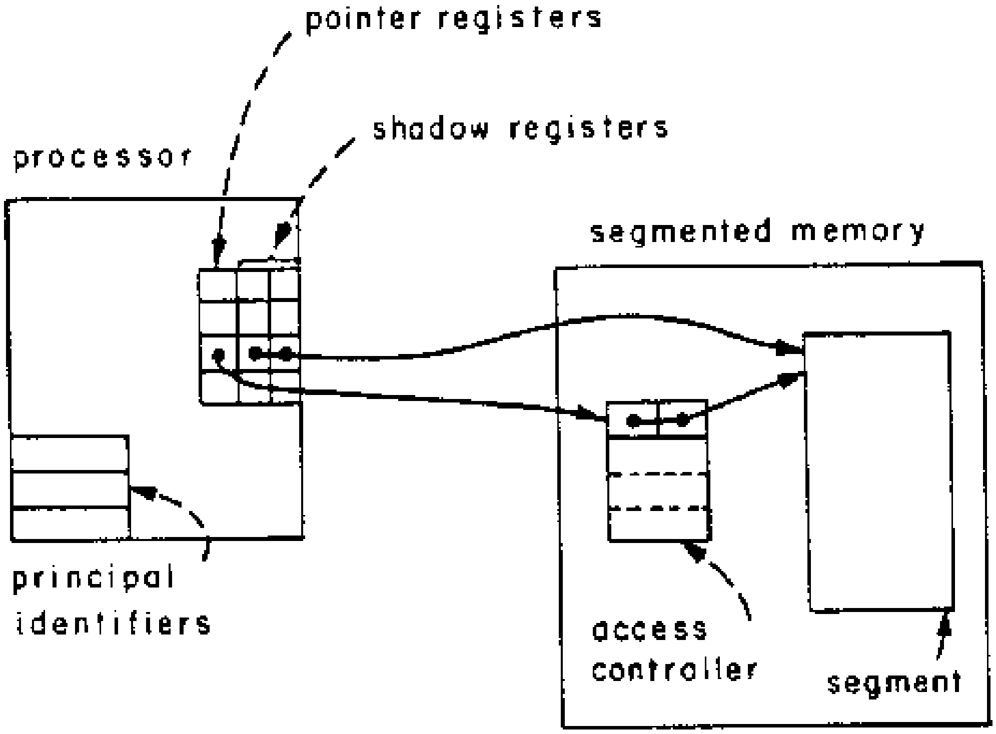
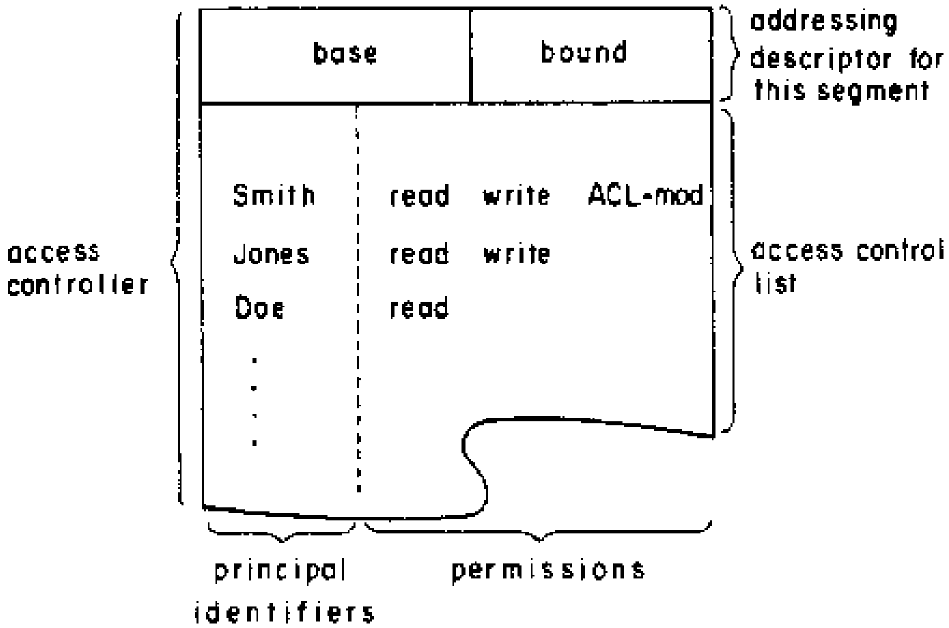


Fig. 11. Use of "shadow" capability registers to speed up an access control list system. When a pointer register containing a unique identifier is flrst wed, the shadow register is automatically loaded from the access controller to which the unique identifier refers. Later wes of that pointer register thw do not require reference to the access controller. Storing of a pointer register means storing of the unique identifier only; the shadow register is never stored.



Flg. 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 modfications to the access control list of this access controller. Jones cannot modify the access control list, even though he can read and write in the segment described by this access controller. Doe is even more constrained.

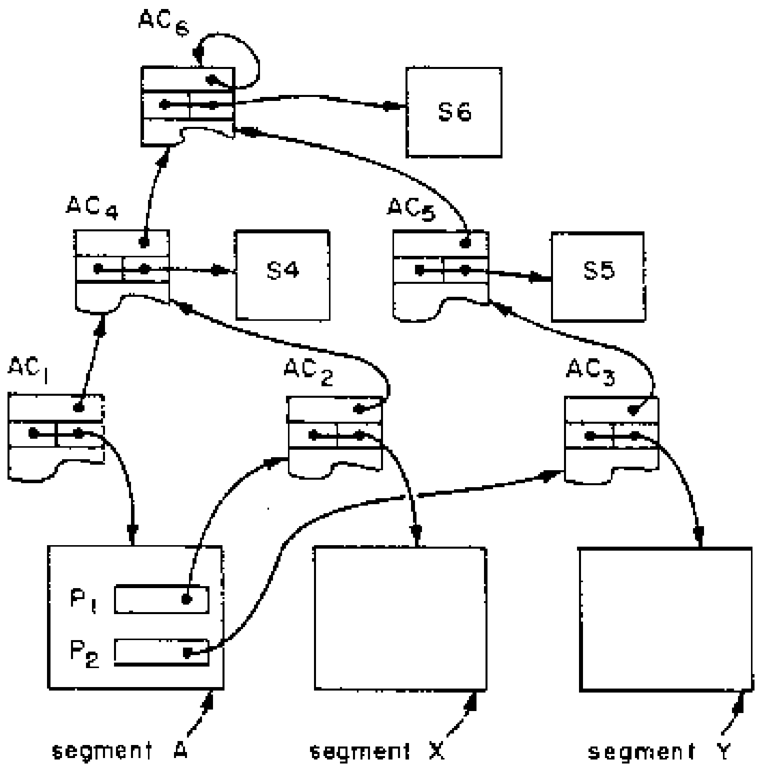


Fig. 13. Hierarchical control of authority to modify access control lists. Each access controller has an extra field in addition to those of Fig. 12; the extra field contains the unique identifier of some higher level access controller. Authority to access segments A, X, and Y is controlled by access 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 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_1, is that if a user attempts to modify the access control list of AC_1, the backpointer is foUowed, leading to AC_1. Only if the user's principal identifier h 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 P_1 and P_2 found in segment A.

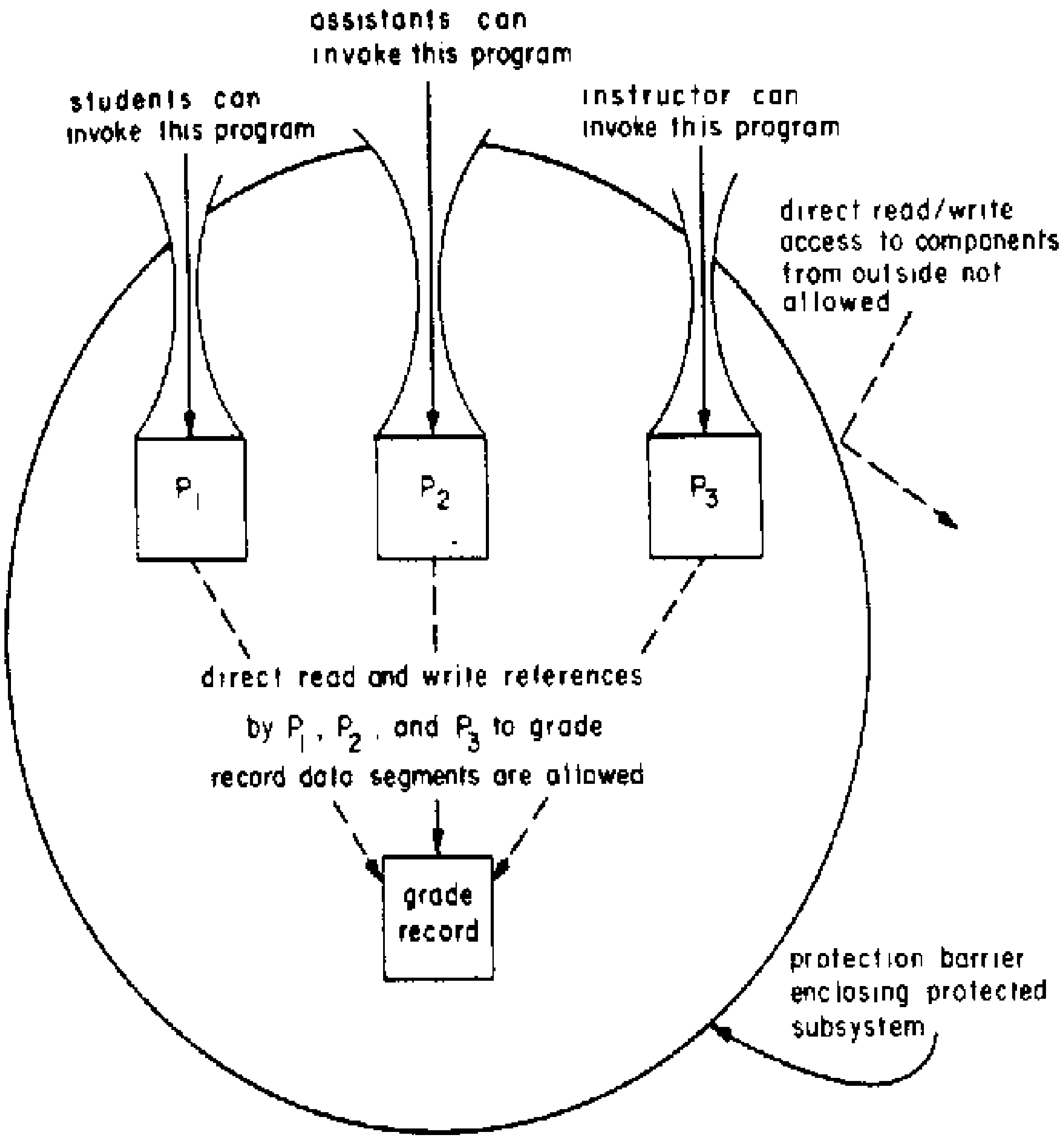


Fig. 14. A protected subsystem to implement the grade-keeping system described in the text. P_1, which can be invoked by all students in the subjoct, is programmed to return the caller's grade for a particular assignment or the distribution of all grades for an assignment. P_2 which can be invoked by 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.