PROTECTION AND ACCESS CONTROL IN OPERATING SYSTEMS

Butler W. Lampson

In *Operating Systems*, Infotech State of the Art Report 14, 1972, pp 311-326

INTRODUCTION

I should like to explain what protection and access control is all about in a form that is general enough to make it possible to understand all the existing forms of protection and access control that we see in existing systems, and perhaps to see more clearly than we can now the relationships between all the different ways that now exist for providing protection in a computer system. Just in case you are not aware of how many different ways there are, let me suggest a few that you might find in a typical system:

1 a key switch on a console
2 a monitor and user mode mechanism in the hardware of the machine that decides whether a given program can execute input/output instructions
3 a memory protection scheme that attaches 4-bit tags to each block of memory and decides whether or not particular programs can address that block
4 facilities in the file system for controlling access; such a system allows a number of entities, normally referred to as users, to exist in the system and controls the way in which data belonging to one user can be accessed by other users.

All these mechanisms are mostly independent, and superficially they look almost completely different. However, it turns out that there is a fair amount of unity at the basic conceptual level in the way in which protection is done, although there is enormous efflorescence in implementations.

I should like to make it clear precisely what I am not talking about in discussing protection of computer systems. I am concerned only about what goes on inside the system; the question of how the system identifies someone who proposes himself as a user I do not intend to consider, although it is of course an important problem. If someone walks up to a Teletype terminal, sits down, and says, "My name is Joe Bloggs. I'd like to use the system." the system is then confronted with the question of whether it accepts him as Joe Bloggs. Once he has been accepted as Joe Bloggs, inside the system there is a certain set of rules that are applied to decide exactly what Joe Bloggs will be allowed to do. That set of rules in a sense operates within its own world, a world over which the system designer has complete control and can, if he wants to badly enough and can assume total lack of failures in the hardware, make perfect. If you design carefully, you can construct a system that is logically clean and you can say precisely what the rules will be and which kinds of access will and will not be allowed.

The problem of identifying Joe Bloggs is completely separate from this; it is much more akin to the problem that is normally referred to by military organizations as the security problem. Identification consists of determining that the impulses coming into the machine along a particular wire emerge from some source that is authorized to be Joe Bloggs. There are a lot of techniques for dealing with this: passwords are used, terminals are locked up in rooms, data is encoded. However, these techniques are totally different from what is required inside the computer system, because outside the machine you do not have complete control: you are dealing with the real, physical world where a person can be hit on the head and telephone lines can be tapped. Inside the machine, the author of the system, at least in principle, can exercise control over what the user programs are doing.

CONTEXTS

There is a basic idea underlying the whole apparatus of protection and access control; it is the idea of a program being executed in a certain context that determines what the program is authorized to do. For example, if you have a program running on System/360 hardware, there is a bit in the physical machine that determines whether the machine is in supervisor state or in problem state. Thus, two contexts are established by the physical hardware. The hardware then enforces cerain rules about what can be done in each context. In the supervisor state, you can execute input/output instructions, do XPSWs, and so on. In the problem state, you are not allowed to do those things. To give another example, the System/360 hardware also implements most of the machinery necessary for a 16-context system, by means of the 4-bit protection key that is put on each block of memory. There are 4 bits in the register and there are 4 bits on each block of memory; if you are executing in problem state, the 4 bits in the register must match the 4 bits of memory, or else some other obscure condition must be true. In that way you can have 16 different protection contexts, depending upon which of the 16 different possible 4-bit combinations you put in the register. Depending upon what is in the register, the program being executed will be able to access different regions of the memory. However, in this case, the hardware alone does not provide the control; you also need some software. In fact, a fair amount of software is needed to set up the information in all the memory registers and so on. So here hardware plus some of the operating system defines these 16 contexts. In something like OS/360 MVT, for example, this enables 15 tasks to be run simultaneously; the sixteenth is used up for the supervisor.

To give yet another example, in a time sharing system, it is customary for the system to keep a file directory for each user. In the directory is a list of files and in the files is the user's data. In a simple system, let us say, there is no way for one user to get at any other user's data. Therefore, each user constitutes a protection context defined by the time sharing system, and within that context programs can access that user's files but cannot access any other files.

In all the examples I have given, except the first one of the supervisor and problem states, there is no way in which a program itself can initiate a change from one context to another. If you have a user program running in a time sharing system, for example, you cannot stipulate that you want to become another user; no such facility is provided. Only the system can decide to stop running one particular user and start running another. It is only the system that can change the context.

On the other hand, for the hardware, in the problem state/supervisor state example, there is a mechanism for changing contexts: namely, if you are in the probelm state, you can execute an SVC and get into supervisor state; if you are in supervisor state you can execute an XPSW and get into problem state. This is an example I will come back to later when discussing how to pass control from one protection context to another.

So, you cannot have protection without this fundamental idea of some definition of what the program can do at any given time. That is exactly what we mean by protection: that it has been specified somewhere that a program can do certain things and that there are other things that it cannot do.

Many different words have been used for this idea. The people on the MULTICS project at MIT use the term *rings*. Other people have used the term *protection spheres*. I will use the term *domains* because I want a word that is fairly neutral and does not carry geometrical implications about the way in which the things are related; as we will see, it is not necessary to have any fixed ideas about how domains are related.

This idea of a protection domain constitutes a fundamental building block; it is the context within which a program executes and it determines what the program will be allowed to do. That context is defined by some machine or system on which the program is running and, as is always the case when talking about how contexts and environments are defined, there are many different levels at which you can look at it. For example, in the cases we were discussing before, we talked about the two domains that are defined by the System/360 hardware, and then we talked about the large number of domains that are defined by a time sharing system that might be running on that hardware. Thus, depending upon which level you look at, you will see different kinds of domains, different kinds of things that can be accessed or not accessed, defined by the underlying system.

CONTROL TRANSFER

Given that we have the idea of protection domains, the other basic thing that we need is some mechanism for transferring control from one domain to another; in other words, we need some way of getting out of a certain restricted region, that is, out from under the control of a certain set of rules about what you can access, over to being under the control of a different set of rules.

Let me go back to the problem state/supervisor state example to illustrate two aspects of this question of control transfer. If the supervisor is running and it wants to transfer control to the user, what does that mean? The supervisor thinks that it is all powerful; it now wants to give control to something that is not going to be so powerful, that is, it wants to restrict the available set of actions that can be performed by the program to which control is to go. So, there is a taking away of control, a taking away of power, a taking away of facilities when the supervisor transfers control to the user. The supervisor simply sends control to a user program that then does not have all the great powers that the supervisor has; the user program is operating in a more restricted domain. That is a very simple kind of control transfer.

In the other direction, matters are not quite so simple, because there is a subtle point involved, namely, that if you want to switch from problem state to supervisor state then you cannot just say that you want to transfer control from a user program to the supervisor. I know of a machine that has a fine instruction that causes a transfer of control and a switch to be made from problem state to supervisor state. Unfortunately, that is not very useful, because it does not provide any protection. Obviously, it enables you to switch control from problem state to supervisor state and to send control to the next instruction, and then you would go on executing in supervisor state, which means that you have complete freedom to transfer from problem state to supervisor state and no particularly useful purpose is served by having the two domains in the first place.

So, the important thing that we see in looking at this example is that the mechanism for transferring control from one domain to another cannot be completely free; it has to be constrained or you will have no protection.

There are a lot of ways to constrain such control transfers. Again, hardware offers us a very simple example. If, on System/360, you want to send control to the supervisor, ignoring the question of what happens if you make an error, one way to do it is to execute a *supervisor call* instruction. That does two things: it switches from the problem domain to the supervisor domain and it sends control to a fixed location in the supervisor. This fulfils the essential requirement that you do not allow unrestricted switching from one domain to another. In particular, the switching from one domain to another is accompanied by some definite rule about where execution continues; in this case, execution continues at a fixed point in the supervisor where a particular piece of code is located. The supervisor retains control of the situation because it is not possible to send control to an arbitrary place in the supervisor, but to only this one fixed place.

Thus we have seen the example of transferring from a big domain, the supervisor, to one that has less power, the problem state domain, and an example of transferring from a small domain to a bigger one. It should be fairly clear that the same mechanism that works for transferrring from a small domain to a bigger one could work also for transferring from one domain to another one that is related to the first in some arbitrary way. The essential thing is that you do not have complete freedom, in fact that you do not have any freedom at all, to decide where execution will continue after the domain has been switched. It is the new domain that must have that control. There are, of course, exceptions, for example, the supervisor giving control to a user. In this case, the supervisor has complete freedom in deciding where exactly the control is to go.

THE MESSAGE SYSTEM

I should now like to illustrate these ideas, and show how completely general protection can be obtained, by means of a very simple mechanism, which I shall describe in terms of an idealized system called the *message system*. This system turns out to be more or less copied from the basic structure of the RC 4000 multiprogramming system at Regnecentralen, which was described in the Communications of the ACM (Brinch Hansen P *The nucleus of a multiprogramming system*, Comm ACM 13, 4 (April 1970) pp238-241). However, the description given there was for totally different purposes and it might be viewed as an accident that it serves my purpose here.

This is a very straightforward system. We think of a lot of independent processes and we identify each process with a domain. We are going to say that nothing is shared between these processes. So, if it helps you to think of it that way, you may think of a lot of independent computers, each with its own memory and input/output units. So far, we have no communication and so the protection is very straightforward; you have all these domains and no mechanism for passing control from one domain to another. There is no protection problem, because there is no sharing. On the other hand, as there is no sharing, there is no way of communicating from one domain to another, which is not very satisfactory. Usually, you need to be able to communicate from one domain to another, and the purpose of a protection mechanism is to allow that kind of communication, to allow somebody to call on somebody else to do something for him: for example, to allow a user program to call on a supervisor to initiate an input/output transfer.

Let us see how such communication could work in the system that I am describing. Figure 1 shows the situation we have. There are a number of independent processes (domains D_1, D_2,...), each one of which runs completely separately from all the others. Memory is not shared; input/output devices are not shared; nothing is shared.

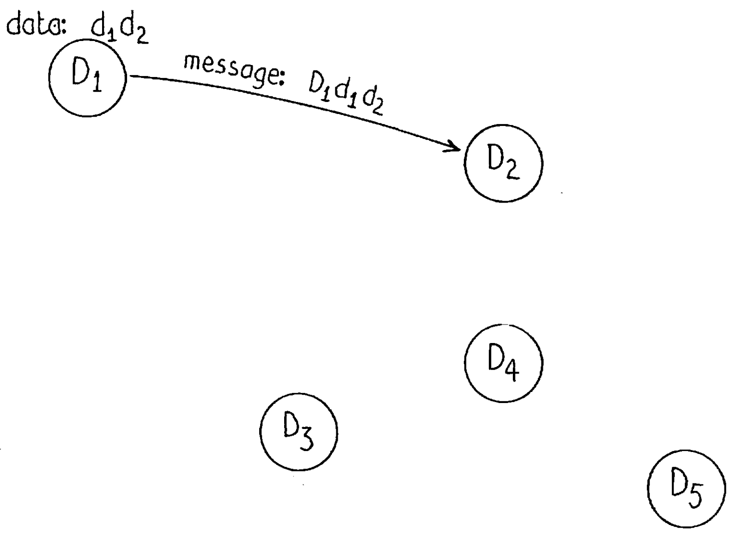

Figure 1: Domains in the message system

The communication mechanism will be a very simple one. Any domain can send a message to any other domain. A message consists of two parts: a string of data (d_1 d_2 in Figure 1) preceded by a tag that identifies the domain that sent the message. When a domain wants to send some data to another domain, the system absorbs that data and puts it into the queue of messages for the target domain together with the identifier of the domain that has sent it. When the target domain is ready to receive that message, it gets the identifier of the sending domain followed by the string of data.

With this mechanism you can implement an arbitrary protection system. Furthermore, if you want the most complete generality of protection, you will not be able to get by with anything less. Obviously, in terms of minor details concerning whether of not the processes are really independent, you can do better; but conceptually you need this basic structure of completely independent things that communicate via these messages.

Now I shall discuss how, with this system, I can model the kind of protection system to which we are accustomed. For instance, a typical program may run along for a while, then make a supervisor call to get some input/output transfers done, then make another supervisor call to open a file, and so on.

The first thing to observe is that with a scheme like this I can model subroutine calls very easily. For example, let us consider how D_1 can call D_2. D_1 sends a message containing a set of parameters to D_2. When D_2 receives the message, it recognizes that the message constitutes a subroutine call and executes the subroutine with the parameters received in the message. When the subroutine execution is finished, D_2 returns by sending a message containing the return value(s) to D_1.

The important thing about this scheme is that the subroutine call is a protected one. Let us examine why this is so. First, it is not possible for D_1 to send control to an arbitrary place, because the passing of control is not achieved by stipulating a jump to some location, but rather by sending a message and it is the interpretation of the message that decides what is actually going to happen. Interpretation is done by the receiving domain, which therefore has complete control over what exactly is to happen. For example, when D_2 gets the message from D_1, it looks at the parameters and, on the basis of what the parameters say, decides what to do. So, we have a protected entry in the same way as for the supervisor call, where we had protection because control always went to the same place. In the case of this subroutine call, control always goes to the same place in the sense that it always goes to the place where D_2 is waiting for a message. There is no mechanism by which D_1 can force D_2 to transfer control. The only thing D_1 can do is send a message and hope for the best; it is a matter of interpretation what is to be done with that message.

The return is protected for the same reason. There is no way in which D_2 can return to the wrong place when he finishes, because the return again consists of passing a message. D_1 is waiting for the message, and when it arrives D_1 can recognize that it is the return and continues. There is no way in which one domain can force another domain to send control to some unwanted place.

Let us consider the case where something goes wrong in a domain. For instance, suppose D_1 is a supervisor and D_2 is a user program. It is possible for the user program to go into an infinite loop; but it is not acceptable for the supervisor to lose control and so a protection mechanism for passing control between these domains has to be able to deal with such a situation. This is easily arranged. As soon as the system has a reliable process, say D_3, then the supervisor (D_1) can send D_3 a message asking D_3 to send a return message after a length of time, say 5 milliseconds, within which the user program (D_2) should have completed its task and returned control to D_1. Then, if D_1 receives the message from D_3 before hearing from D_2, he knows that something has gone wrong with D_2 and can decide what corrective action to take.

Finally, a most important thing to remember is that there are a lot of domains in the system. Domain D_2 may be expecting an authorized call from D_1, but not from D_4. What if D_4 tries to call D_2 by sending it a message? For example, suppose that in a time sharing situation a certain user says that he wants to open a certain file; how is he prevented from telling the system to open a file belonging to some other user, which according to the rules, is not to be allowed? This is the reason for putting identification on the message. The identification is the key to giving the protection system flexibility. The domain that is being called knows which domain is calling him, because the basic underlying protection system provides that information in guaranteed form. In the example above, D_2 is in a position to decide whether or not it wants to do the things it is being asked to do by D_1. If D_1 is making an unauthorized request then D_2 can find out about that by making whatever internal decisions are necessary and can then refuse to fulfil the call. For example, if D_1 is authorized to ask for magnetic tape units but D_4 is not, and D_2 receives a request to read 50 words from magnetic tape, then the first thing D_2 does is to look and see whether it was D_1 that made that request. If so, D_2 reads the 50 words; if D_4 made the request, D_2 does not fulfil it.

Since D_2 is a program, it can make such a decision in an arbitrarily complex way. It may make the decision in a very simple way; for example, if D_1 asks D_2 to open the file *alpha* belonging to user *X* then D_2 may have a simple table saying that domain D_1 is user *X*, domain D_2 is user *Y*, and domain D_3 is user *Z*. So if D_1, which is user *X*, asks for user *X*s file to be opened, that is fine; on the other hand, if user *Z* asks for user *X*s file to be opened, that is no good. Of course, there might be some more complicated rule. For example, in a real life situation such as a large data base managed by domain D_2, there might be a rule stipulating that a certain class of users is not allowed to access the data base more than 50 times a day, (That may sound ridiculous but in fact it is not; there are examples of collections of data with the property that for particular users to know any small amount of the data is not important, but for them to know a lot of it could constitute a serious security threat.) The existence of such a protection requirement means that you must have a system with flexibility. There is no way that you will be able to build into some fixed scheme a way of dealing with a situation like that described above unless you know beforehand exactly what will be required.

The message system is intended to model the most general kind of protection, and I claim that you cannot do any better than this if you really need generality. On the other hand, this model is not acceptable in practice because it is too hard to use. There are two things wrong with it. One of them is that there is no way for one domain to force another domain to do something, which makes debugging impossible. Debugging depends on the idea that somebody is more informed than somebody else; in other words, the person doing the debugging is more important than the program that is being debugged. He is allowed to force the program that is being debugged to do things, and so you need some mechanism for forcing domains into certain states.

The other much more serious thing that is wrong with this scheme is that as described it is unusable in the same way as the hardware delivered by the manufacturer is unusable. You have to have assemblers, compilers, loaders, and a lot of conventions for using the basic hardware because, although the basic hardware has a tremendous amount of power, the power is not organized in the right way. Similarly, on top of a general protection structure like the one I have described, you have to put some conventions so that the system knows about users and files, or about data bases and the number of times that they can be accessed, or whatever. Otherwise, each new user has to write the protection routines that decide who can access his files, which is an unreasonable amount of work for him to have to do. Such a scheme requires some standard facilities, and some standard way of talking about protection; the scheme cannot be as general as the one I have described but it should cover the majority of cases. However, we must remember that we do not want to give up the ability to do things in the most general way in the cases where that is required.

Before continuing on this line I should like to discuss a particular aspect of the scheme illustrated in Figure 1 that often causes a lot of trouble. Suppose that D_1 and D_2 are domains belonging to two users of a time sharing system and that D_1 wishes to use one of D_2's files. Suppose also that this use has been authorized by the two users, Bill (D_1) and John (D_2), say, having an agreement. John has agreed that Bill can get hold of the contents of one of his files. Bill tells his domain, D_2, to send a message to D_1 to get the file. The message tells D_2 that it is from D_1. The problem is this: how does D_2 know that D_1 is the authorized domain? Internally, these domains are just identified in an arbitrary way. The only requirement on the internal identification is that each domain have a different one.

Now we have the following problem: granted that it is possible to distinguish all possible senders, how does a receiver know what the *meaning* of a code is. Here is a simple but complete solution. Let us say that Bill asks his domain, D_1, to give him its identifying number. (We assume that each domain can find out its own identifying number.) Let us suppose that for D_1 it is 23. Then Bill can shout across the room to John that the identification number of his domain D_1 is 23. Then John can tell that to his domain D_2. Then, when 23 comes to D_2 as the domain identification in a message, D_2 knows that it can allow the identified domain access to the file in question.

You may think that this is no good, that 23 is like a password and Bill has given it away by shouting it across the room. But it is not like a password. No one is made any more powerful by knowing that Bill's number is 23. The 23 is significant only when it appears as the identification of a message, and the message identifications are always supplied by the system. The system will never let 23 appear as a message identification except when the message is set up by domain D_1. Hence the only way in which anyone can make use of the information that 23 is the identification for D_1 is to grant certain privileges to messages with that identification that they otherwise would not grant. Users other than Bill cannot masquerade as the domain with identification 23, because there is no way that they can get a 23 stuck on as a message identifier. The identification is always stuck on by the system and the essential feature of this protection scheme is a guarantee that it always sticks on the right identification.

In most existing systems there is no good analogy to this identification scheme; most of them do not have this kind of generality. For example, in the case of the supervisor-user situation, most existing systems do not provide anything like the scheme I discussed. When the user does a supervisor call, control goes to location 5, or wherever, and the supervisor itself has to remember which user it was running; the hardware does not give any help.

THE OBJECT SYSTEM

I am now going to describe a system called the *object system*, which is a specialization of the message system I have described. The object system is not as generalized as the message system but it is still a generalization of most of the protection facilities that actually exist.

With the object system, we envisage the world as being composed of objects: objects are whatever needs to be protected. A protection system is not so much concerned with what constitutes an object as with remembering certain facts about the relationship between objects and domains. From the protection system's point of view an object is simply a number, say, a 64-bit number, or at least a number large enough to make it possible for the system to guarantee that we will never have the same number for two different objects.

We can construct a two-dimensional matrix, as shown in Figure 2, with domains along one dimension and objects along the other. The function of the protection system now is to maintain this matrix and to perform certain operations on it. Each entry in the matrix can contain a list of what we call *access attributes*, which essentially are intended to describe what kind of access a particular domain can have to a particular object. For example, some typical access attributes are shown in Figure 2: D_1 has access attributes READ to object O_1, and D_2 has access attributes READ and WRITE to object O_1.

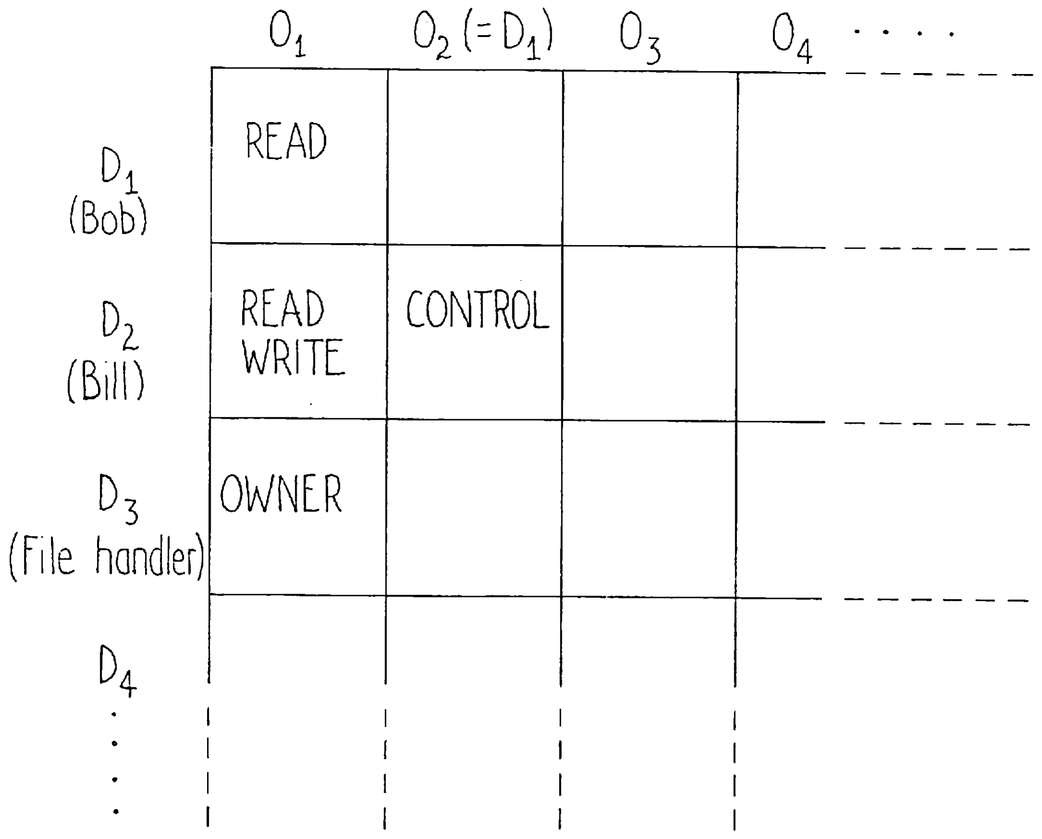

Figure 2: Matrix of access attributes

From the point of view of the protection system these attributes do not have any significance, they are just strings of characters. All the protection system is doing is remembering these attributes. This provides a set of conventions that allow you to do certain kinds of protection more conveniently than you could with the pure message system. The only thing a protection system can do with these attributes is remember them, and sometimes it adds new attributes or deletes old ones, according to certain rules. That is really the entire function of the protection system: to maintain this matrix.

Users of a protection system interpret these strings, put them in, and take them out, according to their own interests. For example, suppose that D_3 is a file handler and that D_1 and D_2 are users, called Bob and Bill respectively. Then suppose that Bill wants to create a file. He calls the file handler and tells it that he wants to create a file. The file handler tells the protection system that it wants to create an object. The protection system then tells the file handler that it can have object O_1 and puts OWNER as an attribute in the matrix (see Figure 2). (OWNER is one of three attributes that the protection system actually understands for itself; all others it simply considers to be character strings.)

OWNER means, in a sense, that the object belongs to the domain, and the domain can do whatever it wants with that object. In particular, if you are the owner of an object you can add access attributes for that object to the matrix. For example, let us say that it is a convention that when you create a file you get READ access and WRITE access to it. This means that, in the example of Bill creating a file, the file handler tells the protection system to put in READ and WRITE as access attributes for Bill with respect to the file O_1.

You will notice that the file handler was interested in the strings READ and WRITE. What does that mean? Let us see what happens. The file handler has created the file required by Bill. In addition, the file handler makes entries for this object in its own internal tables; also, it may do something to the disk in order to create the file, but that has nothing to do with the protection system. The protection system knows only that it has created and assigned a new 64-bit number and it tells the file handler what that number is. The file handler has a table that correlates O_1 with the relevant disk unit. The protection system knows nothing about disk units.

At some later time, if Bill wants to read from that file, he makes a call to the file handler and tells it that he wants to read from file O_1. The first thing the file handler does is to look in his entry for O_1 and ask whether Bill has a READ entry; if not, Bill cannot read from the file; if there is such an entry, the file handler looks up O_1 in its own tables, finds the disk address, and figures out what to do.

That is the basic scheme of things: we are trying to separate out the data handling required for protection from everything else. For implementation, this is not so convenient; it might be much more convenient to keep the disk address at the top of the matrix column rather than in a table that the file handler has to look up. But from an abstract viewpoint, that would be bad because it confuses the protection system with the file system; logically, the kinds of thing you do with protection are the same for lots of different kinds of objects and you do not want the protection system confused with other systems.

Given a matrice of the sort shown in Figure 2, there is a set of rules on how entries in the matrix can be modified:

1 an owner of an object can add access attributes for that object to the attributes for other domains;
2 a domain can copy across the access attributes it has itself for any given object to other domains;
3 a domain can delete access attributes from any other domain it controls;
4 an owner of an object can delete access attributes for that cbject.

Application of rule 2 as it stands can turn out to be inconvenient and so a *copy flag* is defined to specify whether or not an attribute can be copied.

Rule 3 has to do with the problem that I mentioned earlier; we need to have some way of controlling domains so that debugging is possible. In particular, some objects might themselves be domains. Domains need to have the protection system applied to them just like everything else, so domains will be objects just like everything else. For example, object O_2 might actually be equal to domain D_1, and then if D_2 has the attribute CONTROL for O_2 this means that domain D_2 controls D_1, which happens to be object O_2. Therefore, D_2 can take away access attributes from D_1.

Rule 4 is a matter of debate. In most systems, if one owns an object and can give other people access to it, then one can take that access away; in extreme cases one can even delete the object. However, some people (notably Jack Dennis and his students) argue that such an approach is a mistake and that once you have granted access to an object to someone you should not be allowed to take that access away; in some sense, you made a contract with him to the effect that he has access to that object and you cannot take it away without his consent.

Notice that deleting access with CONTROL means that you delete access from the rows of the matrix; deleting access with OWNER means that you delete access from the columns. Ownership refers to an object; control refers to a domain.

This scheme corresponds fairly closely to schemes that are in common use for allowing access to files and processors in operating systems. Some care has been taken to make it sufficiently general to cover all the various implementations of access control.

Now I will briefly suggest two popular implementations of this matrix. The obvious implementation as a two-dimensional array, shown in Figure 2, is clearly not acceptable, because there are lots of objects and the matrix is very sparse.

The two popular implementations I wish to mention correspond to storing the matrix by rows and storing it by columns.

The implementation that stores the matrix by rows uses what is known as a *capability list;* for each domain there is a list of object-attribute pairs, and each of these pairs is called a capability. This implementation has a lot of obvious advantages.

The implementation that stores the matrix by columns uses what is known as an *access list*. This implementation is common in file systems where one specifies for each file what access attributes each user has.

Both of these implementations has advantages and disadvantages and we frequently find a hybrid scheme in which the access list implementation is used for files but there is a special operation for opening a file that converts the access list implementation to a capability list implementation.