Dynamic protection structures

by B. W. LAMPSON

Berkeley Computer Corporation
Berkeley, California

INTRODUCTION

A very general problem which pervades the entire field of operating system design is the construction of protection mechanisms. These come in many different forms, ranging from hardware which prevents the execution of input/output instructions by user programs, to password schemes for identifying customers when they log onto a time-sharing system. This paper deals with one aspect of the subject, which might be called the meta-theory of protection systems: how can the information which specifies protection and authorizes access, itself be protected and manipulated. Thus, for example, a memory protection system decides whether a program P is allowed to store into location T. We are concerned with how P obtains this permission and how he passes it on to other programs.

In order to lend immediacy to the discussion, it will be helpful to have some examples. To provide some background for the examples, we imagine a computation C running on a general multi-access system M. The computation responds to inputs from a terminal or a card reader. Some of these look like commands: to compile file A, load B and print the output double-spaced. Others may be program statements or data. As C goes about its business, it executes a large number of different programs and requires at various times a large number of different kinds of access to the resources of the system and to the various objects which exist in it. It is necessary to have some way of knowing at each instant what privileges the computation has, and of establishing and changing these privileges in a flexible way. We will establish a fairly general conceptual framework for this situation, and consider the details of implementation in a specific system.

Part of this framework is common to most modern operating systems; we will summarize it briefly. A program running on the system M exists in an environment created by M, just as does a program running in supervisor state on a machine unequipped with software. In the latter case the environment is simply the available memory and the available complement of machine instructions and input/output commands; since these appear in just the form provided by the hardware designers, we call this environment the *bare machine*. By contrast, the environment created by M for a program is called a *virtual* or *user machine*.[6] It normally has less memory, differently organized, and an instruction set in which the input/output at least has been greatly changed. Besides the machine registers and memory, a user machine provides a set of *objects* which can be manipulated by the program. The instructions for manipulating objects are probably implemented in software, but this is of no concern to the user machine program, which is generally not able to tell how a given feature is implemented.

The basic object which executes programs is called a task or *process*;[5] it corresponds to one copy of the user machine. What we are primarily concerned with in this paper is the management of the objects which a process has access to: how are they identified, passed around, created, destroyed, used and shared.

Beyond this point, three ideas are fundamental to the framework being developed:

1. Objects are named by *capabilities*,[3] which are names that are protected by the system in the

sense that programs can move them around but not change them or create them in an arbitrary way. As a consequence, possession of a capability can be taken as prima facie proof of the right to access the object it names.

2. A new kind of object called a *domain* is used to group capabilities. At any time a process is executing in some domain and hence can exercise the capabilities which belong to the domain. When control passes from one domain to another (in a suitably restricted fashion) the capabilities of the process will change.
3. Capabilities are usually obtained by presenting domains which possess them with suitable authorization, in the form of a special kind of capability called an *access key*. Since a domain can possess capabilities, including access keys, it can carry its own identification.

A key property of this framework is that it does not distinguish any particular part of the computation. In other words, a program running in one domain can execute, expand the computation, access files and in general exercise its capabilities without regard to who created it or how far down in any hierarchy it is. Thus, for example, a user program running under a debugging system is quite free to create another incarnation of the debugging system underneath him, which may in turn create another user program which is not aware in any way of its position in the scheme of things. In particular, it is possible to reset things to a standard state in one domain without disrupting higher ones.

The reason for placing so much weight on this property is two-fold. First of all, it provides a guarantee that programs can be glued together to make larger programs without elaborate pre-arrangements about the nature of the common environment. Large systems with active user communities quickly build up sizable collections of valuable routines. The large ones in the collections, such as compilers, often prove useful as sub-routines of other programs. Thus, to implement language X it may be convenient to translate it into language Y, for which a compiler already exists. The X implementor is probably unaware that Y's implementation involves a further call on an assembler. If the basic system organization does not allow an arbitrarily complex structure to be built up from any point, this kind of operation will not be feasible.

The second reason for concern about extendibility is that it allows deficiencies in the design of the system to be made up without changes in the basic system itself, simply by interposing another layer between the basic system and the user. This is especially important when we realize that different people may have different ideas about the nature of a deficiency.

We now have outlined the main ideas of the paper. The remainder of the discussion is devoted to filling them out with examples and explanations. The entire scheme has been developed as part of the operating system for the Berkeley Computer Corporation Model I. Since many details and specific mechanisms are dependent on the characteristics of the surrounding system and underlying hardware, we digress briefly at this point to describe them.

Environment

The BCC Model I is an integrated hardware and software system designed to support a large number (up to 500) of time-sharing users. This system consists of two central processors, several small processors, a large central (core and integrated circuit) memory, and rotating magnetic memory. The latter contains more than 500×10^6 bytes, including approximately 12×10^6 bytes of drum having a transfer rate of more than 5×10^6 bytes per second.

The hardware allows each process more than 512k bytes of virtual memory. The central processors can accommodate operands of various sizes including 48- and 96-bit floating point numbers. The addressing structure allows characters, part-word fields and array elements to be referenced directly. The subroutine-calling instruction passes parameters and allocates stack space automatically. System calls are handled exactly like ordinary function calls; when arrays or labels are passed to the system they are checked automatically by the hardware so that they can be used by the system without further ado.

The memory management system organizes memory into *pages*. A page is identified by a 48-bit unique name which is guaranteed different for each page ever created in the system. Tables are maintained in the central memory which allow the page to be found in the various levels of the memory system. These tables are automatically accessed by the address mapping hardware the first time the page is referenced after the processor starts to run a new process. Thereafter its real core address is kept in fast registers. It is therefore unnecessary for any program other than a small part of the basic system to be concerned about the location of a page in the memory system; when it is referenced, it will be brought into the central memory if it is not already there. Extensive facilities are provided, however, to allow a process to control the level in the memory hierarchy of the pages it is interested in. The work of managing the memory is done by a processor with

read-only program memory and data access to the central memory; this processor has a 100 ns cycle time, so that it can handle the large amount of computing required to keep up with demands placed on the memory system. Another small processor handles the remote terminals, which are multiplexed in groups of 20 to 100 at remote concentrators and brought into the system over high-speed lines.

Pages are grouped into *files*, which are treated as randomly addressable sequences of pages. The only mechanism provided to access the data in a file is to put a page of the file into the virtual memory of a process. Files and processes are named and have protection information associated with them.

Domains in action

Before plunging into a detailed analysis of capabilities and domains, we will look at some of the practical situations which these facilities are designed to serve. They all have the same general character: several programs with different privileges exist. Each program corresponds to one domain. Some of the domains control others, in the sense that the capabilities of a controlled domain are a subset of those of its controlling domain. As a first example, consider the command process CP of an operating system. This program accepts a command, perhaps from a remote terminal, and attempts to recognize it as a call on a program X which CP knows about. If it succeeds, CP calls on X for execution, passing it any parameters which were included in the command. To do this, CP must set up a suitable environment for X to function in. In particular, enough memory must be provided for X to run, X must be loaded properly, and suitable input/output must be available. When X is finished, it will return and CP can process a new command.

The key point is that we want CP to be protected from X, to ensure that the user's commands continue to be processed even if X has bugs. In particular, we want to be sure that

1. X does not destroy CP's memory or files, so that CP can continue to run when X returns.
2. CP can stop X if it goes wild. Usually we want the ability to set a time limit and also to intervene from the terminal.

In other words, we want CP and X to run in separate domains, as illustrated in Figure 1 (since this is an informal discussion, we do not trouble to distinguish carefully between the program X and the domain in which it runs). Here we have shown the call from CP

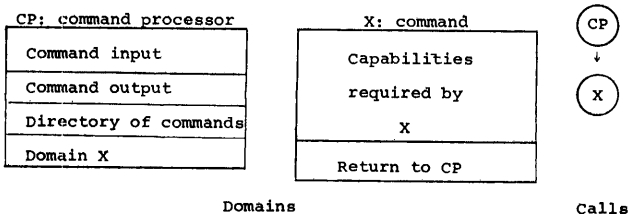


Figure 1—A command processor and its command

to X in two forms: in the picture on the right, and as a return capability in X. The reason for the capability is that X cannot return with a simple branch operation, since it would then be able to start CP running at any point, which would destroy the protection.

Suppose now that we want to allow X to get additional commands executed. X might, for example, be a Fortran compiler whose output must be passed through an assembler. A simple way to do this is to put the assembler input on a file called, say, FORTRANTEMP, and issue the command.

ASSEMBLE FORTRANTEMP, BINARY

This command is just a string, which can easily be constructed by the compiler X. To get it executed, however, X must be able to call CP. This situation is illustrated in Figure 2; note the call capability in X, which is quite different from the return capability. We are ignoring for the moment the question of how CP knows that X is authorized to call the assembler.

If the idea of the preceding paragraph is pursued, it suggests the value of being able to switch the source of command input and the destination of command output in a flexible way. By these terms we mean the

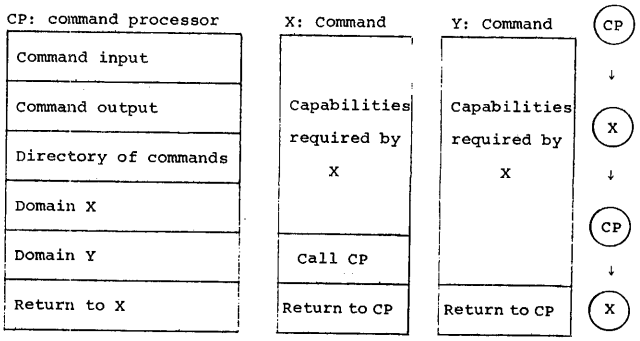


Figure 2—A recursive command processor

traffic between a program and the entity by which it is directed. In a time-sharing system this is normally a terminal at which the user is sitting; in a non-interactive system it will be a file of control cards. It is often desirable, however, to switch between the two, so that routine processing can be done automatically when the user's attention is elsewhere, yet he can regain control when things go awry. Again, it is not uncommon to wish to capture a complete record of a conversation between user and machine for later analysis and replay. More radical, it may be of interest to replace the user at his terminal with a program which can manipulate the strings of characters which constitute commands and responses. In this way major changes in the external appearance of a system can be obtained with little effort.

All of these things can be accomplished by giving interactions with the command I/O device the form of calls to a different domain which acts as a switch. A generalization to include the possibility of different command devices for different domains is easy. Thus, a user may initiate a program in a domain X which, while continuing to communicate with him, starts a subsidiary domain and feeds it commands. The subsidiary, unaware of the way in which it is being driven, may iterate the process by creating Z. The key fact which makes it all work is the isolation of one domain from others. Thus, Y may decide to close all its files without disturbing X, since Y has no way of even knowing about X's files, much less accessing them. Z, on the other hand, can be an open book to Y. Various aspects of the situation are illustrated in Figure 3.

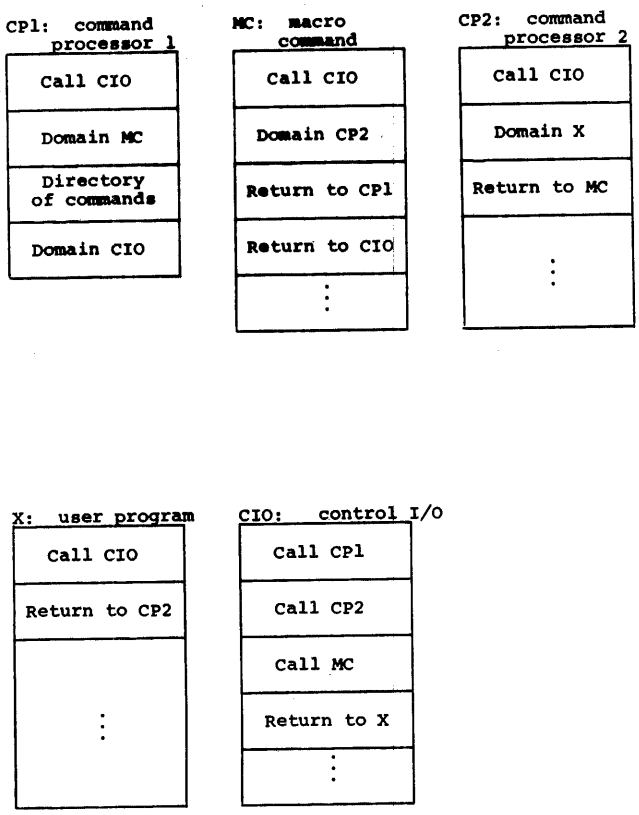


Figure 3a—Switchable control I/O—the domains

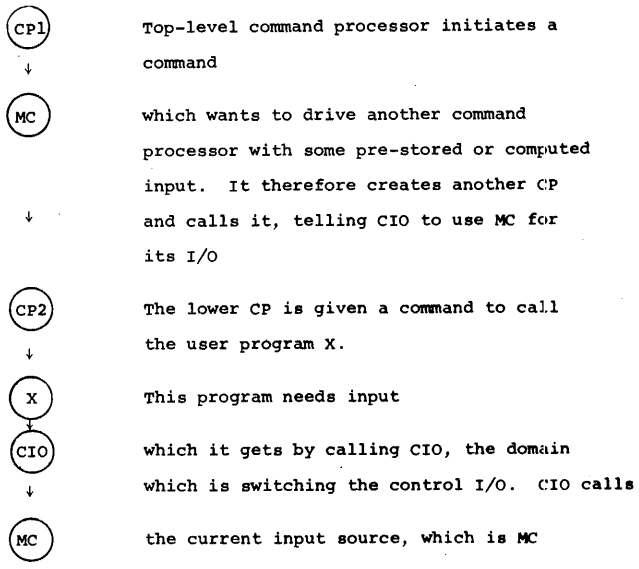


Figure 3b—Switchable control I/O—the calls

This section concludes by analyzing a problem of great practical importance: how to construct a debugging system. This example is a good source of insights into the facilities required of a protection system because of the great variety of things which can be expected to go wrong during debugging. There are two domains, one for the debugger D and one for the program X being debugged. We of course want D to be protected from X. Equally important, we want X to be completely open to D, so that every object accessible to X is also accessible to D, and furthermore that D can find all the objects accessible to X as well as access them. Otherwise D will not be able to find out what X has done or to undo any damage. Furthermore, we want D to be able to imitate any actions which X can take, so that D can create suitable initial conditions for debugging parts of X. Thus, D needs operations which, given a capability for X, allow D to

find all the capabilities in X
copy capabilities between D and X
destroy capabilities in X
enter X at any point with any machine state

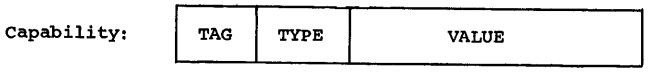
With these powers, D can also handle domains which X has created, since it can get hold of X's capabilities for them. Breakpoints can be inserted in X in the form of calls on D.

Domains and capabilities

The nature of capabilities

As we have already said, a capability is a protected name of an object. When any object is created, a capability is created to name it; without the capability the object might as well not exist, since there is no way to talk about it. The capability may be thought of as an ordinary data item enclosed in a box which prevents tampering with the contents. Thus, for example, it may be convenient to make a capability for a file consist of simply the disc address of its index. This is entirely satisfactory, since programs which handle the capability cannot modify it. If they could, disaster would ensue, since any program could put any desired disc address into a file capability, and there would be no protection at all. If the machine hardware allows a word to be tagged so that it cannot be modified except by the supervisor, then we have precisely what we want for a capability. The situation is illustrated in Figure 4. It should be possible to load and store such a word (including the tag bits) in order to give programs the necessary freedom to manipulate the names of the objects they are working with.

If this kind of hardware is not available a different and potentially confusing implementation is required. The potential can be kept from realization by referring back to the "pure" implementation of the last paragraph. What is required is to hide the capabilities away in the supervisor and provide programs with unprotected names which can be used to refer to them. When a program running in domain D presents one of these names, it is necessary to check that it actually names a capability which belongs to D. This can easily



TAG = read-only, except to supervisor

TYPE = FILE

VALUE = disk address of index

Figure 4—Structure of a capability

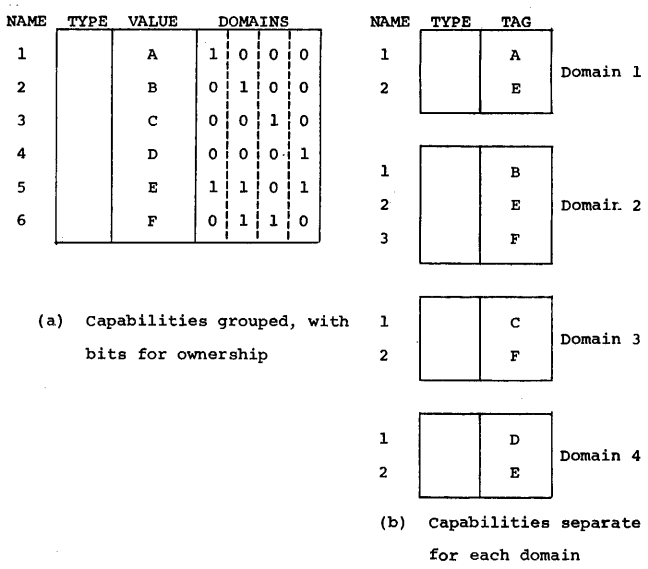


Figure 5—Capabilities and unprotected names

be done, if there are n such capabilities, by using numbers between 1 and n for the names.[3] An attractive alternative, if domains can be grouped into larger units which share many capabilities, is to number the domains from 1 to i and the entire collection of capabilities from 1 to n and to attach a string of i bits to each capability. Bit d is on exactly when the capability belongs to domain d. Figure 5 illustrates.

A somewhat more expensive implementation is to search a table associated with the domain whenever an unprotected name is used. This scheme shares with the bit-string idea the advantage that it is easy for different domains to use the same names for the same object.

There are capabilities for all the different kinds of objects in the system. On the Model I these are

files
pages of memory
processes
domains
interrupt calls
terminals
access keys

Domains and memory

The nature of a domain is considerably more dependent on the underlying system than is the case for capabilities, mainly because of the treatment of memory. From a purist's viewpoint, every access to a

memory word is an exercise of a capability for that word. A more moderate position, and one which is quite feasible on suitable hardware, is to view each access as the exercise of a capability for a *segment* which contains the word.[2] The mapping hardware which implements segmentation is thus viewed as part of the capability system, and a satisfying unity of outlook is gained. Since a segment is identified by number, the preceding section applies. We shall not consider the formidable difficulties which arise if different domains use different names for the same segment.

If segments are accessed through capabilities like everything else, then a domain consists of nothing more than a collection of capabilities. On machines not equipped with the proper hardware a domain has an *address space* as well. In the Model I this is a list of the pages which occupy each of the 64 slots for pages in the 128k memory which is accessible to a user program.

It is also necessary to deal with the fact that the hardware does not allow one domain to access the address space of another one directly. This fact is of great importance when we consider how data is passed back and forth between domains, since it implies that arrays cannot be passed simply by specifying their addresses. It is therefore extremely convenient to include as part of a call the ability to pass scalar data items, and essential to include the ability to pass capabilities. From this foundation arbitrarily complex communication can be built, since capabilities for pages, files and domains can be passed. Thus, if an array needs to be passed as a parameter, it is sufficient to pass capabilities for the pages or file containing the array, together with its base address and length. The called domain can then put the pages into its address space and access the array. This is of course much less convenient than passing an entire segment as a parameter, but it is quite workable.

An alternative approach is to organize the hardware so that the address space of one domain is a subset to that of another. This eliminates all problems when the smaller one calls the larger, although it does not help at all when we want to share only part of the address space. A subset organization fits well with a linear or "ring"-like system[4] in which the domains are numbered, and the capabilities of domain i are a subset of those of domain i-1. As we shall see, there are good reasons for wanting a more flexible scheme, but for a great many applications a linear ordering is quite satisfactory. To allow these to be handled more efficiently, the Model I hardware breaks the address space of a process into three rings:

monitor
utility
user

in decreasing order of strength. The hardware enforces a restriction that addressing cannot go into a higher ring. It also provides protected entry points into the utility and monitor rings and automatically checks addresses passed into these rings as parameters to ensure that they are legal in the ring from which they came.

This simple hardware-implemented structure permits three domains to transfer control around among each other and to address each other's memory in a very convenient and efficient way. The price paid is a rigidity in structure, and a drastic incompatibility with the main, software-implemented domain mechanism. The incompatibility is resolved by requiring a change in ring to be reported to the software, except when the only processing to be performed before returning the original ring can be done with the capabilities of the original ring. Short calls thus remain cheap, while the overhead added to longer ones is not excessive.

Domains and processes

The relationship between domains and processes is another area greatly influenced by the surrounding system. The logical nature of the two kinds of object allows a great deal of freedom: in fact, a domain has much the same appearance to a process that a segment of memory does. The storage for capabilities provided by a domain can accommodate many processes, and a single process can switch from one domain to another (subject to restrictions which are considered in the next section).

In the Model I, however, storage is allocated in 2k pages, and one of these, called the *context block*, is used to hold the system-maintained private data for each process. The cost of having a process is thus high, and there is considerable incentive to minimize the number of processes; usually one is enough per computation, if advantage is taken of the interrupt facilities described later. When the usage of space in the context block is analyzed, it turns out that there are only two items which would have to be duplicated to allow several processes to run with the same address space. These are a 14-word machine state and a stack used for local storage when the supervisor is executing in the process. This stack has a minimum of about 60 words and can grow to several hundred words at certain points during supervisor execution. It is therefore the

main barrier to the existence of cheap processes. The problem can be greatly alleviated by allocating stack space dynamically at each function call and releasing it at each return, but this would require some major changes in system organization.

Although processes are expensive, domains are quite cheap, since the bit-string method is used to assign capabilities to domains. Each process in the Model I can have about a dozen domains associated with it. The process can run in any of its associated domains but in no others. This implies that two processes never run in the same domain.

In a system in which processes are cheap, it is possible to take an entirely different approach which encourages the creation of processes for every purpose. In such a system, parallel processing is of course greatly facilitated. In addition, free creation of processes can be used to give a somewhat different form to many of the facilities described in this paper.[3]

It is perhaps worthwhile to point out that a machine whose addressing is not organized around a stack or base registers cannot reasonably run several processes out of the same domain unless they are executing totally disjoint code, because of the problem of address conflicts.

Transfers of control

Calls

The only reason for creating a domain is to establish an environment in which a process may execute with different protection than that provided by any existing domain. If this objective is to be fulfilled, transfers of control between domains must be handled with great care, since they generally imply the acquisition of new capabilities. If it is possible for a process running in domain X to suddenly jump into domain Y and continue execution at any arbitrary point, X can certainly induce Y to damage the objects accessible through Y's capabilities.

To provide an adequate mechanism for transfers between domains, we introduce the idea of a protected entry point or *gate*, and make the rule that transfer into a domain is normally allowed only at a gate. A gate is a new kind of capability which can be created by anyone with a capability for the domain. It specifies a location to which control is to go when the gate is used. Gates can be passed around freely like other capabilities, and each one may be viewed as conferring a certain amount of power, namely the power to accomplish whatever the routine entered by the gate is designed to do. With gates it is possible to selectively distribute the powers of a domain in a flexible way.

A transfer through a gate usually takes the form of a subroutine call; some provision must therefore be made for a return. It is not satisfactory to create another gate which the called process may return through, since he might save it away and use it to return at some later and unexpected time. Instead, the domain and location to return to are saved on a *call stack* in the supervisor, from which the return operation can retrieve them. It is possible to call a domain recursively with this mechanism, a feature which is generally desirable and also quite important for the trap and interrupt system about to be described.

In order to allow the stack to be reset in case of an error, or for any of the other reasons which prompt programmers to reset stacks, a jump-return (n) operation is provided which returns to the domain n levels back. Protection is maintained by requiring the domain doing the jump-return to have capabilities for all the domains being jumped over.

Traps

A *trap* is caused by the occurrence of some unusual event in the execution of the program which requires special handling, such as a floating point overflow, a memory protection violation or an end of file. When a trap occurs, it forces control to go to a specified place, where presumably a routine has been put to deal with the event. Whether any particular event causes a trap or simply sets a flag which can be tested by the program is a decision which should be under the programmer's control. Traps may be initiated by hardware (e.g., floating overflow) or may be artifacts of the software; as with most distinctions between hardware and software implementation, this one is of little importance, and we expect all traps to be transmitted to the program in the same form, regardless of their origin.

These are all obvious points which are generally accepted, and have even become embedded in the definition of PL/I. What concerns us here is the relationship between traps and domains, which is not quite so obvious. The basic problem is that the response to a trap must be made to depend on the environment in which is occurs. The occurrence of, say, a floating overflow is simply a fact, and has nothing to do with who is running. The action to be taken, on the other hand, is entirely a function of the situation. Consider the example in Figure 6. If a floating overflow occurs with the call stack in state (b), it is clear that

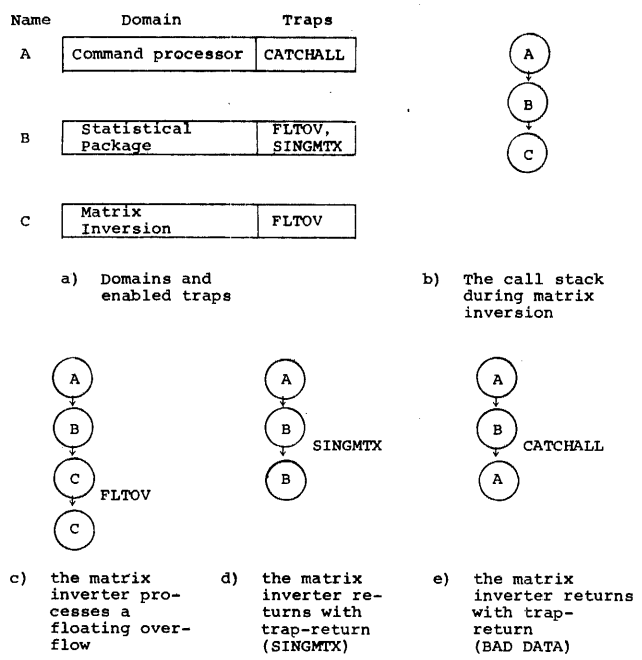


Figure 6—Traps and trapreturns

C should have the first chance to handle the trap. If it is not interested, the domain B which called it should have the second chance. In state (c), on the other hand, domain B should have the first chance, and then A. The reasons for this is that we do not wish to give up control to a weaker domain when a trap occurs.

The idea is then the following: Each domain is considered to have a *father*. When a trap occurs, it is first directed to the domain S which is running. If S does not have the trap enabled, the father of S is tried in the same way. If no one can be found to handle the trap, there are two possibilities:

ignore it;
generate a catchall trap which any domain that lacks a father is forced to handle.

If a domain T is found with the trap enabled, it is called with the name of the trap as argument. It can then return and allow execution to proceed if it is able to clear things up. Alternatively, it can do a jump-return to someone farther back on the call stack if it finds the situation to be hopeless. An important property of this scheme is that the trap routine can do arbitrarily complex processing without disturbing the situation at the time of the trap.

Conceptually, we wish to think of traps as identified by symbolic names. Each domain must then include a list of names of the traps it has enabled. Corresponding to each hardware-generated trap is a standard name. Software-generated traps can use any names, including the ones for hardware traps. This makes it easy for a subroutine to simulate the occurrence of a hardware condition which it may not be convenient to produce.

A simple extension of the return operation to a *trap-return* allows a routine to signal an error without leaving any traces of itself; the trap-return does a return and immediately causes the specified trap, without allowing any execution beyond the return point. The domain which handles the trap then sees it as having occurred in the calling routine, which is exactly what is wanted. Thus in Figure 6 we have a matrix inversion routine which processes its own floating overflows, but reflects two other conditions to its caller with trap-return. Another useful convention is to disable the trap when it occurs. This makes it much less likely that the program will get into a loop, especially for such traps as illegal instruction and memory protection violation.

Interrupts

There remains one more way to cause a transfer between domains: the occurrence of an *interrupt*. This is not intended to be the normal mechanism for communication between cooperating processes; the basic block and wake-up mechanisms[5] are expected to perform that function. There are times, however, when it is desirable to force a process to do something, even if it is not paying attention. Two obvious reasons for this are:

a *quit* signal from the terminal, which indicates that the user wants to regain control over a process which has gone into a loop, or perhaps simply become unnecessarily wordy;

the elapse of a certain amount of time, which has much the same meaning.

The action required in these two cases is different. When a timer interrupt is requested (and there may be two kinds, for real time and CPU time) the desired action is usually to call a specific domain, often the one which is setting the timer. If another domain wants a timer, it will use one which is logically different. The user's quit signal, on the other hand, is context dependent like a trap; the desired action is a function of the routine which is running when the signal arrives. Thus an iterative root-finder may interpret a quit as an indication that the solution is accurate enough, but the debugging system under which it may be run-

ning will curtail its printing when it sees a quit and await a new command. This analysis suggests a simple implementation: convert the quit into a trap from the currently executing domain. Each interrupt, then, will give rise to a call or a trap, depending on its type as declared by the programmer.

Even when we see how to convert them into operations within the process, interrupts still present one serious problem which does not arise in the handling of traps. This is the fact that a program occasionally needs to be allowed to compute for a while without losing control. Usually this happens when modifications are being made to a data base; if a quit signal should appear or a timer run out halfway through this operation, the data is left in a peculiar state. The obvious solution is to allow a process to become non-interruptible for a limited period of time. The function of the limit is to prevent the process from getting into a state from which it cannot be retrieved; exceeding it is a programming error and always causes the process to become interruptible again and an error trap to occur, regardless of whether an interrupt is actually pending. The limit is properly measured in real time, since its primary purpose is to put a bound on the frustration of the user at his console.

Non-interruptibility is a process-wide condition. It must be possible, however, for a newly-called domain to extend the limit exactly once, so that it can function properly even though its caller is about to exceed his limit. The limit is thus part of a call stack entry. When a return occurs, the old limit comes back into force, and an immediate trap may occur if it has been exceeded.

Table I summarizes the operations connected with transfers of control between domains.

TABLE I—Operations for transfers

Operation	*Arguments*
Call	Gate, Parameters
Return	Parameters
Jump	Gate, Parameters
Jump-return	Depth, Parameters
Trap	Trap number
Trap-return	Trap number

Proprietary programs

The remainder of this paper deals with the protection problems introduced when objects are allowed to have external, mnemonic names. The examples in this section are intended to introduce this subject, and are also of interest in their own right. Suppose then that a user U has a program executing in domain P and wishes to perform a circuit analysis. P has generated the input data for the analysis, and intends to use the results for further calculation. Within the system M on which P is running, some user V has written a suitable analysis program A which he has offered for sale, and U has decided to use V's program. It happens that U and V are competitors.

Both users in this situation have selfish interests to protect. First, and most obvious, V does not want his program stolen. He therefore insists that while it is executing U must not be allowed to read it. Equally important, however, is the fact that U does not want V's program to be able to read the calling program P and its data; although U may not be trying to market P, it, and especially its data, contain valuable information about U's current development work which must be kept from competitors. The relationship between U and V, and between their programs P and A, is therefore one of mutual suspicion. Each is willing to entrust the other with just enough information to allow the circuit analysis to be completed, and no more. The system must support this requirement if it is to be a suitable vehicle for selling programs.

Furthermore, care must be taken beyond the programs. While P is running it needs the ability to access U's files by name, to read input data and record results. This privilege must certainly not be extended to A, since it can learn even more about U's secrets by examining his files than by looking at his program, not to mention the possibility of modifying them. On the other hand, A may need access to V's files to obtain data for the analysis and to collect statistics and accounting information; this access must not be available to P. The protection mechanisms must therefore provide for isolating P and A at the level of file naming as well as on the lower levels which have been the subject of this paper so far.

What is required then is a system facility something like this. V establishes A as a *proprietary program*, specifying the file on which it resides. Another user's program P may then ask the system to *attach* this file. To do this, the system creates a new domain A, installs the program in it, provides it with some storage, and returns to P a gate into A. When P wants to call A, he uses the gate and passes whatever parameters he thinks are needed for A to function. When A is finished, he returns. The protection mechanisms we

have been discussing prevent undesired interference between P and A. Safeguards for the files are discussed below.

The example above is one of a great variety of similar situations. The system itself creates many of them. A LOGOUT command, for example, requires special access to accounting files and to capabilities for destroying a process, but it would be nice to call it with the standard command processor. Similarly, driving a special peripheral like a printer requires special capabilities. If a company maintains a large data base, it may wish to give different classes of users access to different parts of it by allowing them to call different accessing programs. These and many other applications fall within the general outline established by our proprietary program example. We now proceed to consider how to handle the file naming problems it presents.

External names

Table II lists the goals of a naming system for objects, and indicates some of the distinctions between the use of capabilities in names which have been discussed in previous sections, and the use of *external names*, which are strings of characters such as 'FILE1' or 'CIRCUIT'. In summary, it says that capabilities are very convenient for use by a program, since they are cheap and self-validating. On the other hand, they are very bad for people, since they cannot be typed in or remembered. Names for people should also have the property that the same name can refer to many different objects, the distinctions to be made by context. Thus, Smith's file 'ALPHA' is not the same as Jones' 'ALPHA'.

TABLE II— Goals of a naming system for objects

Goal	Achieved by Capabilities	Achieved by external names
Names are mnemonic		X
Names can be relative to other names		X
Names can be used externally		X
Possession of name authorizes access	X	
Names are cheap to use	X	
Names can be manipulated by programs	X	X

Techniques for achieving all these goals are well known. They depend on the introduction of a new kind of object called a *directory*, which consists of pairs: <external name, capability>, and an operation of *opening* an object by supplying the name to obtain the capability. Since the external name is interpreted relative to a directory, there is a suitable basis for establishing the context of a name. A tree-structured naming system is implicit in the scheme, because directories are themselves objects accessed by capabilities. It is now easy to see how a program in a domain D accesses the objects belonging to owner U. When D is created, it is supplied with a capability for U's directory, which it simply exercises.

There is more controversy over the proper methods of accessing objects belonging to other users. A popular approach is to use passwords: a public read-only directory is filled with capabilities for all other directories which allow the objects in them to be accessed provided a correct password (usually different for each object) is supplied as part of the opening operation. This method is not satisfactory. First, it is inconvenient, since it requires the person accessing the file to remember the password. Second, it is insecure. If he writes the password down, or includes it in a program, the possibility increases that it will become known. It is bad enough to have to use a password to obtain entry to the system, but at least only one password is involved, it is used only once per session, and it can be changed, if need be after each session, without too much fuss. None of these things is true of passwords attached to files: there are many of them, many people need to know them, and one must be used each time a file is opened. This scheme has no advantage except economy of implementation.

A method based entirely on capabilities suffers only one of these drawbacks: it is inconvenient, but secure. It is also, however, quite complex. The idea is that if a file (or anything else) is to be shared, a capability for it should be passed from its owner to those who wish to share it. The problem is that a capability, being a protected object, must be passed through protected channels; it cannot be sent in a letter, even a registered letter. The solution is illustrated in Figure 7. Every user has (at least) two directories, a private one which he works with, and a *transfer directory*. The public directory PUB, for which every user has a read capability, contains write capabilities for all the transfer directories. The object is to move the capability for X from PDA to PDB. Proceed as follows:

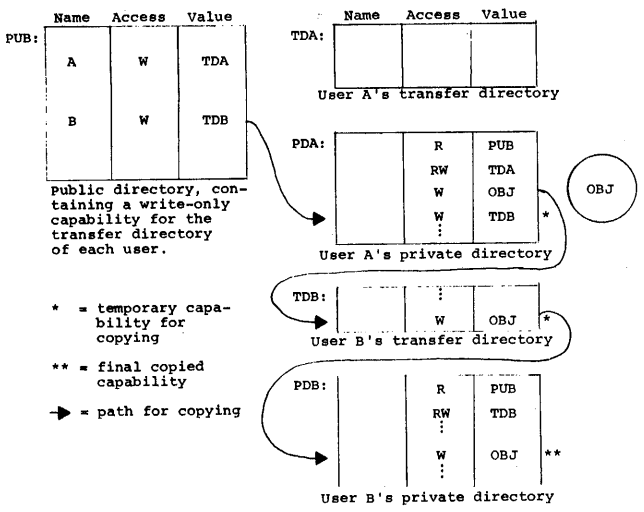


Figure 7—Sharing capabilities without access keys

A moves a capability for TDB into PDA
Using it, A moves his capability for X to TDB
B moves the capability for X from TDB to PDB

Since only B can access TDB, security is preserved. A malicious user can confuse things by writing random capabilities into the TDs, but it is easy for B to check that he has gotten the right thing. Furthermore, if X is a directory, future communication can be carried out quite conveniently, since A and B can then communicate through X without any worries about outside interference.

A much better method is based on the simple idea of attaching to a directory entry a list of the users who are allowed to access it; with each user we can also specify options, so that Rosenkrantz may be granted write access to the file while Guildenstern can only read it. This scheme, which was first used in CTSS,[1] has two drawbacks. The first is that if the list of users who are authorized to access a file is long, it takes a lot of space to store it; this problem is especially annoying if there are several files to be accessed by the same group of users. The second drawback is that there is no provision for giving different kinds of access to different domains of a computation. Both difficulties can be overcome in a rather straightforward manner.

Before we pursue this point, it is important to notice why the difficulty encountered above in the capability-passing scheme does not arise here. We can think of the computation of a logged-in user as possessing a special kind of capability which identifies it as belonging to him. If SMITH is the user, we will refer to this capability as SMITH*, meaning that the string

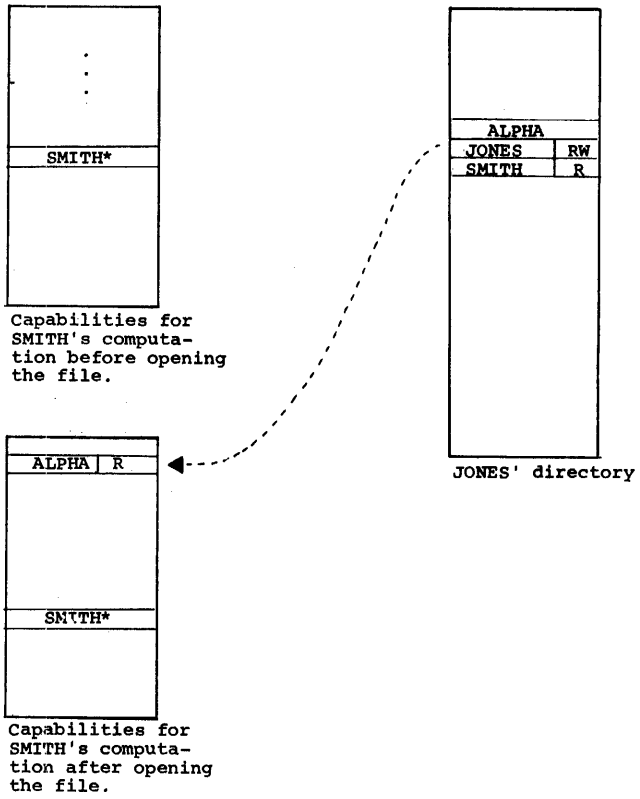


Figure 8—Use of access keys

'SMITH' has been enclosed in a tamper-proof box. When JONES wishes to give SMITH access to his file ALPHA, he puts the name SMITH on the access list; JONES can do this since he has a capability for ALPHA. When a computation presents the capability SMITH*, the system observes that the string (or user number) which is the contents of the capability matches the string on the access list and grants the access. At no time is it necessary for JONES to have SMITH* in his possession. He needs only the name SMITH which, since it is not a protected object, can be communicated to him by shouting across the room. Figure 8 illustrates.

To generalize the method we need two ideas. One is that of an *access key*. This is an object (i.e., it can be referenced only by using a capability) which consists simply of a bit string of modest length, long enough that the number of different access keys is larger than the number of microseconds the system will be in existence. Any user may ask the system for a new access key; the system will create one never seen before and return a capability for it. The object SMITH*

mentioned in the last paragraph is an example of an access key; one is kept for each user in the system. Since an access key is an object, capabilities for it appear in the directories and are protected exactly as is done for any other object (since the access key is a small object, it may be convenient for the implementation not to give it any existence independently of the capabilities for it, i.e., to make the value of the capability the object itself, rather than a pointer to it as in the case of files). To give a group of users access to some files, all we have to do is distribute a new access key GROUP* to the users and put GROUP on the access list for each file. The distribution is accomplished by creating GROUP* and putting all the users on its access list; once they have copied it into their directories they can be removed from the access list, so that no space need be wasted. In practice, as we have pointed out, numbers of perhaps 64 bits would be used instead of strings like 'GROUP'.

The second idea is not new at all. It consists of the observation that since an access key is just an object, different domains can have different access keys and hence different kinds of access to the file system. Thus, for example, a user's computation may be started with two domains, one for his program with his name as access key, and the other for system accounting with an access key which allows it to write into the billing files. With a single suitable access key, a domain can easily get hold of an arbitrarily large collection of other objects which are protected by other keys, since the first key can be used to obtain other keys from the directory system.

SUMMARY

We have described a very general scheme for distributing access to objects among the various parts of a computation in an extremely specific and flexible way. The scheme allows two domains to work together with any degree of intimacy, from complete trust to bitter mutual suspicion. It also allows a domain to exercise firm control over everything created by it or its subsidiaries.

REFERENCES

1 P A CRISMAN editor
The compatible time-sharing system: A programmer's guide
MIT Press 2nd ed Cambridge Mass 1965

2 J P DENNIS
Segmentation and the design of multi-programmed computer systems
J ACM Vol 12 Oct 1965 589

3 J B DENNIS E C VAN HORN
Programming semantics for multiprogrammed computation
CACM Vol 8 No 3 March 1966 143

4 R M GRAHAM
Protection in an information processing utility
CACM Vol 11 No 5 May 1968 368

5 B W LAMPSON
A scheduling philosophy for multi-processing systems
CACM Vol 11 No 5 May 1968 347

6 B W LAMPSON et al
A user machine in a time-sharing system
Proc IEEE Vol 54 No 12 Dec 1966