COPS - A MECHANISM FOR COMPUTER PROTECTION*

Gregory R. Andrews

TR 74-214

October 1974

Department of Computer Science
Cornell University
Ithaca, New York 14850

*This work was supported in part by the National Science Foundation (USA) through Grant GJ-36273. Author's address: Computer Science Department, Upson Hall, Ithaca, NY. 14853.

COPS - A MECHANISM FOR COMPUTER PROTECTION*

Gregory R. Andrews
Cornell University

A computer protection mechanism is a set of tools for controlling the actions of computations and safeguarding stored information. This paper describes a new mechanism, COPS, which is a kernel of data structures, primitive operations, and a monitor and is used to specify and enforce the capabilities of actors (processes and procedures). COPS can be used to implement a variety of security policies and systems and to enhance software reliability. Its tools are sufficient to solve problems in the areas of isolation, controlled sharing, restricted access, mutually suspicious interaction, and confinement.

Key Words and Phrases: protection, kernel, operating system, security, access control

CR Categories: 4.3, 6.2

*This work was supported in part by the National Science Foundation (USA) through Grant GJ-36273. Author's address: Computer Science Department, Upson Hall, Ithaca, NY. 14853.

COPS - A MECHANISM FOR COMPUTER PROECTION

Gregory R. Andrews

INTRODUCTION

Throughout the brief history of automatic digital computers, the desire to safeguard stored information and control the effects of executing programs has existed. Initially, the goal was to limit the effect of malice or error in order to preserve the integrity of operating system software. At present, the goals have expanded to include the ability to store sensitive information and keep it private, to selectively share information while controlling its use, and to permit computations to cooperate while retaining their independence and internal security. The set of tools in a computer system which enables access relations among computations, information and system resources to be specified and enforced is a protection mechanism.

Recently, the viability of "capability machines" as an architecture for implementing protection has become obvious [3,4]. The goal of the work presented here is to specify the contents of capabilities and propose a set of primitive instructions for their manipulation. The resultant protection kernel, which is independent of other operating system services, provides numerous benefits:

(1) The mechanism becomes a candidate for hardware implementation and the attendant effeciency which results.

(2) Operating system components can be designed using all of the protection facilities; this should lead to more reliable and secure systems [8].

(3) Properties about the mechanism, such as correctness [9], can be established independent of any software using its facilities.

This paper identifies numerous protection problems and proposes a specific mechanism, COPS[1], for their solution. The mechanism can be used both to implement a variety of security systems, such as tree-structured file systems, and to aid software reliability by limiting the effect of errors and enforcing program modularity. It is a synthesis and extension of previous work, notably that of Dennis and Van Horn [2], Lampson [6], Spier [11], and the HYDRA group [12].

By controlling the address mapping from logical objects to physical memory, COPS makes it possible to isolate user information from operating system software as well as other users; at the same time, no memory management policies are imposed. Information sharing is provided by a small set of primitives which manipulate the rights of computations according to a non-restrictive protection philosophy, the principle of control. This principle allows the definer (creator) of an object to either have complete control of its use, or delegate any or all of his authority; it both permits flexible system policies and preserves the security of information. Arbitrary restrictions on actions can be imposed by implicitly invoked, user-defined monitors. All computations are treated as equals so mutually suspicious subsystems can be readily implemented. Finally, it is possible to use the mechanism to design computational services which are message confined, meaning that they cannot

---

[1] Cooperation, Privacy, and Sharing

remember their input or divulge it by sending messages to other computations. COPS' properties are informally described in this paper; they are formalized in [1].

EXAMPLE - A SIMPLE FILE SYSTEM

Before describing the components of COPS in detail, we consider the protection problems involved in implementing a simple file system. Suppose the file system has the structure indicated in Figure 1 and has three main components: an external storage manager, a file manager and a file access subsystem. The external storage manager selects free storage for new files and compacts storage when necessary. The file manager is called by users to create directories and new files and to change directory entries; he interacts with the external storage manager. In order to access files, users call the file access subsystem which performs the access, transforming the data if necessary. We assume that a file directory is associated with each user to indicate the files accessible to him. As indicated in Figure 1, two users may share a file.

Three types of problems appear in this example and should be solvable by a protection mechanism; they are <u>isolation</u>, <u>controlled sharing</u> and controlled cooperation. In order to prevent the file system computations from becoming erroneously or maliciously corrupted, their contents should be isolated from, namely inaccessible to, their users. Equally important, a user may wish to use a file to store sensitive information, such as employee records or the results of classified experiments, and may therefore want the file isolated from other users and the system. Providing isolation for user information requires both that the file manager give up the ability to access a newly created file and that the external storage manager not be able to examine or alter the contents of external storage occupied by files. Isolation is made difficult in practice by the desire of the memory manager to move information about, for example to compact free storage or move data in a memory hierarchy. We call the problem of managing an object while being isolated from its contents the manager problem.

Many problems require some degree of sharing, or at least are more efficiently solved if sharing is permitted. In Figure 1, both $User_1$ and $User_2$ have access to $File_2$. If $User_1$ is the creator of $File_2$, he might wish to limit the access granted to $User_2$. For example, he might wish to allow $User_2$ to only read the file, or perhaps only read a record of the file, or even further, perhaps only read the file between 8:00 and 5:00 on the last day of the month. In general, arbitrarily complex conditions might be placed upon the type of access to or control over an object. If isolation is possible and sharing is controlled, then objects are secure in the sense that all access to them is authorized.

Computations cooperate by sharing information and by calling upon each other for services. If $File_2$ in Figure 1 is a proprietary program created by $User_1$, $User_2$ might be authorized to execute it. During execution, $File_2$'s contents should not, however, become accessible to $User_2$ or the program might not remain proprietary. In addition, $User_2$ might be suspicious of the actions of the program in $File_2$ and want to insure that the program cannot access any more information than it needs to execute. This relation between $User_1$ and $User_2$ is one of mutual suspicion [10]. In order to allow mutually suspicious computations to interact, a protection mechanism must permit computations to have disjoint rather than nested (master/slave) sets of rights to stored information.

In the file system of Figure 1, we assume that users must call upon the services of the file access subsystem in order to access files. If a user has sensitive information in a file, he most probably wants to prevent its leakage to other computations. The file access subsystem can perform its function and be completely trusted if it is confined [7]. A confined computation is one which cannot either remember its input or divulge it to any other computation except when explicitly authorized by its caller. Confinement is an extremely difficult problem because of the variety of non-obvious information channels which exist in computer systems; examples are the ratio of IO to computing and the frequency of calling a procedure. It is possible, though, to prevent the file access subsystem from remembering its input or divulging messages where a message is a value either written into a storage location accessible to another computation or passed as a parameter to another computation [1].

OVERVIEW OF COPS

COPS protects all objects in a computer system by controllin the actions of actors (processes and procedures). It, as well as any protection mechanism, can be abstractly described by a quintup $\langle O,S,M,P,E\rangle$ of components:

(1) O: The set of objects protected.

(2) S: The protection state containing object descriptors and capabilities which define authorized actions.

(3) M: A set of monitors which determine the legality of every attempted action by consulting the contents of the protection state and the environment of the executing computation.

(4) P: A set of primitive instructions used to effect changes in the protection state.

(5) E: A set of primitive instructions used to effect changes in the execution environment.

With respect to protection, a system has four classes of objects: actors, information structures, memory resources, and non-memory resources. Actors, as the name implies, are the objects who take actions and consequently are to be protected against; in COPS, the two types of actors are processes and procedures. Each actor is uniquely identified by a name.

Information structures are logical collections of information, analagous to segments, which have a unique name, a type, and a mapping indicating the memory resources in which the contents of the structure are stored. Information structures can be subdivided into control structures and computing structures. Control structures (control segments) contain the capabilities and/or execution control variables of actors; examples are process descriptors, procedure activation records, and reference structures which contain a set of capabilities and can be used to implement objects such as argument lists or file directories. Computing structures (data segments) contain the code, constants, and variables accessible to actors.

The physical objects of a system are the memory and non-memory resources. A memory resource is any readable and writable storage media and has a descriptor (analogous to a page table) specifying its name, type, owner, status, and value. The type distinguishes between different kinds of memory such as pages and sectors and the value is the locations (a contiguous set of addresses) named by the resource; the meaning of the owner and status fields will be described shortly. Non-memory resources have a name, type, and value and represent resources such as peripherals, CPU's, and channels. The types of objects and their relation to each other is summarized in Figure 2.

Besides object descriptors, the protection state of COPS contains capabilities [2,3,4,5,6,11,12] which have the abstract form:

(object specification, (access attributes, control attributes))

As mentioned, capabilities are stored in control structures associate with actors; the domain of an actor contains all his capabilities and thus defines all his authorized actions. The object specification names an object, perhaps modified to apply to only part of the object, and the attributes identify authorized actions. In particular, access attributes authorize actions on the object, such as "Read" or "Change", while the control attributes authorize actions on capabilities for the object, such as "Grant" or "Destroy". Specific attributes for the different types of objects will be define as we proceed.

The enforcer of protection in COPS is the Basic Monitor. For every action on a processor, the monitor determines if capabilities authorizing the action on all operands exist in the environment of the executing computation. The environment contains all capabilities in the domains of the executing process and the executing procedure plus capabilities for any arguments passed when the environment was activated. COPS also allows users to define their own monitors in order to implement non-standard restrictions on actions; these user monitors are invoked by the Basic Monitor.

Any system being protected by COPS is assumed to have an initial protection state containing object descriptors and capabilities for those objects loaded when the system begins operation. Subsequent changes to the protection state are caused by the eight COPS protection state primitives. These primitives fall into two categories: those which manipulate capabilities for memory objects and those which manipulate capabilities for other objects. As mentioned, attempted actions are validated with respect to the environment of execution. The final component of COPS is a set of environment change primitives such as Call a procedure and Start a process.

MEMORY MANAGEMENT

Since information in computers is stored in memory, all actions that affect memory must be controlled in order to protect its contents from unauthorized access. In COPS, every memory resource always has an owner. An actor who owns a memory resource has the power to Create information structures which map into his memory or to Allocate his ownership to another actor. An actor who has owned and allocated a memory resource becomes a manager of the resource with power to Deallocate the memory, Move the information it contains, and change the mapping (Changemap) of information structures which map into it. Ownership is not permanent but can instead be allocated as the occasion warrants.

The actions authorized on memory resources are expressed by memory capabilities. An actor who owns memory possesses a capability of the form:

(memory name, (Create, Allocate))

while a manager has a capability of the form:

(memory name, (Move, Deallocate, Changemap))

We now define the memory management primitives; Create will be discussed in the next section. Our primitives permit flexible memory management without sacrificing either the ability to isolate information or manage its mapping to memory.

The only direct access to memory allowed by COPS, that is access which is not through an information structure, is one that permits information to be moved (copied). Move($Memory_i$, $Memory_j$) moves information from $Memory_i$ to $Memory_j$ and is allowed if an actor has "Move" capability for both memory objects (i.e. he is a manager), $Memory_j$ is empty, and they both have the same owner. $Memory_j$ must be empty and $Memory_i$ and $Memory_j$ must have the same owner in order to prevent a manager who executes Move from being able to acquire access to the moved information. We assume that Move is a memory to memory type of transfer (i.e., intermediate storage, if any, is inaccessible to the caller of Move). This operation can be used to coalesce file storage, swap copies of primary storage onto a paging drum, or generate checkpoints. Move does not, however, enable any actor to examine or alter the contents of the copied storage.

Capabilities for memory objects are manipulated by the Allocate and Deallocate primitives. Allocate(Memory_i, Actor_j) is used to pass ownership of Memory_i to Actor_j, and results in the allocator becoming a manager of Memory_i. Deallocate(Memory_i, Actor_j) takes ownership of Memory_i away from Actor_j and returns it to the deallocator; in addition, Memory_i is purged to prevent a manager from discovering its contents and the mapping of any information structures to the object is broken. These primitives can be used to enable the memory management module of an operating system, for example, to give a user subsystem control over a subset of memory.

COPS has one primitive to manipulate memory descriptors. This primitive, Changemap(Memory_i, Memory_j) causes the value field of one memory object to be switched with that of another; this enables the address mapping from information structures to memory to be changed, for example by a page swap module. To execute Changemap, an actor must not only have "Changemap" capability for both Memory_i and Memory_j, but Memory_j must be empty or contain information previously Moved from Memory_i and both objects must have the same owner. As with Move, these extra conditions prevent a manager from illicitly acquiring access to information in managed memory. When used in combination with Move, Changemap can effect the protected movement of information without compromising its security.

An implementation of COPS might require other memory primitives to create, destroy and/or change the size of memory objects; they are not considered here due to their machine dependence. We do assume, however, that descriptors are changed only if the memory is empty. The memory capabilities and primitives discussed in this section are summarized in Figure 3. Their use will be illustrated shortly to solve parts of the file system example.

INFORMATION STRUCTURE MANAGEMENT

The goal of information structure control in COPS is to provide means for actors to share access to information while controlling its use. The manipulation of capabilities for information structures operates according to a new concept, the principle of control, which means that the creator of an object can completely control its use or can delegate any or all of his control to other actors. Four COPS primitives exist to manipulate access capabilities; they are Create, Grant, Ungrant, and Destroy.

Create(type, initial memory) defines a new information structure by building a descriptor for it and returning a capability authorizing full access to and control over the structure. As mentioned in the previous section, ownership of memory is the prerequisite for the creation of information structures mapping into the memory. The capability returned to the environment of the creator authorizes him to use the new object ("Read," "Change", and either "Execute", "Transfer", "Switch", or "Use") and control capabilities for it ("Grant", "Ungrant", and "Destroy"). The different types of information structures and the attributes governing their use are summarized in Figure 4. Create allows information structures to overlap in the sense that they occupy the same memory; this enables complex data structures to be built and controlled.

The actor who executes Create is initially the only actor with a capability for the newly created object. If one actor wishes to share the object with another, the Grant primitive is used. Grant(Capability, Control $Structure_i$) stores a capability in the specified control structure, which may be a process descriptor, a procedure descriptor, or a reference structure (such as a file directory). In order to Grant, an actor much have "Grant" control for the Granted capability, must Grant a subset of his attributes, and must have "Change" access to the affected control structure. Information structure capabilities contain a part field (see Figure 4) which enables one actor to limit another's access to part of any object, for example a record of a file. The part names a subset of the memory of an information structure; its implementation is dependent upon the representation of memory in a machine using COPS. For example, on a paged machine the part could be specified by a page name, displacement, and length.

In addition to limiting access to part of an object, one actor can place arbitrary restrictions on the actions of another by Granting an indirect capability, indicated in Figure 4. An indirect capability flags some of the access and control attributes and names a monitor procedure. Any action which uses an indirect capability causes a trap to the monitor procedure if the attribute authorizing the action is both present and flagged. The user-defined monitor can impose arbitrarily complex conditions on access, such as write only values between one and ten, or can perform measurements based upon access, such as traces.

Because information structure capabilities in COPS contain control attributes as well as access attributes and because Grant can be used to share any attributes, including control attributes, flexible control policies can be implemented. For example, the file manager of our file system example can give a user both access to and control over a newly created file. Using the Ungrant primitive the file manager can even take away his own capability for the file. Ungrant(Capability, Control Structure$_i$) takes a capability away from the named control structure. It can be executed by an actor if he has "Ungrant" control of an object, "Change" access to the affected control structure, and a capability containing a superset of the attributes in the Ungranted capability.

Destroy(Object$_i$) is the final primitive used to manage information structure primitives; it destroys an object's descriptor and all capabilities for it. Use of this primitive is governed by the "Destroy" control attribute. More than one actor could possibly have "Destroy" power over an object since this attribute can be Granted like any other. Destroy is potentially complex when an actor is destroyed, since the actor can have numerous capabilities.

The direct and indirect types of capabilities of COPS combined with the above four primitives enable information structures to be managed in a variety of ways. In particular, we shall use them shortly in order to solve the file system example. Numerous other examples are given in [1].

We do not specify access attributes for non-memory resources since they are machine-dependent. Examples, however, might be attributes authorizing "Input" or "Output" on devices, or "Start" and "Stop" on processors. For COPS, we assume that some attributes exist and that capabilities for non-memory resources are represented and manipulated in the same manner as those for information structures.

ENVIRONMENT MANAGEMENT

Actions in COPS are validated with respect to the environment of a computation executing on a processor. An environment consists of the capabilities of an executing process and procedure and capabilities for the arguments, if any, passed to the environment when it was activated. An environment change occurs whenever the executing process changes, for example when a process is awakened, the executing procedure changes, for example on a call, or when just the argument list changes, for example on a recursive call.

In order to cause the environment to change, we assume that an implementation of COPS contains primitives such as Start, Stop and Resume for processes and Call and Return for procedures and argument lists.[2] Start activates a process, Stop blocks one, and Resume re-activates one; they are each authorized by the "Switch" access attribute in capabilities for processes. These three primitives save or restore all three components of the execution environment (process, procedure, argument list) and may call a scheduler.

Call and Return cause the executing procedure to change and can pass parameters. COPS permits capabilities as well as values

---

[2] We are not concerned with the way the environment changes but only with how the changes affect the protection state so we are not committed to any particular set of primitives. The ones described are a representative set, however.

to be passed subject to the restriction that any capability which is passed must be Grant-able by the caller. Both Call and Return are authorized by the "Transfer" attribute in capabilities for procedures. Call is assumed to save the procedure and argument list components of the Calling environment, to build an argument list (perhaps implemented by a reference structure), to create any necessary activation record for the Called procedure, and to set control registers for the new environment. Return effectively does the reverse of Call.

The environment can also change implicitly as the result of a trap, for example a page fault or invocation of a user-monitor named in an indirect capability. A trap is considered to be similar to a call where the arguments passed to the trap handler identify the trapping procedure and the operation which caused the trap. Return from a trap handler resumes execution of the trapped environment in the same manner as a usual Return.

THE BASIC MONITOR

The remaining component is the Basic Monitor which authenticates every action taken in a system controlled by COPS. Its operation is indicated by the algorithm in Figure 5. The Basic Monitor might in general need to check many conditions, such as those already stated for Grant, in order to determine if appropriate capabilities exist. Note that the Basic Monitor might trap to a user-monitor for extra checks on any or all of the operands of an attempted action.

A SIMPLE FILE SYSTEM REVISITED

In order to illustrate COPS' utility, we now outline a solution to the file system example. Figures 6, 7, and 8 contain code skeletons for each of the components of the system of Figure 1. Figure 6 illustrates the possible actions of the External_memory_manager on both requests for storage and release of storage no longer needed. The Request_storage procedure may need to compact external storage in order to select a block of the requested amount; Move and Changemap can be used to copy the contents of a file to another block of storage and change the address mapping of the file's descriptor to point to the new block. Because COPS separates information access from storage management, External_memory_manager is not able to examine or alter information in storage allocated to files. It can, however, Move the information and Deallocate the storage, in which case the information is purged.

The File_manager of Figure 7 creates a new file by selecting storage and Creating an information structure mapping into it. The user on whose behalf the file is created acquires access to it by means of a capability stored in his directory. Since File_manager Ungrants all its own capabilities for the file, except "Destroy", it retains no ability to access the file. If File_manager does not execute any COPS' primitives other than those indicated, it has no access to the file after Return. This procedure illustrates the utility of Granting control as well as access and of Ungranting; in effect, ownership of the file passes from File_manager to the user.

The user on whose behalf a file is created initially has the only access capability for it. He can use Grant to share access with other users and, by Granting an indirect capability, can arbitrarily restrict another user's access.

In order to access a file, a user calls the File_access procedure outlined in Figure 8 and passes capabilities for a buffer and a file as parameters. If the File_access procedure never has local capabilities to "Change" any object then it cannot either remember input parameters nor write their contents into an object accessible to another actor. If, in addition, the File_access procedure does not transfer to other actors, then the user's input parameters cannot be passed as parameters to another actor. Any procedure meeting these two conditions is <u>message confined</u>[3] because it cannot send a message to another actor (proven in [1]); these conditions do not prevent covert leakages, however [7]. Note that the File_access procedure of Figure 8 is <u>not</u> prevented from doing useful work, since temporary storage areas can be passed as parameters.

Although greatly simplified, this example illustrates three important relations between a system and users which are realizable with COPS. First, the use of storage can be managed without compromising the security of information stored therein. Second, objects can be created and destroyed by one actor while their use is completely controlled by another. Third, service computations can perform useful work and yet be prevented from retaining or divulging their input by any direct means.

[3] An important feature of these conditions is that they only depend on the capabilities, not the code, of the computation.

SUMMARY OF COPS

Our protection mechanism, COPS, has a protection state containing actor capabilities and object descriptors, a Basic Monitor, and a set of primitives to change the protection state and execution environment. The components, including algorithms for each primitive, are described in detail in [1]. The mechanism is independent of operating system software and as such can be used to program protection in operating systems as well as user subsystems; the file system example is an indicative application.

The Basic Monitor enforces the restriction that an actor can only act in ways authorized by the capabilities in his environment. COPS is correct in the sense that an action is allowed if and only if there exist capabilities authorizing it and the protection state changes only as the result of a state changing primitive. User-defined monitors are implicitly invoked by the Basic Monitor whenever a capability does not expressly authorize or prohibit an action but indicates that further checks are required.

The protection state primitives enable computations to manage memory and control access to objects. The access right primitives (Create, Grant, Ungrant, and Destroy) operate according to a new philosophy, the principle of control, which is more general than the principle of ownership [11] and permits a variety of protection policies to be implemented. At the same time, the primitives prevent an actor from violating the security of any object because he is prevented from increasing his own capabilities (proven in [1]). Since each computation executes in an envelope defined by its environment, the powers of a computation can be dynamically changed without requiring changes to the executed code. The environment change primitives make it possible for computations to cooperate without compromising their internal security.

By fixing the number of object types, COPS is simpler than the HYDRA mechanism [12] in the sense of requiring less overhead. It does not, however, prevent user defined types and type checking; in fact, COPS can be used to construct a HYDRA-like mechanism (see [1]). The reference type of object is analagous to a capability segment [2,4] but does not appear in recent mechanisms. It enables arrays of capabilities to be manipulated and communicated in the same ways that arrays of variables are handled. The concept of object descriptors is a generalization of the idea of segment descriptors and/or page tables used in virtual memory systems. Since COPS protects all objects in a computer system, it includes descriptors for physical resources in addition to ones for information structures.

One of the major points of difference between COPS and other mechanisms is the inclusion of the logical to physical address mapping in the protection state and the definition of four memory management primitives. One school of thought says protection is strictly a logical relationship between actors and information structures and is consequently not concerned with physical memory. Unfortunately, isolation of information requires both protection of physical memory and control of access to information structures. COPS does not say how memory is managed but merely provides the framework within which its use is controlled. Our Allocate and Deallocate primitives were suggested by Spier [9] who first recognized the importance of both logical and physical protection. The Move and Changemap primitives are new, however, and are included to enable the manager problem to be solved.

The access attributes of capabilities are similar to those in other capability based mechanisms [2,3,4,5,6,10,11,12]. We, however, permit access to be restricted to part of an object in a direct manner. The parts field can be used to enable part of an object to be passed as a parameter without having to pass access to the entire object; Schroeder [10] first observed the need for this facility in order to handle mutually suspicious actors. The indirect capability is another important feature of COPS which permits user-defined access and control restrictions to be both explicitly associated with an object and implicitly invoked. Existing mechanisms require calling a caretaker procedure which implements an access restriction. By indirect capabilities, as opposed to caretakers, not every action need be monitored and a change in access restrictions, for example from a monitored to an unmonitored condition, merely requires a change in a capability not in the code using the capability.

Our four access management primitives, Create, Grant, Ungrant, and Destroy, are similar to ones in other mechanisms [5,6,11,12]. As with HYDRA [12], their use is governed by means of control attributes contained in capabilities. Unique, however, is that they can be manipulated just like access attributes; in particular, all attributes can be given away. In addition, the requirement that a Granter must have "Change" access to the affected object enables capability propagation to be limited and enables the implementation of message confined computations.

Finally, environment change primitives similar to ours have also appeared elsewhere. The major difference is COPS requirement that a capability can be passed as a parameter only if it is Grantable; this helps provide complete control of all access capability propagation.

IMPLEMENTATION CONSIDERATIONS

Although COPS has yet to be implemented, recent advances in the design and construction of capability addressing machines [3,4,10] lend credence to our belief that an implementation can, indeed, be effcient. A capability machine provides addressing and access control using capabilities very similar to COPS'. Each capability names an object and the access and control associated with it. Instructions executed on such a machine reference a "current capability list" (analagous to a descriptor segment) for their operands and, in parallel, perform address decoding and basic monitoring. Capability registers, in addition to the usual data registers, exist for use in manipulating capabilities. Finally, Call and Return instructions exist to change the environment of execution.

The relation of COPS to capability machines is that COPS defines the types of objects and their relationships, the types of capabilities and their attributes, the primitives used to change the protection state (capability lists), and the monitor. In essence, the capability machine provides the architecture to implement COPS. On an existing machine, such as Plessey's System/250 [4], the COPS primitives could be implemented in software as a protection kernel of procedures entered by the machine's Call/Return mechanism. If these primitives are the only ones with access to the capability registers, then all other procedures (user or operating system) would have to Call them to manipulate capabilities. Indirect rights and parts would, however, have to be implemented by interpretation since current capability machines do not provide for them.

The above approach should result in a reasonably efficient implementation since all enforcement is done in hardware. Changes to the protection state occur much less frequently in practice than other operations so the overhead incurred by software primitives should be acceptable. Another possible approach is to implement COPS on a micro-programmable machine where the capabilities, descriptors, monitor, and perhaps even the primitives are micro-coded. Regardless of the approach, however, some potentially serious problems need to be solved in order to efficiently manage the large number of object descriptors, minimize the memory references on address decoding, and prevent corruption of the protection state caused by asynchronous activity or hardware error.

CONCLUSION

This paper has presented a logical description of a mechanism, COPS, for controlling all actions in a computer system. Its components have been designed to make it possible to isolate objects, to insure the security of their contents, and to allow their use to be flexibly controlled. In addition, COPS makes it possible to implement mutually suspicious and message confined computations. These properties are defined and proven in [1]. The logical next steps are to demonstrate its feasibility by carrying out an implementation and to apply its tools to the design of secure systems.

ACKNOWLEDGMENTS

The author is most grateful to Alan C. Shaw for the numerous hours he has spent discussing and constructively criticizing the work reported in this paper. Charles G. Moore III also provided helpful suggestions.

BIBLIOGRAPHY

1. Andrews, G.R. COPS - a protection mechanism for computer systems. Technical Report 74-07-12, (Ph.D. Dissertation), Computer Science Group, University of Washington, 1974.

2. Dennis, J.B. and Van Horn, E.C. Programming semantics for multi-programmed computations. Comm. ACM 9, 3(March 1966), 143-155.

3. Fabry, R.S. Capability-based addressing. Comm. ACM 17, 7 (July 1974), 403-412.

4. Hamer-Hodges, K.J. A fault-tolerant multiprocessor design for real-time control. Computer Design 12, 12(Dec. 1973), 75-81.

5. Jones, A.K. Protection in programmed systems. Ph.D. Dissertation, Carnegie-Mellon University, 1973.

6. Lampson, B.W. On reliable and extendable operating systems. Working material, Second NATO Conference on Software Engineering, Rome, October 1969.

7. Lampson, B.W. A note on the confinement problem. Comm. ACM 16, 10(Oct. 1973), 613-615.

8. Linden, T.A., et. al. A panel session -- security kernels. AFIPS Conf. Proc. 44, 1974.

9. Popek, G.J. Access control models. Center for Research in Computing Technology Report 28-72 (Ph.D. Thesis), Harvard University, Dec. 1972.

10. Schroeder, M.D. Cooperation of mutually suspicious subsystems in a computer utility. MIT Project MAC Technical Report TR-104 (Ph.D. Dissertation), September 1972.

11. Spier, M.J. A model implementation for protective domains. Preprint of a paper in International Journal of Computer and Information Sciences, June, 1973.

12. Wulf, W.A., et. al.. Hydra: the kernel of a multiprocessor operating system. Comm. ACM 17, 6(June 1974), 337-345.

Figure 1

A Simple File System

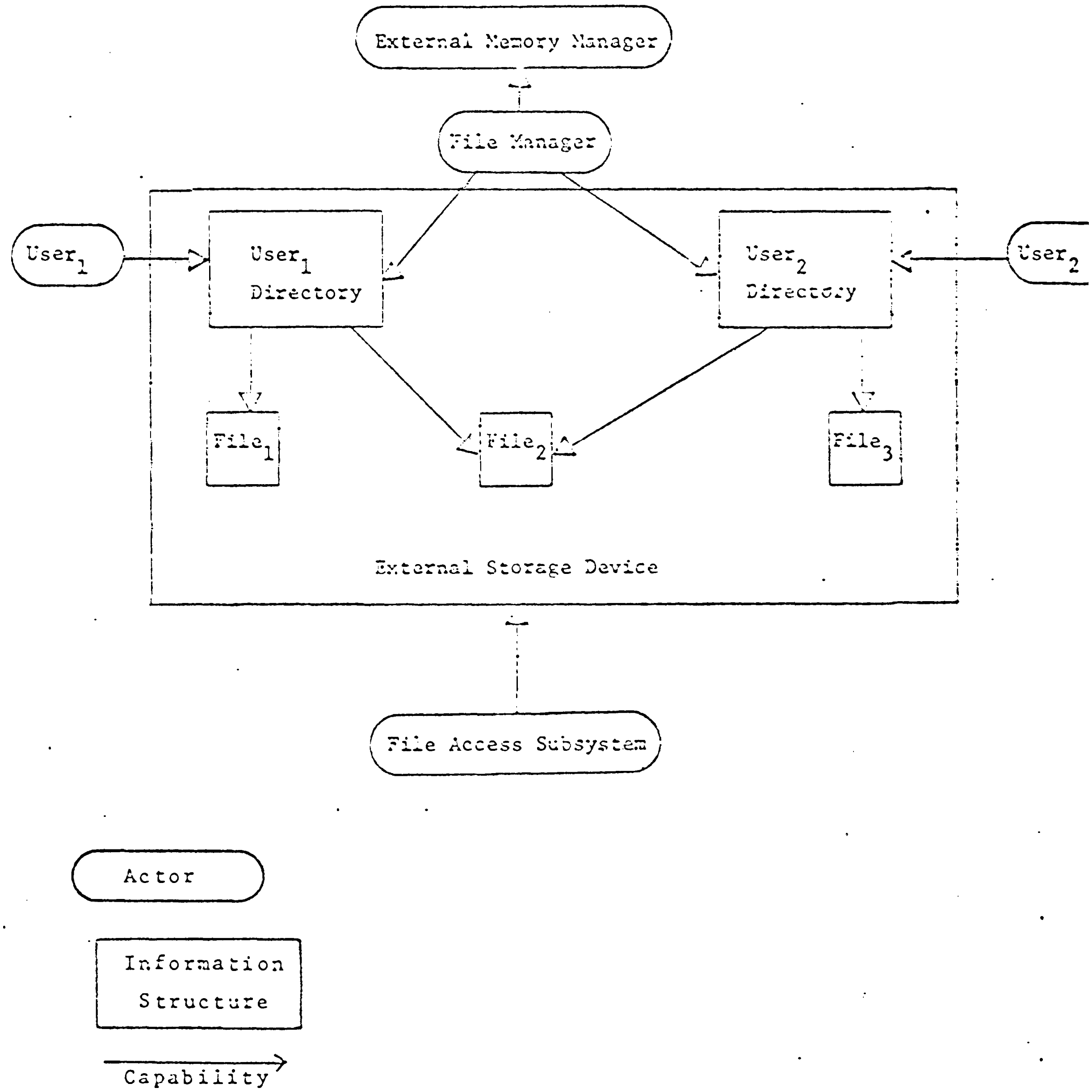

Figure 2

Relationship of Object Types

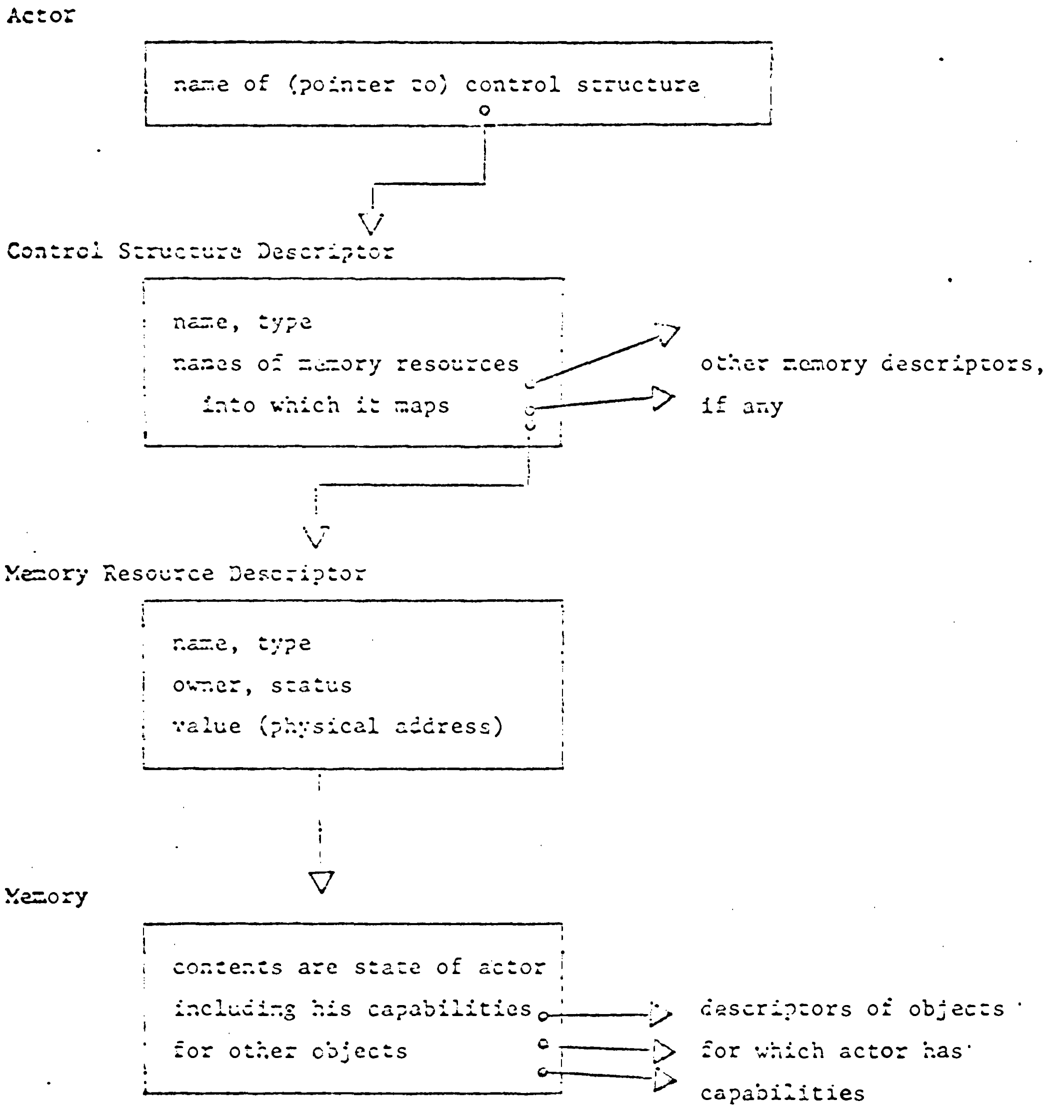

Figure 3

COPS Memory Management

Memory Descriptor

name, type, status, owner, value

Memory Capability

of an <u>owner</u>: (memory name, (Create, Allocate))

of a <u>manager</u>: (memory name, (Move, Deallocate, Changemap))

Memory Primitives

Move ($Memory_i$, $Memory_j$)

Allocate ($Memory_i$, $Actor_j$)

Deallocate ($Memory_i$, $Actor_j$)

Changemap ($Memory_i$, $Memory_j$)

Figure 4

Information Structure Management

Information Structure Descriptor

name, type, names of memory objects into which structure maps

Information Structure Capabilities

<u>form</u>: direct ((object name, part), (access, control))

indirect ((object name, part), (access, control), (access flags, control flags), monitor procedure name)

<u>attributes</u>:

object type	access	control
process	Read, Change, Switch	
procedure	Read, Change, Transfer	Grant, Ungrant Destroy
reference	Read, Change, Use	
computing	Read, Change, Execute	

Information Structure Primitives

Create (type, $Memory_1, \ldots, Memory_n$)

Grant (Capability, Control $Structure_i$)

Ungrant (Capability, Control $Structure_i$)

Destroy (Information $Structure_i$)

Figure 5

The Basic Monitor

```
procedure  Basic Monitor(action, operand_1,...,operand_n);
    /* determine if attempted action is authorized by capabilities in
       the Current Environment */
    begin
       for i:= 1 step 1 until n do
          if no capability in the Current Environment authorizes operand_i
             as the ith operand of action then return(deny action)
          else if a capability authorizing operand_i as the ith operand
             of action is indirect and the authorizing attribute is
             flagged then trap(monitor for operand_i);
       return(permit action)
    end Basic Monitor;
```

Figure 6

A Simple File System - External Memory Manager

```
procedure  External_memory_manager;
    /* This procedure has two entries, as indicated.  We assume that it
       owns all free external storage and manages all external storage
       which is in use. */

    begin
    entry: Request_storage (amount, user directory);
       begin
          If necessary compact file storage using Move and Changemap;
          Select free storage;
          Allocate(storage, File_manager);
          Return
       end;

    entry: Release_storage (storage, user directory)
       begin
          Deallocate(storage, user directory);
          Return
       end
    end External_memory_manager;
```

Figure 7

A Simple File System - File Manager

```
procedure File_manager;
    /* This procedure would have numerous entries for file and directory
       operations. We only illustrate file creation.*/
    begin
       entry: Create_file(user name, file type, size);
          begin
             Call Request_storage(size); /* get storage for File */
             fileid:= Create(type, storage); /* Create file */
             Allocate(storage,user directory);/* charge user for its
                                                  use */
             Grant( (fileid, (Read, Change, Use, Grant, Ungrant)), user
                    directory);/* give user access and Grant, Ungrant
                                  control for file */
             Ungrant( (fileid, (Read, Change, Use, Grant, Ungrant)),
                    File_manager);/* take all but "Destroy" control
                                      away from self */
             Return(fileid)
          end;

       other File_manager procedures

    end File_manager;
```

Figure 8

A Simple File System - File Access Procedure

```
procedure File_access (device, operation, file, buffer);
    /* This procedure performs IO on the specified device. We assume
       that the procedure has capabilities for all devices but no
       capabilities for buffers or files. */

    begin
       format I/O operation;
       doIO (device, operation, file, buffer);
       /* This procedure could transform the output (input) data, for
          example to encrypt (decrypt) it. */

    end File_access;
```