DYNAMIC CONTROL OF MODULE INTERCONNECTIONS

COINS Technical Report #84-35

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts, 01003

David Stemple
Stephen Vinter
Krithivasan Ramamritham

DYNAMIC CONTROL OF MODULE INTERCONNECTIONS

David Stemple
Stephen Vinter
Krithivasan Ramamritham

University of Massachusetts
Amherst, Massachusetts
July 1984

ABSTRACT

Large software systems require various levels of control in order to operate correctly and predictably. The levels range from control over the scope and types of variables to control over modules' connections to other modules. In this paper we examine the Gutenberg system, an operating system kernel designed to provide module interconnection control as one of its fundamental services. In the Gutenberg system, modules are processes that are connected to each other by communication channels called *ports*. Each port is associated with an abstract data type operation and can be created by a process only if the process has the *capability* to execute the operation on the type. Thus a port represents the privilege of one module, the port user, to request a service (an abstract data type operation) provided by another module (the type manager). Capabilities to create ports for requesting operations are contained in a *capability directory*, which is traversed by processes to gain these capabilities.

In the Gutenberg system, the ability of one module to connect to another can change dynamically through the traversal of the capability directory, through the sharing of subdirectories in the capability directory, and through the transfer of capabilities along ports. Controlling dynamic module interconnections requires controlling these actions. We present an example in which different access control policies are implemented using the module interconnection control facility provided by Gutenberg. This example demonstrates that an operating system kernel using a capability scheme for controlling module interconnections in terms of abstract data types is useful for exercising a robust range of control; more importantly, it shows that such control can be achieved without the high cost of a uniform capability-based approach to protection.

This material is based upon work supported in part by the National Science Foundation under grant MCS 82-02586.

1. INTRODUCTION

Large software systems require various levels of control in order to operate correctly and predictably. The levels range from control over the scope and types of variables to control over modules' connections to other modules. In this paper we examine the Gutenberg system, an operating system kernel designed to provide *module interconnection control* as one of its fundamental services [Ramamritham et al. 83a].

In general, a module is a program unit, for example, a procedure, a package [Dod 80], or a process. Typically, one module communicates with another to provide some service or to obtain some service. Services include controlling resources, managing abstract data types, providing and delivering data, storing data for future delivery, and providing access to other modules. Module interconnection refers to the communication relationships existing between modules. Module interconnection control refers to the specification and management of module interconnections.

Programming language facilities for interconnection control generally focus on the static properties of modules and are appropriate for the control of access within a process. These connections can involve access to variables, constants, data structures and entry points. Control over these accesses, both in terms of allowing them and ensuring their consistency, is most often exercised at compile-time, e. g., via strong typing and module nesting, as in Pascal.

On the other hand, the system described in this paper is designed to handle module interconnection problems beyond those manageable by compile-time or link-time checking mechanisms; namely, dynamic control of *process* interconnections.

In the Gutenberg system, modules are processes that are connected with each other through communication channels called *ports*. Each port is associated with an abstract data type operation and can be created by a process only if the process has the *capability* [Dennis and Van Horn 66] to execute the operation on the type. Thus a port represents the privilege of one module, the port user, to request a service (an abstract data type operation) provided by another module (the type manager). Capabilities to create ports for requesting operations are contained in a *capability directory*, which is navigated by processes to gain these capabilities. Ports in Gutenberg represent the only method of process interconnection and all direct communication between processes is by means of port operations.

In the Gutenberg system, the ability of one module to connect to another can change dynamically through the traversal of the capability directory, through the sharing of subdirectories in the capability directory, and through the transfer of capabilities along ports. Controlling dynamic module interconnections requires controlling these actions.

An important aspect of module interconnection control is ensuring that a process can only be connected to those processes whose services the process is *capable* of obtaining. In this sense, controlling process interconnections implicitly controls the services available to a process. When a service manifests itself as an operation on a resource, it is the access to the resource that is controlled. If multiple processes are allowed access to the resource, control of access also controls the sharing of the resource. Thus, control of process interconnections, control of access to resources, and *protection* of resources [Saltzer and Schroeder 75] deal with the same issue, and therefore are provided by a single mechanism in Gutenberg.

The emphasis of this paper is on the processes that access shared resources and those that manage these resources. In Gutenberg, a resource is an instance of an abstract data type (termed an *object*), and hence can be accessed only through the specific operations defined by the object's type. When a process requests service, it requests access to an object via an operation defined on the object; the service is provided (i.e., the operation is performed) by the process managing the object.

The rest of this paper is organized as follows. Section 2 briefly surveys previous approaches to module interconnection control. In section 3 we introduce ports and the capability directory, and discuss how shared objects are managed in the Gutenberg system. Section 4 discusses those features of Gutenberg which provide for the sharing and protection of objects. Section 5 is devoted to an application requiring dynamic control for achieving different access control policies. We consider the bibliography application first presented in [Wulf et al. 74] to demonstrate how Gutenberg achieves resource protection while providing flexible sharing. We show how the bibliography abstract data type is created, and how the creator of the type shares with other subjects the capability to create and access instances of the type under different access control policies.

2. APPROACHES TO MODULE INTERCONNECTION CONTROL

As mentioned in the introduction, module interconnection control in most programming languages is based on the static properties of processes and deals only with control of access to entities within a process. The work reported in [Kieburtz and Silberschatz 78, McGraw and Andrews 79, and Ancilotti et al. 83] incorporates capabilities into programming languages in an attempt to provide the flexibility of

controlling dynamic access to shared objects. Although this is an important extension of programming languages, the policies enforced by any language can be compromised in a number of ways. For example, the user could modify object code on secondary storage or develop programs in a different language. We feel this is an inherent limitation of the use of programming languages to provide protection, hence we have developed a programming language independent facility for controlling process interconnections. This facility is a part of the operating system kernel and thus is neither dependent on compilers nor, as we will see in subsequent sections, on the integrity of user-managed files.

There are other reasons to separate module interconnection control from programming language facilities. While we take a different approach to module interconnection control than that of DeRemer and Kron [DeRemer and Kron 76], we agree with their contention that "structuring a large collection of modules to form a system is an essentially distinct and different intellectual activity from that of constructing the individual modules." Therefore, we believe that the separation of Gutenberg module interconnection control from programming language facilities is justified on two counts. First, it is "natural" to use a distinct and different tool (the Gutenberg system) for a distinct and different activity (dynamically controlling the interconnections among a large collection of modules). Second, it provides for secure but *dynamic* control of interconnections in a manner that programming language methods do not.

Facilities for the control of module interconnection independent of particular programming languages have been proposed in [DeRemer and Kron 76, Tichy 79, Clarke et al. 83]. These facilities are declarative in nature and, like programming language facilities, focus on the static nature of interconnections rather than on dynamic control. Therefore, these facilities are inadequate for providing module interconnection control while supporting dynamic access to shared objects.

Capability-based operating systems, e. g., Plessey 250 [England 74], Hydra [Cohen and Jefferson 76], and iMAX [Kahn et al. 81], provide for dynamic control through the use of transferable, unforgeable capabilities for accessing modules. Capabilities are means for providing the control implicit in the abstract data type paradigm, i.e., the limiting of object access to a set of operations defined by the type. To date, however, capabilities have been expensive to implement either in software or hardware, and need further development before they can demonstrate their cost-effectiveness. The main problem seems to be their use for controlling accesses both to entities external to a module and to local data and procedures. The Gutenberg system represents an attempt to avoid this problem by using a capability scheme for controlling process interactions, but not for local procedure calls and other access within a process. We call this approach a *nonuniform object model of protection*: Data local to a process *are not* protected by the kernel; resources shared by multiple processes are structured as objects and *are* protected by the kernel. This approach, along with the avoidance of user-defined capabilities as discussed in the next section, distinguishes Gutenberg from other capability-based systems.

3. CONTROL OF PROCESS INTERCONNECTIONS IN GUTENBERG

In this section we first introduce the Gutenberg kernel objects, and then show how the control of access to kernel objects is used to control module interconnections providing access to user-defined objects.

Gutenberg's nonuniform object model of protection is based on a few basic *kernel objects* and the use of these system objects for expressing the protection requirements of *user-defined objects*. To help distinguish between kernel and user-defined objects we say that processes execute kernel *primitives* to manipulate kernel objects and request *operations* to manipulate *remote* user-defined objects, i. e., objects managed by other processes. The objects recognized by the kernel are:

- *processes*: the subjects which share access to objects;
- *ports*: used to request access to shared objects;
- *the capability directory*: a unified structure, organized similar to a UNIX file directory [Ritchie and Thompson 74], containing capabilities. It is a *stable* structure in that it existence does not depend on the existence of any process. It contains four kinds of stable capabilities:
  - *operation capability*: represents the privilege to create a port for use in requesting a named operation on a given user-defined object type; it is *linked* to a manager definition capability corresponding to the object type.
  - *manager definition capability*: represents the privilege to create new operation capabilities linked to this capability, and to redefine operations' implementations; it points to the executable image of the process that manages the object on which the operation is defined; corresponds to a type capability in Hydra.
  - *cooperation class capability*: represents the privilege to participate in a cooperative activity identified by a unique identifier, the *cooperation class identifier*. An example of a cooperative activity is communicating with the process that manages a shared instance of an object type.
  - *subdirectory capability*: represents the privilege to obtain a group

of capabilities, a subdirectory, in a restricted way. The capabilities contained in a subdirectory are said to be *registered* in that subdirectory.

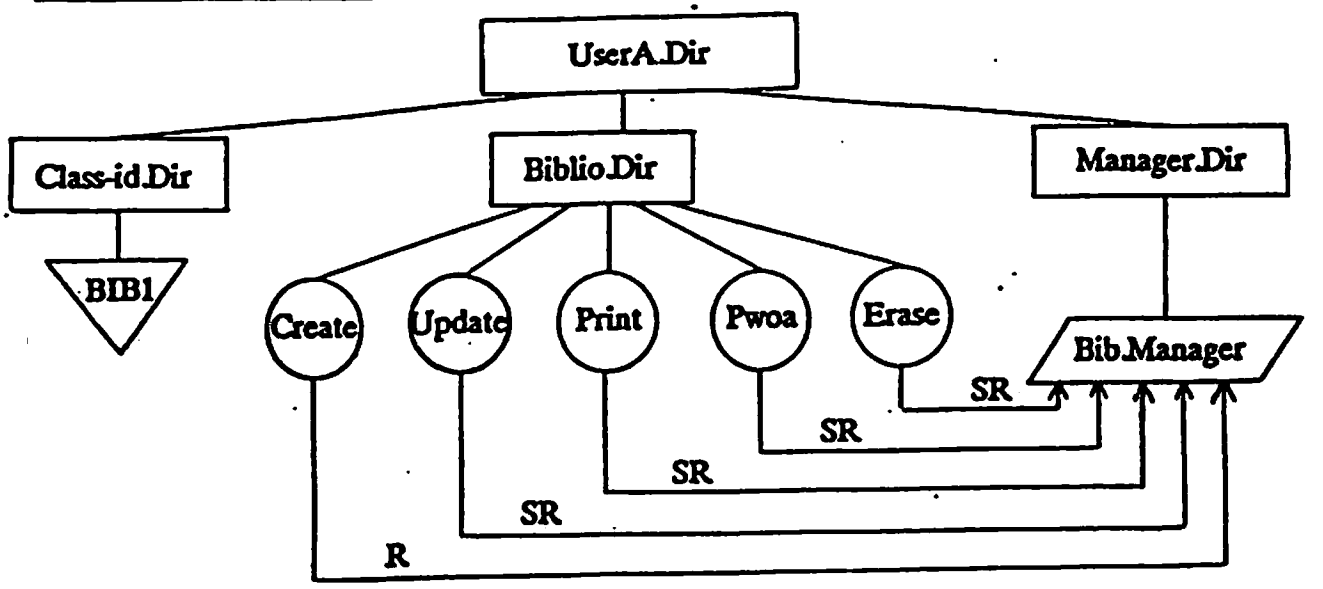
- *transient capabilities*: these capabilities are port capabilities and copies of stable capabilities from the capability directory; a process's transient capabilities are stored in its capability list *(c-list)*, and only exist for the duration of the process's existence.

A capability in other systems consists of: an object identifier, the object's type, and the subset of the operations defined by the type that are permitted by the capability. Since object types can be user-defined, the definition of operations in some capabilities are determined by the user.

Capabilities in Gutenberg are somewhat different, consisting of: a specific kernel primitive, a list of kernel-object identifiers constituting legal parameters for the primitive, and a list of primitives that can be used to manipulate the capability itself (called *capcaps*, for capabilities on a capability). Capcaps include the privilege to transfer, register (make stable), hold (make transient), and modify the capability. A capability permits a process with that capability to exercise the kernel primitive in the capability. For example, with an operation capability a process can perform the kernel primitive CREATEPORT with the name of the operation being a parameter to CREATEPORT.

The purpose of the capability directory is to restrict the acquisition of capabilities by processes in a consistent and orderly way. At any time a process has a single subdirectory of the capability directory designated as its *active directory*. A process can change its active directory to any subdirectory registered in its current active directory. (Figure 1 presents the structure of the subdirectory used for a bibliography application [Wulf et al. 74].) A process may only use stable capabilities

that are registered in its active directory. Each active directory has *rights* associated with it (independent of the capabilities registered in the subdirectory) that further restrict the primitives a process may use to manipulate and exercise the capabilities registered in the directory. These rights are determined by the particular subdirectory capability a process uses to make the subdirectory its active directory.



Rectangles denote subdirectory capabilities, circles denote operation capabilities, parallelograms denote manager definition capabilities and diamonds denote cooperation class identifier capabilities.

Bib.Manager manages an instance of a bibliography abstract data type. Create, Update, Print, Pwoa (Print without Annotations) and Erase are operations on the bibliography object. To request these operations, a process creates a port using one of the operation capabilities and then executes a RECEIVE (R) for Create and a SELECTRECEIVE (SR) for all other operations. BIB1 is a cooperation class capability and identifies a particular bibliography.

Figure 1: The bibliography subdirectory.

In addition to the capabilities in a process's active directory, a process may exercise the capabilities in its *capability list* (*c-list*). Capabilities in a process' c-list, unlike those registered in the capability directory, are owned by the process. Such capabilities are called *transient* because they exist only for the lifetime of the owning process. A transient capability comes into existence when a process *holds* a capability in its active directory (i. e., copies it into its c-list), receives the capability from another process via a port, or creates a port. In the last instance, the capability to access the created port is placed on the c-list. A process *holds* a capability in its active directory in order not to lose the capability when it changes its active directory to another subdirectory. Transient capabilities contribute to the flexible use of capabilities in Gutenberg without compromising the security of the system because the c-list may only be manipulated through kernel primitives.

The kernel maintains a list of port capabilities as part of a process' c-list, called the process's *p-list*. The p-list is part of the capability list because a port capability is a transient capability that gives a process the ability to execute a *particular* kernel primitive on the port. The possible kernel primitives on ports include SEND, RECEIVE, and SELECTRECEIVE (SELECTRECEIVE is intended for bidirectional communication).

Note that capabilities, ports and the capability directory are kernel objects and hence can themselves be manipulated only through the kernel primitives defined on them. For instance, the capability directory is traversed and its contents modified by processes through kernel primitives, e. g., CHANGE-DIRECTORY, REGISTER, and DELETE-MANAGER. When a process requests the execution of a kernel primitive the kernel uses the c-list and the active directory of the process to check whether it

has a legitimate privilege for executing the primitive.

We now move from kernel objects to user-defined objects and examine how user-defined objects are shared and protected. By user-defined objects we mean those objects managed by one process but accessible by other processes via operations requested using ports. We are not concerned here with objects purely local to a process since the Gutenberg kernel is not involved in their protection beyond the ordinary aspects of memory management.

Each user-defined object is managed by a manager process associated with the object's type. The creation of a user-defined type involves creating a manager definition capability (a kernel object) for the type, using the manager definition capability to create operation capabilities (kernel objects) corresponding to operations defined on the object type, and registering them in a subdirectory (a kernel object). The set of possible operations that may be performed on an object are determined by the operation capabilities linked to the object's manager definition capability.

In order for a process to request an operation on a shared object the process must create a port using one of the operation capabilities that is linked to the object's manager and then use the port to request the particular operation on the object. The operation capability must reside in the process' active directory or c-list when the port is created. In this way, operation capabilities are capabilities to create ports to access sharable objects. For example, in Figure 1 the only operations that can be performed on a bibliography are those for which operation capabilities (such as Create) are linked to the Bib.Manager manager definition capability. A process may request a bibliography operation only if it obtains one of the operation capabilities in Biblio.Dir (e. g., by making Biblio.Dir its active directory), creates a

port using the operation capability, and then uses the port to request the operation.

In Gutenberg, the only way a process can request an operation on a user-defined object is to execute a kernel primitive on a port, either SEND, RECEIVE, or SELECTRECEIVE. Ports are very restrictive in how they allow access to objects: First, the kernel associates the name of the operation in an operation capability with a port that is created using the capability. This association is not changeable and the operation is not included as a parameter to port primitives after port creation. Thus, a process may only use the port to request the operation for which the port was created. Second, each port capability entitles the owning process to execute only a single type of port primitive for sending and receiving, namely, SEND, RECEIVE, or SELECTRECEIVE.

In summary, a module interconnection manifests itself as a port between a process requesting an operation on a user-defined object and a process managing the object. Ports represent all module interconnections in Gutenberg. Establishing an inter-module connection, *i.e.*, a port, involves checking for an operation capability in the active directory (a subdirectory in the capability directory) or c-list of the requesting process. Thereafter the kernel performs access authorization for a user-defined operation simply by checking that the requesting process has the privilege to access a particular port. Thus, creating and accessing user-defined objects involves using system-defined capabilities to authorize access to system-defined objects, and does not involve checking user-defined capabilities as in other capability-based systems. Support for user-defined types is thereby achieved efficiently through the efficient implementation of kernel primitives.

4. SHARING AND PROTECTING OBJECTS IN THE GUTENBERG SYSTEM

In general, at a given instant, a process has the ability to access a sharable object if either of the following hold:

- The process has a port attached to the manager of the object.
- Operation capabilities in its active subdirectory or c-list permit the process to create a port to access the object.

The ability of process to access shared objects can change dynamically due to the following Gutenberg features:

- The dynamic initiation of and connection to manager processes of shared objects; the manager initiation protocols provide the needed flexibility to control the sharing of the objects managed by these processes.
- The transfer of capabilities on ports; by transferring its capability for an operation on an object, a process endows the recipient process with that capability, and this results in the sharing of access to the object between the two processes.
- The traversal of the capability directory; moving from one active directory to another provides a process with a different set of capabilities, and hence a different set of objects become accessible to it. Also, since a particular subdirectory may be reachable by two different processes, a process may implicitly share a privilege with another by placing it in such a subdirectory.

This section is devoted to a discussion of how dynamic sharing can be accomplished in Gutenberg. We examine how the protocols for initiating manager processes affect sharing in the Gutenberg system, introduce the Gutenberg capability transfer mechanisms, and discuss how capability directory traversal can be controlled. We conclude this section with a formal characterization of resource sharing in Gutenberg.

Recall that to perform an operation on a sharable object, a process must create a port to the object's manager. In order to create the port, the process must have a transient or stable operation capability, linked to the manager definition for

the object's manager, and must specify the operation name when the port is created. Based on the manager definitions, the kernel makes the determination of which protocol to use when it receives the first operation on a newly created port.

In the first protocol, a manager process is initiated from the manager definition only if there is no manager process currently in the system which was created using this manager definition. If a process initiated from the manager definition already exists, the port being created is attached to this process. Otherwise, a new process is created. This protocol provides a means to produce a manager process that manages all instances of an object type, and to automatically connect port-creating processes to this manager.

A second protocol allows new processes to be initiated selectively based on a parameter supplied at port creation time. This parameter, which is used to form groups of cooperating processes, is called a *cooperation class identifier* (class-id). A port can be associated with a class-id by specifying the class-id as a parameter to the port creation request. In this case, one of two actions occurs when the port is first used to access the object. If there is no current process associated with the class-id that was initiated from the definition, one is created and the port is attached to it. If such a process exists, the port is attached to it. Thus, each manager process created using this protocol is associated with a distinct class-id, can be designed to manage one instance of an object type, and may serve multiple ports. As a result, the processes with access to ports connected to such a manager process share access to the object instance managed by that process. (A complete discussion of how class-ids can be used to facilitate other forms of cooperative interaction among processes can be found in [Stemple et al. 83].)

Objects can also be shared between processes via the transfer of processes' capabilities. Since the kernel is the manager of the capability directory and ports, it controls processes' capabilities and monitors the transfer of capabilities between processes. There are three ways in which one process can transfer some of its capabilities to another; in all cases capabilities are transferred over a port connecting the sending and receiving processes.

The first method of capability transfer is the transfer of a port capability. The sending process loses the port capability, and therefore the privilege to execute the object operation associated with the transferred port; the receiving process obtains this privilege.

The second method of capability transfer is the transfer of an operation capability (which can be associated with a cooperation class) that can be used to create a port to access an object (identified by the cooperation class). In fact, the receiving process may use the operation capability to create many ports.

The third method of capability transfer is by the transfer of a subdirectory capability. The receiving process can use the subdirectory capability to make the subdirectory its active directory, thereby allowing it to use the capabilities in the subdirectory.

The two issues of concern when transferring capabilities are the *sharability* and *stability* of the transferred capabilities. Obviously a capability is sharable by the receiving process if the receiver can further transfer it to another process. A capability that can be registered in a subdirectory is also sharable since the subdirectory may be reachable by other processes.

A transferred capability (which is initially transient by virtue of being in the recipient's c-list) can be made stable by the receiving process registering the capability in its active directory. Capability registration allows a user[1] to create data types during one "run" and use them during a later "run". Otherwise, the capability does not exist beyond the life of the receiving process.

Through the use of capcaps the sending process has control over whether a capability can be registered or further transferred. In order to prevent a receiving process, say p2, that executes on behalf of a user U2, from further *transferring* a capability, a sender, say p1, executing on behalf of user U1, should set the capcaps of the transferred capability such that it cannot be further transferred. Sending process p1 can prevent p2 from further *sharing* the transferred capability by setting the capability's capcap so that neither its transfer nor its registration is possible by p2. Allowing a receiving process to register a capability also implicitly allows the process to share the capability because the subdirectory in which the capability is registered may be sharable. Thus, in general, registration implies sharability.

Though most registration of capabilities produces defacto sharing, one Gutenberg feature, the *private* subdirectory, allows registration of a capability while restricting its sharability. The capabilities registered in a user's private subdirectory may only be exercised by that user's processes; other users' processes may register capabilities in this subdirectory but may not exercise the capabilities. If capabilities are registered without the register and transfer capcaps set, the user's processes have no way of sharing the capabilities. The control over private subdirectories is provided

---

[1] The term user refers to any entity on whose behalf processes execute. A concrete example of a user is a person who can log into the system.

through the rights associated with subdirectories. Instead of user U1's process p1 transferring a capability to user U2's process p2, p1 registers the capability in U2's private subdirectory and sets the capcaps of the capability so as not to allow any of U2's processes to transfer or (re)register the capability. Thus, since a private subdirectory is only reachable by U2 processes, and U2 processes can neither transfer the capability to any other process nor register these capabilities in any other subdirectories, such capabilities are not sharable. This feature has two benefits. First, it allows a user to give another user permanent capabilities without allowing further distribution of the capabilities. Second, it allows a user to provide capabilities to other users without requiring that the recipients have an active process, while avoiding the use of access lists.

In order to precisely define the capabilities possessed by a process, we introduce some formalism.

Active_directory(p) stands for the active directory of process p. We will use active_directory(p) to refer to both the subdirectory capability for the active directory of p and the subdirectory itself. We will mean the former for d1 and the latter for d2 when we state that subdirectory d1 is registered in d2.

Capcaps(c) is the set of capabilities (privileges) to manipulate the capability c itself via kernel primitives. For example, if (register $\in$ capcaps(c)) and c is a transient capability, then c can be registered in any active directory, and thereby be made stable.

Rights(d) for a subdirectory capability d, denotes the set of rights for manipulating the capabilities registered in d possessed by a process making d its active directory through some traversal of the capability directory. For instance, if process p is to register a capability in a subdirectory d, then $(d = \text{active\_directory}(p)) \wedge (\text{register} \in \text{rights}(d))$.

Let D = Set of subdirectory capabilities in the capability directory,
M = Set of manager definition capabilities in the capability directory,
O = Set of operation capabilities in the capability directory, and
C = Set of cooperation class identifier capabilities in the capability directory.

Based on the following relations among capabilities in a capability directory,

DR $\subset$ D X D, where (d2 DR d1) if directory d2 is registered in d1,
MR $\subset$ M X D, where (m1 MR d1) if manager m1 is registered in d1,
OR $\subset$ O X D, where (o1 OR d1) if operation o1 is registered in d1, and
CR $\subset$ C X D, where (cc1 CR d1) if class identifier cc1 is registered in d1,

the capability directory can be viewed as representing the relation R where

R = DR $\cup$ MR $\cup$ OR $\cup$ CR.

Thus, we can say that capability c1 *is registered in* directory d1,
or more formally that the predicate **registered(c1,d1)** is true, if

c1 R d1.

Directory d2 *is reachable from* subdirectory d1, or more formally,
the predicate **reachable(d2,d1)** is true, if

d2 DR^* d1.

In general, capability c1 *is obtainable from* subdirectory d1,
or more formally, the predicate **obtainable(c1,d1)** is true, if

$\exists$ d2, reachable(d2,d1) $\wedge$ registered(c1,d2)

For example, for the capability directory in Figure 1, obtainable(CREATE,UserA.Dir) is true.

Note that the predicate **obtainable** only relates to the capabilities that are obtainable through the *traversal* of the capability directory; it does not include the capabilities obtained by a process via the *holding* of a capability registered in a different active directory. As mentioned earlier, a process with proper privilege may *hold* a capability that is in its active directory, causing a transient copy of it to be created and placed in the process's c-list, and then change its active directory. For a process p to make capability c registered in directory d transient, it is necessary

that:

(d = active_directory(p)) ∧ registered(c,d) ∧ (hold ∈ rights(d)) ∧ (hold ∈ capcaps(c))

We now turn our attention to the set of capabilities possessed by a process. To do so, we define some predicates: **dircap?(c)** is true if c denotes a subdirectory capability, false otherwise. Similarly, one can define **opcap?(c)**, **mancap?(c)**, and **classcap?(c)**. Let us define **c-list(p)** as the list of transient capabilities possessed by process p. Similarly, **p-list(p)** is the set of port capabilities possessed by process p. Additionally, we define **capabilities(p)** to be the set of capabilities process p has in its c-list as well as those obtainable from the current capabilities. Thus,

c ∈ capabilities(p) IFF
  c ∈ c-list(p) ∨
  c = active_directory(p) ∨
  [(d = active_directory(p)) ∧ obtainable(c,d)] ∨
  [d ∈ c-list(p) ∧ dircap?(d) ∧ obtainable(c,d)]

In the following summary of the effects of capability sharing mechanisms in Gutenberg, (i) refers to the effect on the process that initiates the sharing, (ii) refers to the effect on the recipient, and (iii) refers to the effect of the recipient's use of the transferred capability on other processes (including p1).

1) Transfer of a port capability by process p1 to p2
 (a) without the transfer capcap
  i. deletes the port capability from p1's p-list.
  ii. adds the port capability to p2's p-list.
  iii. cannot affect the p-list of any other process p3.
   (since p2 cannot transfer the port to any other process).

 (b) with the transfer capcap
  i. deletes the port capability from p1's p-list.
  ii. adds the port capability to p2's p-list.
  iii. any process can potentially use the port
   (since p2 can transfer the port to any process).

2) Transfer of a non-port capability c by process p1 to p2
 (a) without the register capcap and without the transfer capcap
  i. does not affect c-list(p1).

ii. adds c to c-list(p2).
iii. does not affect the c-list of any other process p3. May affect capabilities(p3) if, with c, process p2 can modify subdirectories accessible to p3.

(b) without the register capcap but with the transfer capcap
i. does not affect c-list(p1).
ii. adds c to c-list(p2).
iii. may affect the c-list of process p3 if p2 transfers c to p3. May affect capabilities(p3) if, with c, p2 can modify subdirectories accessible to p3.

(c) with the register capcap
i. does not affect c-list(p1).
ii. adds c to c-list(p2); also, c can be registered if there is a capability d where: (d $\in$ capabilities(p2)) $\wedge$ dircap?(d) $\wedge$ (register $\in$ rights(d)).
iii. c is available to any process p3 if (c $\in$ capabilities(p3)) after it is registered by p2. May affect capabilities(p3) if, with c, p2 can modify subdirectories accessible to p3.

3) Register of capability c in a subdirectory by process p1
(a) by registering c in a private directory accessible to p2
i. does not affect c-list(p1).
ii. adds c to capabilities(p2).
iii. may affect capabilities(p3) if, with c, p2 can modify subdirectories accessible to p3.

(b) by registering c in a directory d, where (d $\in$ capabilities(p2))
i. does not affect c-list(p1).
ii. adds c to capabilities(p2).
iii. may add to process p3's capabilities, if (d $\in$ capabilities(p3)). Also, if c is transferable, p2 may transfer it to p3. May affect capabilities(p3) if, with c, p2 can modify subdirectories accessible to p3.

In summary, to ensure that a transferred capability is available to no process other than the one it is transferred to, the transferring process should:

1. exclude both transfer and register capcaps, in which case the transferred capability will exist only as long as the receiving process exists, or

2. register the capability (without the register or transfer capcaps set) in the recipient's private directory, in which case the transferred capability will remain *stable* but unsharable.

The complete effect of transferring a capability with the transfer capcap or the

register capcap can be specified only if the behavior of each process in the system is specified. Given a dynamic system, this problem is undecidable [Harrison and Ruzzo 76].[2]

As the example in the next section illustrates, the capability directory traversal and the capability transfer mechanisms discussed above can be used to obtain differing degrees of access control in the dynamic sharing of objects.

5. CONTROLLED SHARING OF A BIBLIOGRAPHY SYSTEM

We have adapted the bibliography abstract data type example that was presented in [Wulf et al. 74]. After reviewing the example, we show how the bibliography data type is created, and investigate ways in which the type's creator may share access to bibliographies and the bibliography program.

The bibliography system allows users to store, manage, and display annotated references. A bibliography is managed by an abstract data type manager, which recognizes a legal set of operations on the bibliography data type. These operations are *Create*, to create a new bibliography; *Update*, to modify bibiographic entries; *Print*, to display entries; *Print without annotations (PWOA)*; and *Erase*, to erase a bibliography.

---

[2] It sometimes becomes necessary to revoke capabilities transferred to a process. Capability revocation is beyond the scope of this paper. However, details regarding capability revocation in Gutenberg may be found in [Ramamritham et al. 83b].

User A creates the bibliography data type by registering a manager capability, Bib.Manager, in subdirectory Manager.Dir, and operations on the type in a subdirectory Biblio.Dir in the capability directory (see Figure 1). We would like to have each bibliography managed by an independent process, and allow bibliographies to be shared. Hence processes should be initiated from Bib.Manager using the second manager initiation protocol discussed in the previous section; each port will be associated with a cooperation class-id which identifies the bibliography accessed via the port. Biblio.Dir, as shown in Figure 1, contains one operation capability for each operation defined on the type, each of which is linked to Bib.Manager.

Suppose user A, the bibliography data type creator, desires to create a bibliography. To do so, User A (1) obtains a cooperation class-id, BIB1, (2) invokes the kernel primitive to create a port P, specifying the CREATE operation and the class-id, and (3) executes a RECEIVE operation on P, causing the kernel to initiate a new manager process. The new manager creates a new, empty bibliography and *sends* an acknowledgement to user A on port P. The manager process and the newly created bibliography are now associated with class-id BIB1. This means all ports created using class-id BIB1 and the operation capabilities in Biblio.Dir will be attached to this manager process. To perform an operation on the bibliography user A creates a port specifying the operation and the class-id and executes a SELECTRECEIVE sending request details to the manager via the port; the manager performs the operations and sends the results to the requester via the same port.

Since, in general, bibliographies are permanent entities, it is necessary to register BIB1 in a subdirectory that A has access to so that subsequent processes, created on behalf of A, have access to bibliography BIB1. Hence A registers BIB1 in Class-id.Dir.

User A can share his bibliographies and the bibliography system using the capability transfer mechanism in three ways:

- by transferring to other processes the capability for a subdirectory containing operation capabilities for the object type (such as the capability for Biblio.Dir in Figure 1);
- by transferring capabilities to create a port (which are transient capabilities created from the operation capabilities); or
- by transferring ports connected to the manager of a specific bibliography.

We begin by examining the transferring of subdirectory capabilities.

Assume user A wishes to share the bibliography system, but not his bibliographies, with user B. That is, A wants to let user B create and use B's own bibliographies. A can accomplish such sharing by transferring B a subdirectory capability for the Biblio.Dir subdirectory. B can request class-id's from the kernel, use them to create new bibliographies, and perform the full set of operations on the bibliographies. However, because B has not been given the class-id's identifying A's bibliographies, and class-id's are unforgeable, B is incapable of obtaining capabilities for A's bibliographies.

Suppose user A wants to ensure that user B will not destroy Biblio.Dir nor delete any operation capabilities in it. A can protect his capabilities yet share them by not including the register or delete rights in the subdirectory capability he transfers to B. B can then make the subdirectory his active directory, but will not

be able change its contents in any way. Another approach is for A to create a new subdirectory, register in it the capabilities for operations that A desires to let B use, and transfer to B a full capability (e.g, including the register and delete rights) for this subdirectory.

Assume user A wishes to share bibliography BIB1 with user C. However, A desires to transfer to user C only the capabilities to the Print and PWOA operations on BIB1. A accomplishes this by transferring a subdirectory capability for a subdirectory containing the Print and PWOA operation capabilities that have BIB1 *merged* with them. This merging has two meanings. First, C may only use the operations with this particular class-id. Second, the class-id may not be separated from the operation capabilities, and so C cannot use the class-id with an Erase or Update operation capability that it may have received from another user.

As we saw in section 4, when transferring a capability to another process, user A has complete control over whether the transferred capability includes the transfer and register capcaps. By not allowing the recipient process to register the received subdirectory capability or to transfer it to a third process, A can be confident that the transferred capabilities are not shared with other processes. As also discussed in section 4, transferring a capability without the transfer capcap does not by itself guarantee that the transferred capability will not be shared. For example, assume user A transfers a capability to a user D process without the transfer capcap but with the register capcap. This allows D to register the capability in one of its subdirectories, and thus permanently store it. If this subdirectory is shared, the transferred capability can also be shared. As we saw earlier, via the use of private subdirectories, Gutenberg permits a recipient to make a shared capability stable while preventing its

further sharing. Thus, instead of transferring a capability to user D, user A registers the subdirectory capability without the transfer and register capcaps in D's private subdirectory, and then only user D processes may use that capability.

So far we have discussed the controlled sharing of capabilities via the transfer of subdirectory capabilities. The second way A can give other processes capabilities is by transferring capabilities that directly allow a process to create ports to the manager of a bibliography. As was the case with the operation capabilities that user C received from user A, the capabilities may be merged with class-id's that restrict their use to particular bibliography instances, and may or may not allow their transfer and registration.

Passing individual capabilities to create ports instead of registering them in a subdirectory and transferring the subdirectory capability does not provide additional functionality, but does provide finer granularity and additional flexibility that may prove to be more convenient without reducing access control. By transferring capabilities to create ports, A does not have to go to the trouble of creating a subdirectory, registering capabilities in it, and managing it after transferring a capability for it.

The third way to share access to bibliographies is for a process to create ports to a bibliography manager and transfer them to another process. In this case each port is already associated with a specific operation on a specific bibliography. Ports are shared differently than operation capabilities from which they were created in that exactly one process is connected to each end of the port. Thus, sharing is at a finer granularity since ports constitute a single access path to a single object. Last, ports are inherently temporary, and are destroyed when their owning processes

terminate.

We have discussed the sharing of capabilities and explored how the register and transfer capcaps affect sharing. User A, as creator of the bibliography data type, was able to share his bibliography system and specific bibliographies, and choose different degrees of access control to meet his requirements. In general, by restricting the processes that have access to a set of operations, class-id's (identifying instances of abstract data types), class-id/operation combinations, and subdirectories, the creator of an abstract data type has complete control over all capabilities that are shared with respect to instances of the type. The creator also has control over the transfer and registration of transferred capabilities. We illustrated the use of the Gutenberg approach to dynamic sharing of resources via the bibliography example.

In general, this approach is applicable to any situation where access to modules providing services to other modules needs to be shared. The creator of a module may implement any access control policy he desires with respect to the services the module provides. For example, if the creator of a data type wants an instance of the type created only if the requestor possesses the proper password, then the creating process can retain the create capability, perform the password authorization itself, and distribute the appropriate capabilities to the authorized requesting process. This is a reference monitor approach [Ames et al. 83] to access authorization that is implemented at the process level where capability management is implemented efficiently at the kernel level.

6. SUMMARY

We have presented a module interconnection specification and control facility which is independent of programming language features and is provided by an operating system kernel. We have argued that such a facility is needed to provide dynamic control of object sharing to simplify the construction of correct and predictable software systems. We also presented a bibliography system in order to illustrate how this facility could be used to control sharing when different kinds of access control were required. This example demonstrated that an operating system kernel using a capability scheme for controlling module interconnections in terms of abstract data types is useful for implementing a variety of access control policies.

We have also shown that a capability-based kernel need not exercise control over local data access, as for example in Hydra [Wolf et al. 74], in order to elegantly control the sharing of non-local data. This nonuniform use of capabilities provides, we believe, a feasible method of providing dynamic control of access where needed without incurring the crippling inefficiencies of uniform capability-based systems.

7. REFERENCES

[Ames et al. 83] Ames, S., Gasser, M., Schell, R., "Security Kernel Design and Implementation: An Introduction," Computer, vol. 16, no. 7, July 1983.

[Ancilotti et al. 83] Ancilotti, P., Boari, M., Lijtmaer, N., "Language Features for Access Control," IEEE Transactions on Software Engineering, vol. SE-9, no. 1, Jan. 1983.

[Clarke et al. 83] Clarke, L., Wileden, J., Wolf, A., "Precise Interface Control: System Structure, Language Constructs, and Support Environment," University of Massachusetts, COINS Technical Report #83-26, Aug. 1983.

[Cohen and Jefferson 76] Cohen, E., Jefferson, D., "Protection in the Hydra Operating System," Proceedings of the 5th Symposium on Operating System Principles, vol. 9, no. 5, 1976.

[Dennis and Van Horn 66] Dennis, J., Van Horn, E., "Programming Semantics for Multiprogrammed Computations," Communications of the ACM, vol. 9, no. 3, March 1966.

[DeRemer and Kron 76] DeRemer, F., Kron, H., "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Transactions on Software Engineering, vol. SE-2, no. 2, June 1976.

[Dod 80] "Reference Manual for the Ada Programming Language", U.S. Department of Defense, July 1980.

[England 74] England, D., "Capability Concept Mechanism and Structure in System 250," Proceedings, Internation Workshop on Protection in Operating Systems, Aug. 1974.

[Harrison and Ruzzo 76] Harrison, M.A. and Ruzzo, W.L., "Protection in Operating Systems," Communications of the ACM, Vol 19, No 8, Aug 1976, 461-470.

[Kahn et al. 81] Kahn, K., Corwin, W., Dennis, T., D'Hooge, H., Hubka, D., Hutchins, L., Montague, J., Pollack, F., "iMAX: A Multiprocessor Operating System for an Object-Based Computer," Proceedings, 8th ACM Symposium on Operating System Principles, Dec. 1981.

[Kieburtz and Silberschatz 78] Kieburtz, B., Silberschatz, A., "Capability Managers," IEEE Transactions on Software Engineering, vol. SE-4, no. 6, Nov. 1978.

[McGraw and Andrews 79] McGraw, J., and Andrews, G., "Access Control in Parallel Programs," IEEE Transactions on Software Engineering, vol. SE-5, no. 1, Jan. 1976.

[Ramamritham et al. 83a] Ramamritham, K., Vinter, S. and Stemple, D., "Primitives for accessing protected objects," Proc. Third Symposium on Reliability in Distributed Software and Database Systems, Oct. 1983.

[Ramamritham et al. 83b] Ramamritham, K., Briggs, D., Vinter, S. and Stemple, D., "Privilege Transfer and Revocation in a Port-based System," submitted for journal publication, Oct. 1983.

[Ritchie and Thompson 74] Ritchie, D. and Thompson, K., "The UNIX Time-Sharing System," Communications of the ACM, vol. 17, no. 7, July 1974.

[Saltzer and Schroeder 75] Saltzer, J.H. and Schroeder, M.D., "The Protection of Information in Computer Systems," Proceedings of the IEEE, vol. 63, no. 9, Sept. 1975, 1278-1308.

[Stemple et al. 83] Stemple, D., Vinter, S. and Ramamritham, K., "Interprocess Communication without Process Identifiers," submitted for journal publication, Oct. 1983.

[Tichy 79] Tichy, W., "Software Development Control Based on Module Interconnection," Proceedings, Fourth International Conference on Software Engineering, Sept. 1979.

[Wulf et al. 74] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., Pollack, F., "HYDRA: The Kernel of a Multiprocessor Operating System," Communications of the ACM, vol. 17, no. 6, June 1974.