PRIVILEGE TRANSFER AND REVOCATION

IN A PORT-BASED SYSTEM

COINS Technical Report #84-34

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

Krithivasan Ramamritham
David Stemple
David Briggs
Stephen Vinter

PRIVILEGE TRANSFER AND REVOCATION IN A PORT-BASED SYSTEM [1]

Krithivasan Ramamritham
David Stemple
David Briggs
Stephen Vinter

University of Massachusetts
Amherst, Massachusetts
October, 1983, revised September, 1984

ABSTRACT

Gutenberg is a port-based operating system being designed to study protection issues in distributed systems. In the Gutenberg system, all shared resources are viewed as protected objects and hence can be accessed only via specific operations defined on them. Processes communicate and access objects through the use of 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 the port's *client* process to request a *service* (an abstract data type operation) provided by the port's *server* process (the type's manager). Capabilities to create ports for requesting operations are contained in a *capability directory*, which is navigated by processes to gain these capabilities.

Privilege transfer is a means of providing servers access to the resources they need to perform their services. In Gutenberg, privilege transfer is accomplished by allowing access to subdirectories of the capability directory and by passing capabilities, including port access capabilities, to processes via ports. It should be possible to revoke transferred privileges when breaches of trust are detected or suspected, when a period of time has passed beyond which the distributor of a privilege does not want the privilege shared, or when an error has been detected. Transfer and revocation of privileges in Gutenberg is the subject of this paper. In particular, we describe the types of privileges in Gutenberg, discuss the mechanisms provided for the transfer and revocation of different types of privileges, and sketch the means for handling exceptions during privilege transfer.

---

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

1. INTRODUCTION

Large software systems are best organized by breaking them down into modules which communicate in limited ways and whose privileges are restricted to those needed to perform the modules' designated services. One way of restricting the privileges of an executing module is for the initiator of the module to explicitly supply the privileges at the time of module initiation. Another way is for the module to receive privileges during execution along communication channels connecting it to its communication partners. Other methods include the granting of privileges, to the module, based on the user (an entity external to the system) on whose behalf the module is being initiated. In this paper, we are concerned mainly with the first two methods of explicit privilege transfer, i. e., during module initiation and as a part of interprocess communication.

In *object-oriented systems*, resources are structured as objects which can be accessed only via the operations defined on them. In such systems, the privileges of a process specify the objects that it can access and the operations that it is allowed to perform on them. A *protection scheme* permits a module to perform operations on a specified object only if the module has the appropriate privilege. *Object-oriented privilege transfer* refers to mechanisms in which privileges transferred are expressed in terms of operations on specific objects. Such mechanisms are provided in several systems **[Ancilotti et al. 83, Kahn et al. 81, Wulf et al. 74]**.

In this paper, we deal with the transfer of privileges in the Gutenberg operating system [Ramamritham et al. 83, Stemple et al. 83], which uses an object-oriented protection scheme to control the interaction of modules represented by processes. Gutenberg is also a *port-based* system in that all communication between processes, including process initiation, is accomplished using ports.

Though not always provided by programming languages or operating systems, the ability to revoke transferred privileges is often useful. Revocation is appropriate, for instance, when a breach of trust has been detected or is suspected, where a period of time has elapsed beyond which it is not desired to share the transferred privilege, and where recovery from an error requires the revocation of a transferred privilege. Hence, Gutenberg provides for the revocation of transferred privileges. One of the assumptions we have made in designing the revocation mechanism is that the occasions for revocation are exceptional and infrequent as opposed to the frequent need for transferring privileges; thus, mechanisms designed for revocation should not unduly complicate nor slow down the transfer mechanism in situations where it is known that the transferred privileges will not be revoked.

Gligor has designed a revocation scheme in which revocation mechanisms form a subsystem above the (verified) protection kernel [Gligor 79]. The premise underlying such a design is that unauthorized retention of access privileges resulting from incorrect behavior of the revocation subsystem will not violate protection requirements; hence, unlike the protection kernel which needs to be verified, the revocation subsystem need not be verified. We believe that retention of revoked privileges is equivalent to the possession of unauthorized privileges and is tantamount to a protection violation. Retention of revoked privilege is not to be allowed or

relegated to any mechanism outside the protection kernel. Thus, the Gutenberg kernel includes the revocation mechanisms, and they will be verified along with the privilege transfer mechanisms.

Systems designed with the goal of protecting resources incur the overheads of checking the privileges of processes when they attempt to access resources. In a dynamic system, i. e., one which allows transfer of privileges during the execution of a process and not just at its initiation (e.g., see **[Stemple et al. 84b]**), the privileges of a process vary depending on what privileges are transferred to it. Hence the system has to keep track of transferred privileges and update the privileges of the processes involved in transfers. If, in addition, a system allows privileges to be revoked, further overheads will be incurred, since in this case the system must keep track of how a receiving process uses the transferred privileges in order to prevent their use after revocation.

Since there are considerable overheads involved in privilege transfer and revocation, it is imperative that protection systems be designed so that these overheads are minimal. One way of achieving this goal is to keep the kernel as simple as possible while providing sufficient flexibility in the protection facilities. Thus, compromises must be made. The set of kernel primitives, the privilege transfer and revocation mechanisms, and the classification of privileges in Gutenberg are the results of the compromises made in meeting the conflicting requirements of simplicity, flexibility and usefulness. In subsequent sections, as we describe the kernel primitives and the protection scheme, we present the design choices and the motivation for our decisions.

Though Gutenberg has been designed to study issues in distributed systems, transfer and revocation of privileges are discussed in this paper only in the context of a centralized system. Research dealing with the issues of distribution is currently in progress and will be reported in a future paper.

The remainder of the paper is organized as follows. The next section presents the main features of the Gutenberg system. It is followed by the details of privilege transfer using the Gutenberg primitives. Revocation of privileges is then discussed and its aspects classified. We then present the Gutenberg means of revoking privileges. Following this we discuss the disruptions possible during privilege transfer and the mechanisms involved in assuring reliable behavior in the presence of disruptions. We end with a summary section.

2. THE GUTENBERG SYSTEM

Gutenberg is a port-based, object-oriented kernel being designed to study protection issues in distributed systems. In the Gutenberg system, all shared resources are viewed as protected objects and hence have specific operations defined on them. Processes communicate and access sharable objects through the use of ports. The basic principle of protection in Gutenberg is: *A port connecting a process to the manager of an object exists only if the process has the privilege to access the object with a specific operation.*

Capability-based operating systems, e. g., Plessey 250 [England 74], CAL [Lampson and Sturgis 76], Hydra [Cohen and Jefferson 76], and iMAX [Kahn et al. 81], provide for the protection of objects through the use of transferable, unforgeable capabilities for accessing objects. 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. This approach we have called 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 use of ports for all access to user-defined, shared objects, distinguishes Gutenberg from other capability-based systems.

Another difference between Gutenberg and other proposed protection systems **[Wulf et al. 74, Lampson and Sturgis 76, Needham and Walker 77]** arises from the way privileges persist in the system. In other systems, the possessor of a privilege is allowed to store the privilege in long-lived user-managed objects, such as files or directories. In Gutenberg, all privileges which persist independently of the existence of processes are maintained in a system capability directory managed solely by the kernel.

Other salient features of Gutenberg are use of the client-server model in port communication and a strict adherence to *functional addressing* in the establishment of ports **[Vinter et al. 83]**. The latter feature is characterized by the creation of ports for the purpose of requesting specified functions rather than to connect to specified processes: A port created by one process is connected to a second process based on

a specified function (a service or abstract data type operation) and a *cooperation class* with which the service is to be associated. The concept and implementation of cooperation class in Gutenberg is itself a unique feature and is treated in a separate paper **[Stemple et al. 84a]**.

2.1 Kernel Objects and Primitives

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.

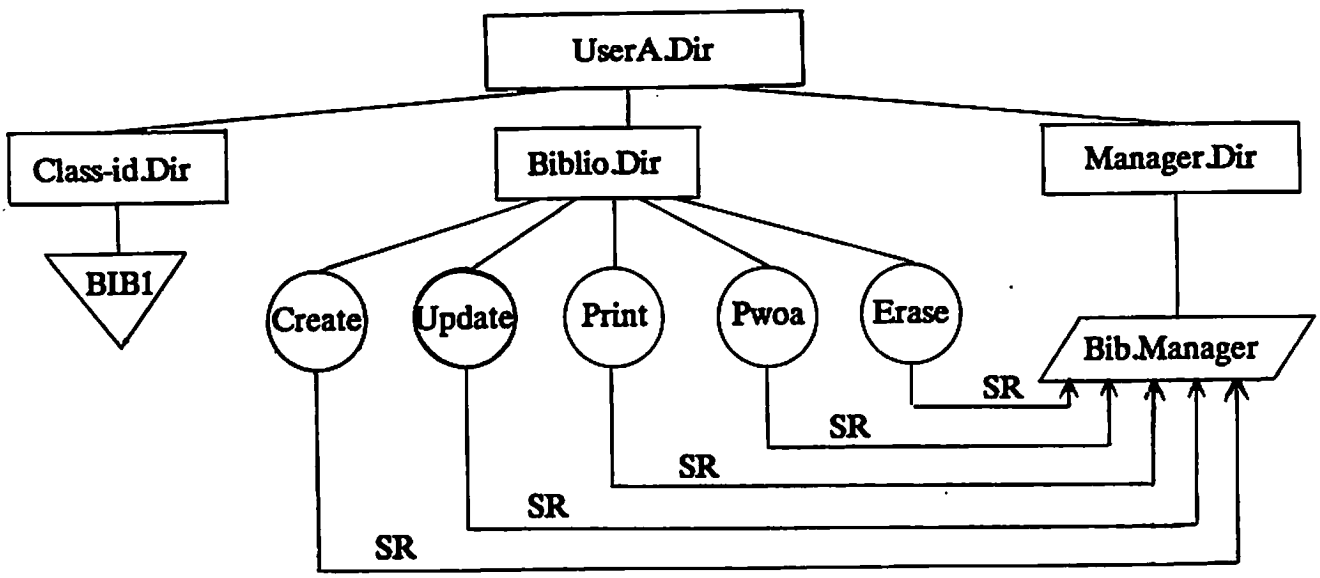
- *transient capabilities*: these capabilities are port capabilities and copies of stable capabilities from the capability directory; a process's transient capabilities are stored in its capability list *(c-list)*, and only exist for the duration of the process's existence.

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 SELECTRECEIVE, denoted by the SR, on the port. 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.

2.2 User-defined Objects

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. 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. The (single) sending or receiving primitive that can be executed by a client on a port is a parameter contained in the operation capability used by the client to create the port. For example, the SR labels (for SELECTRECEIVE) on the links between the operation names and Bib.Manager in figure 1 mean that the SELECTRECEIVE primitive must be used to request each of the bibliography operations.

2.3 Port Primitives

Ports are instances of an abstract data type whose manager is the system kernel and constitute the communication links between processes. At any time in a port's existence it is linked to exactly two processes, its *client* and its *server*. The one-to-one nature of Gutenberg ports was chosen to reduce the complexity of the kernel and because more complex interconnections can be simulated using such ports [Vinter et al. 83]. Both the client and the server processes are limited to a small set of primitives they can execute on the port. Operation capabilities are used at port

creation time to determine the limitations on the port, including whether (and what kinds of) privileges (capabilities) can be transferred on the port.

Client primitives and the corresponding server responses are listed in the following table. When a port is created, the creator, the initial client, gets the privilege, termed the *client privilege*, to execute one of the sets of client primitives in the table, e. g., SEND and REVOKE. When a process is chosen as the server of the port, it gets the privilege, the *server privilege*, to execute the corresponding server primitives, e. g., RECEIVE and REFUSE.

Client Primitives	Server Primitives
SEND REVOKE	RECEIVE REFUSE
RECEIVE	SEND REFUSE
SELECTRECEIVE REVOKE	GETDETAILS SEND REFUSE

Note that exactly one of the two processes has the privilege to execute the SEND primitive on a port.

We now give a short summary of the purposes of these primitives. (Details concerning the primitives may be found in [Ramamritham et al. 83]).

- RECEIVE requests the next message from the port. The caller elects via a parameter to the call, to either block, if there is no message on the port, or execute concurrently with the servicing of the request.

- SEND puts a message on a port. The system has two kinds of SEND primitives: acknowledge-SEND and no-acknowledge-SEND. If the SEND is an acknowledge-SEND, the sending process is informed when its correspondent over the port receives the message, and can choose to block until the receipt of the acknowledgement.

- SELECTRECEIVE (only a client primitive) puts information, termed *request details*, on a port for the server to use in satisfying the request, and when the server responds to the request by executing a SEND, the server's reply is returned to the client as in RECEIVE. The caller may block until the server replies, or execute concurrently with the servicing of the request.

- GETDETAILS gets request details from a port. The caller (the port's server) may block if there is no pending SELECTRECEIVE, and thus no request details, on the port, or it may execute concurrently with the satisfaction of its request.

- REFUSE rejects a client's request for service as unsatisfiable and notifies the requester by setting a status.

- REVOKE revokes privileges sent as part of request details by a SELECTRECEIVE (or in a SEND message up to receipt of the message). The details of this primitive are dealt with in sections 4, 5 and 6.

The choice of these primitives in the system design was based on the desire to keep their number and complexity to a minimum while affording users a set of primitives for building systems of communicating processes with reasonable ease. Thus, we have added to the basic SEND and RECEIVE primitives the bidirectional SELECTRECEIVE and its receiving reciprocal GETDETAILS in order to allow such functions as reading a record with a given key (the key being sent as request details) or a remote procedure call (the procedure's parameters being sent as request details) to be implemented by a single primitive.

In addition to its client and its server, a port has associated with it the process which created it, its *owner*. Initially, the owner and the client of the port are the same process, but, as detailed below, the owner may transfer its privilege as the

port's client to another process, at which point the owner and client of the port are different processes. The owner of a port may destroy the port by calling the kernel with the DESTROYPORT primitive, provided it is the port's client at that time. This restriction prevents the owner from sabotaging another process's use of a port. The owner may transfer its destroy privilege to another process; the receiving process is then regarded by the system as the port's owner. Thus, ownership of a port is identified with the privilege to destroy it.

3. PRIVILEGE TRANSFER IN GUTENBERG

In Gutenberg, privileges are transferred for two purposes. The first is to allow one process, say A, to give another the privilege to access objects that A has the privilege to access. The second purpose is to allow servers to give their server privileges to helper or surrogate processes. A process transfers privileges by sending capabilities via ports. The kernel effects privilege transfer by adding to the receiving process' c-list the capabilities which represent the transferred privilege, and, in some cases, to be discussed below, removing capabilities from the sending process' c-list. In this section we present the rules and restrictions on privilege transfer in the Gutenberg system and describe how privilege transfer takes place. The revocation of transferred privileges is described in the next sections.

There are three ways in which one process can pass an object accessing privilege to another process. The first method is for the client of a port to send its privilege for requesting an operation via the port to another process (using either SEND or SELECTRECEIVE on a second port). The receiving process gets the privilege to access the object via the passed port. The sending process loses the

access privilege, either permanently (SEND) or temporarily (SELECTRECEIVE) because of the one client, one server restriction on Gutenberg ports. The second method of passing privileges to access an object is to pass an operation capability that can be used to create a port for accessing the object. The third method is to transfer a subdirectory capability to another process. The receiving process can use the subdirectory capability to make the subdirectory its active directory, thereby obtaining the privileges contained in the subdirectory. In the last two cases, the sender process does not lose the transferred privilege.

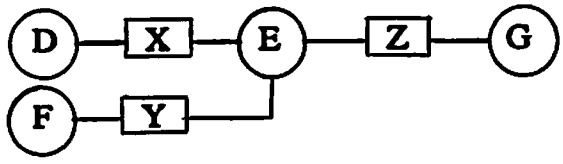
The second purpose of privilege transfer is so that port servers can send their server privilege to other processes, in order to allow the other processes to help in providing the service. As in the case of port clients, such transfer can be permanent (SEND) or temporary (SELECTRECEIVE). Transfer of server privileges is not visible to the client processes.

The ability to transfer port client privileges and the concomitant privileges to access objects facilitates the hierarchical decomposition of systems in the same manner as transferring access privileges via arguments in procedure calls. The ability to transfer both client and server privileges for ports permits interprocess network topologies beyond the hierarchy implied by a pure client-server model of process interrelationships. A need for nonhierarchical interprocess communications arises, for example, in implementing a mixed demand and data driven protocol for database query execution **[Stemple et al. 84a]**.

A process can also transfer a privilege *derived from* the privilege(s) it possesses. This derivation can be done in one of the following ways:

1. *Intersection* of a set of privileges (see figure 2).

2. *Application* of an existing privilege to obtain another. For example, applying the privilege to create ports to derive a port privilege; applying the privilege to a subdirectory to derive a privilege defined within the subdirectory.



X = privilege to create ports to access any object of type T through operation OP defined on type T.
Y = privilege to create ports to access object O of type T through any operation defined on the objects.
Z = the intersection of X and Y, i.e., the privilege to access object O through operation OP.

Circles represent processes and boxes stand for ports.

Figure 2: Transfer of a derived privilege: Specific privilege.

3.1 Restrictions on Privilege Transfer

From the point of view of transfer, privileges can be categorized based on whether they can be transferred to another process or not, whether they can be registered in a capability directory (i.e. can be made stable) or not, and finally, whether they are *exclusive* or not. Whether a particular privilege can be transferred or registered by a process depends on whether the process has the additional privilege to transfer or register the particular privilege, i. e., whether the capabilities representing the privilege have capcaps including the transfer or register privileges.

Port privileges, by their nature, cannot be registered and they are *exclusive* in that they can be held by only one process at a time. The exclusivity of port privileges derives from ports being designed to connect one process to exactly one other process at any time. Ports cannot be made stable (registered) since they exist only as long as they have a client or server process. A process which is either the server or client of a port can, however, transfer the server or client privilege to another process, thereby losing the privilege.

All privileges other than port privileges are *nonexclusive*, i. e. they can be held by more than one process at a time. Nonexclusive privileges are represented by operation, manager definition, cooperation class, and subdirectory capabilities. In the rest of this paper we will use operation and subdirectory capabilities to illustrate nonexclusive privileges. This is done for purposes of simplicity. Privileges represented by cooperation class and manager definition capabilities are handled by the kernel in the same way as it treats those represented by operation and subdirectory capabilities, though the user level ramifications are somewhat different.

Recall that privileges are transferred either by SEND or by SELECTRECEIVE. These two mechanisms of privilege transfer are distinguished as follows. A privilege that is transferred by a SEND primitive is not returned to the sender. A privilege that is received as part of request details of a SELECTRECEIVE primitive can be held only while the recipient is processing the request to which it pertains. When the recipient executes the SEND that satisfies the request, the kernel automatically returns the privilege to the donor, the process which executed the SELECTRECEIVE. The temporary nature of transfer through request details has several implications which will be discussed later.

The Gutenberg system allows privileges to be transferred via SEND and SELECTRECEIVE subject to the following restrictions: The operation capability for the port via which privileges are transferred must specify whether privileges are to be passed on the port, and where: in request details, message, or both. Typing the messages communicated over the port in this way simplifies the kernel's task of approving valid transfers by locating where they will occur, at the expense of decreasing flexibility in the use of an individual port by communicating processes. A second restriction is that if a privilege is passed using the SEND operation, then it must be an acknowledge-SEND. Recovery from a failed transmission before the receiver has removed the message and privileges from the port requires returning the privilege and informing the donor of the failure, and the no-acknowledge-SEND does not furnish any means for accomplishing the latter. In order to standardize the status of a port at transfer time so that the recipient of a privilege can assume its role more easily, it is required that a process may not transfer a privilege to a port if it has a pending request on that port, i.e., if the process has executed an acknowledge-SEND, a RECEIVE, a SELECTRECEIVE, or a GETDETAILS operation which has not yet been satisfied. Since privileges other than ports involve only the transfer of copies, this restriction does not apply to them.

Several other points need to be made about privilege transfer via SEND and SELECTRECEIVE. Since a port is typed such that a process may only execute one of the transfer primitives (SEND, RECEIVE or SELECTRECEIVE) on the port, there is not, in general, a way for an exclusive privilege that has been transferred by a SEND operation to be returned to the donor. As a result, a process transferring an exclusive privilege by the SEND operation must either be the original holder of

the privilege, or have received it from some other process that transferred it by a SEND operation. Although the recipient of a privilege transferred via a SELECTRECEIVE may transfer the privilege to another process if the privilege includes the transfer privilege, the requirement that an exclusive privilege be returned at the time of the SEND which satisfies the request implies that it must hold the privilege when it executes the SEND. If the privilege to be returned by a SEND is not possessed by the sender, the SEND is rejected by the kernel. Hence exclusive privileges received in request details, if they are to be further transferred, can only be transferred in request details of a SELECTRECEIVE, not as part of a message sent by a SEND. The kernel enforces this rule.

To avoid the anomalous situation wherein a port's owner is permanently unable to exercise its destroy privilege because it has irrevocably given away its client privilege, we impose the restriction that the client privilege cannot be separated from destroy privilege when either is transferred using the SEND operation. Note that this is only a restriction on the SEND primitive not on SELECTRECEIVE which is the primitive used to transfer the client privilege but not the destroy privilege. Finally, destroy privilege to a port cannot be transferred through a SELECTRECEIVE, since the exercise of the privilege by the recipient would make returning it meaningless.

The following summarizes the rules and restrictions on privilege transfer in Gutenberg:

- General

  For a process to transfer a privilege, the capabilities representing the privilege must contain the transfer capcap.

  A port on which a privilege is to be transferred must be typed to

allow the transfer; type information is contained in the operation capability used to create the port.

If port access privilege is to be transferred, the transferring process must have no pending request on the port.

If a privilege is to be made stable by a receiving process the capabilities representing the privilege must contain the register capcaps.

- Transfer of privilege using SEND

  SEND must be acknowledge-SEND.

  If privilege is exclusive, sender loses privilege permanently.

  If client privilege to a port is transferred, the owner (destroy) privilege must also be sent; the receiver becomes the owner of the port.

  If the privilege is exclusive, it cannot have been received as a part of request details i. e., by virtue of a SELECTRECEIVE.

  Transferred privilege cannot be revoked after it is RECEIVEd. (This is discussed in section 5.)

- Transfer of privilege using SELECTRECEIVE

  If the transferred privilege is exclusive, sender (executer of SELECTRECEIVE) loses the privilege temporarily; kernel assures return of privilege.

  Destroy privilege to a port may not be transferred.

  Transferred privilege can be revoked subsequently. (This is discussed in section 5.)

3.2 Privilege Transfer Mechanism

To examine how privilege transfer takes place, we now present the sequence of actions that occur during the execution of a SEND and SELECTRECEIVE. In the generic example used in the discussion, process A has a privilege X, which it passes

to process C, with which it communicates through port P1. Figure 3 depicts the situation.

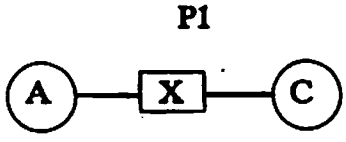


Figure 3: P1 connects A and C.

We first consider the case when A passes the privilege by including it in a message it SENDs on port P1. The normal sequence is:

1. Process A executes SEND on port P1, placing its privilege X in port P1. If X is an exclusive privilege, then the system records that process A no longer holds the privilege, marking it as held by the system.

2. Process C executes RECEIVE on port P1, and obtains A's message. The system records that C now holds privilege X. Note that if C had a pending RECEIVE on port P1, A's SEND satisfies C's request and the privilege is delivered at once to C.

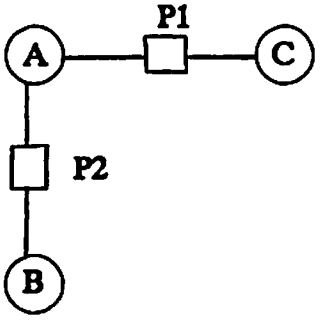
The stages of the SELECTRECEIVE transfer mechanism are:

1. Process A executes SELECTRECEIVE on port P1, placing privilege X in request details that are placed on port P1. If X is an exclusive privilege, then the system records that A has temporarily lost the privilege, and that it is system held. The system also records that privilege X is *outstanding* over port P1, that is, the privilege has been transferred over P1 and that it will have to be returned when the server of P1 executes the SEND that satisfies the SELECTRECEIVE.

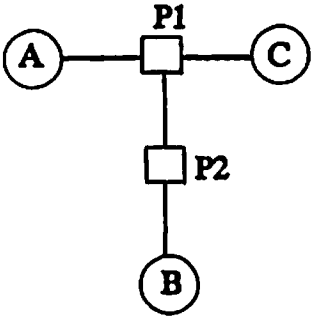
2. Process C executes GETDETAILS on port P1 and receives privilege X. The system records that C holds the privilege.

3. Process C executes SEND on port P1. The system automatically returns the outstanding privilege X to A. (Note: if process C either does not possess privilege X or in the case where X is a privilege to a port and the port has a pending request on it at the time of the SEND, the system rejects the SEND as invalid.)

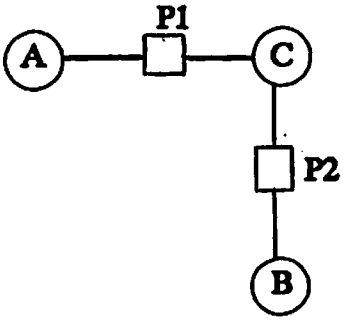
Figure 4 illustrates the case where X is a privilege for port P2 connecting A to B. The stages shown are before, during and after the transfer of the privilege to port P2 by process A to C via port P1.



(a) before the transfer: A has privilege to port P2



(b) during the transfer: the privilege to P2 is held by the system



(c) after the transfer: C has the privilege to port P2

Figure 4: Stages in the transfer of a port privilege.

4. THE REVOCATION OF TRANSFERRED PRIVILEGES

In the previous section we saw how Gutenberg permits the transfer of privileges between processes. The general characteristics of the revocation of transferred privileges in Gutenberg are the subject of this section. In the next section we present the mechanisms of revocation.

In Gutenberg, all privileges are transferred via ports either through a SEND or through a SELECTRECEIVE. In the former case, the transferred privilege is a component of the message sent, whereas in the latter it is transferred as a component of the request details. In either case, the kernel oversees privilege transfers since it is involved in the execution of port primitives. However, it is only in the case of a SELECTRECEIVE that the kernel keeps track of privileges *outstanding* on ports, i. e., privileges received in request details which must be returned when the next SEND is executed on the port. This has a few ramifications: Since privileges transferred via a SEND are irrevocable once the recipient has executed the corresponding RECEIVE, a process must transfer privileges through a SELECTRECEIVE if there is a possibility of revocation. Since only one SELECTRECEIVE can be executed on a port at a time, there can be only one outstanding set of privileges per port. However, since there can be multiple ports connecting two processes, there can be any number of outstanding privilege sets between pairs of processes.

Our revocation mechanisms are designed so that when privileges are revoked, privileges *derived from* the transferred privileges are also revoked transitively. Now we discuss some of the implications of the *revocation of derived privileges*. Suppose a process uses an operation privilege to create ports to access an object and then the

privilege is revoked. Not only is the privilege to create ports (the operation privilege) revoked, the privileges for the created ports are also revoked, causing loss of access to the objects via the ports. This additional revocation pertains only to privileges derived from the revoked privilege(s). To contrast, consider the scenario depicted in figure 4(c). Suppose B SENDs privilege Q to process C and then A revokes C's access to port P2. Here though C *obtained* Q via P2, Q is not *derived* from P2 and hence the revocation mechanism does not guarantee that Q will be returned to B. If it is desirable to return Q to B, B should employ some other means, such as transferring Q to C only through a SELECTRECEIVE, to ensure that Q will be returned to it. The above described revocation semantics was adopted so that revocation is effective while minimizing the resulting overheads. Also, it was realized that it may be impossible to undo all previous uses of a revoked privilege. For example, consider the situation where C reads object O (managed by B) via P2 and places it in one of its internal buffers to which other processes have no access and then privilege to P2 is revoked.

Revocation in Gutenberg is designed to be partial, selective, transitive, and independent **[Gligor 79, Cohen and Jefferson 76]**. It is *partial*, since revoking the privileges transferred via a port results only in the revocation of privileges transferred through that port. Privileges transferred along other ports are not affected. Privilege transfer is *selective*, since by choosing the appropriate port, the revoker can revoke from a selected process, and not from others. An effective revocation has to be *transitive*. Suppose privilege X has been transferred from process A to B. B has transferred the privilege to another process C. Revocation of privilege X by process A is said to be transitive if privilege is revoked not only from B but also from C,

the process to which B had transferred the privilege. If revocation were not transitive, it would be possible for B to obtain the privilege from C after the revocation, thus making the revocation less than effective. Revocation in Gutenberg is *independent* in that revoking the privilege transferred from one process does not affect the privileges transferred from a different process. Gutenberg does not support the automatic reversal of a revoked privilege. In this sense, revocation in Gutenberg is not *temporal*. If there is a need to reverse a revocation, the revoking process must explicitly retransfer the revoked privilege.

There is another dimension to revocation relating to transitive revocation. Consider the following scenario. Process A transfers privilege X to B via a SELECTRECEIVE on port P1. In order to complete the request of process A, process B uses process C to which it sends privileges X and Y along port P2. (See figure 5.)

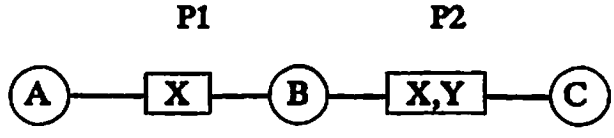


Figure 5: Transfer of multiple privileges.

Suppose A revokes the privileges transferred via P1. The revocation could proceed in two ways.

1. Privileges transferred via P2 are first revoked. This returns X and Y to B. Then X is returned to A.

2. Privilege X is removed from the c-lists of B and C and from the list of outstanding privileges of P1 and P2 and is then returned to A. Privilege Y remains with C.

The first case is termed *complete-transitive* revocation wherein every privilege transfer

that resulted from the first transfer is revoked. In the second case, which is termed *partial-transitive*, only those privileges transferred by the first process are revoked from every process involved. A revoking process, however, revokes all privileges transferred through a port. In neither case is the revoking process able to revoke a subset of the privileges transferred over a port.

Both types of revocation have applications. A complete-transitive revocation is attractive due to its recursive semantics and is applicable in situations where, for example, process B would not transfer its privileges to C but for the fact that process A had permitted its privileges to be transferred to C. A partial-transitive semantics would be useful if, for example, C could complete B's request (or part of it) even without privilege X; this would be prevented if both X and Y were revoked from C. One practical situation arises when, for efficiency reasons, B sends multiple requests to C through a single SELECTRECEIVE and hence transfers a number of privileges needed to complete each request. If partial-transitive revocation is used, even if the privileges needed for one request are revoked, other requests could be serviced. Therefore it seems important to provide for both kinds of revocation. This is made possible by typing a port to indicate whether complete-transitive or partial-transitive revocation is applicable to privileges transferred through that port.

As we shall see in the next section, providing for revocation incurs overheads; for instance, the cost of keeping track of processes using a revocable privilege. Recognizing that in many cases of privilege transfer via SELECTRECEIVE the transferring process may not have any need to ever revoke the transferred privileges, we require a transferring process to specify at the time of the transfer whether the privileges are revocable or not. While this scheme for transfer and revocation might

seem restrictive, we believe it be sufficiently flexible and powerful while minimizing overheads: overheads are incurred primarily when there is a possibility of revocation. The choice of performance over flexibility in this case was guided by the fact that capability-based systems have had performance problems and by our view that, where possible, sacrifice of performance to gain an ability should be at the discretion of the user.

5. THE REVOCATION MECHANISM

A process revokes privileges sent on port P by executing the REVOKE primitive with P as a parameter. In order to understand when a REVOKE might be executed, we have to recall the sequence of kernel actions (see section 3.2) that occur during privilege transfer via SEND and SELECTRECEIVE. Since the kernel serializes the execution of primitives on each port, REVOKE can be treated as if it occurs between kernel calls. In the case of a SEND, a REVOKE is effective only if it occurs between the sender's SEND and the receiver's RECEIVE; if the REVOKE primitive takes effect before the receiver executes a RECEIVE, then the system backs out of the transfer before executing the RECEIVE and the transferred privilege is returned to the sender. If the REVOKE is executed after the RECEIVE, the caller is notified in a status parameter that the REVOKE is impossible.

The rest of this section is devoted to the revocation of privileges transferred via the SELECTRECEIVE since it is the interesting case. There are three points at which revocation might occur: before the server's execution of GETDETAILS, after the server's execution of GETDETAILS, or after the server's SEND in response to the SELECTRECEIVE request.

First we look at the scheme designed for the revocation of exclusive privileges, specifically port privileges. Following that, we discuss the revocation of nonexclusive privileges, specifically, the privilege to create ports and the privilege to access subdirectories in the capability directory.

5.1 Revocation of Exclusive Privileges

Let us assume that process A has transferred privilege X to process C via port P1 as in Figure 3. C could also be given the privilege to further transfer X. Since port privileges are the only exclusive privileges in Gutenberg and we are only considering exclusive privileges in this section, X represents a privilege to a port, say port P2 which is connected to process B. Figure 4 depicts various stages of the transfer of X.

Before relating the details of revocation of port privileges transferred using SELECTRECEIVE, it may help to state the principles that influenced the design:

1. As a corollary to the functional addressing of the system [Vinter et al. 83], whereby a process never knows the identity of the process with which it communicates over a particular port, a privilege transfer should be as invisible as possible to processes that are not directly involved in the transfer.

2. The revocation of privileges should not be delayed by contingencies external to itself, viz., the servicing of pending requests.

One implication of the first principle is that, in Gutenberg, it is possible for a port to have several distinct servers (clients) during its lifetime unbeknownst to the client (server) of the port.

Due to the second principle, the revocation of a privilege for a SELECTRECEIVE port on which there is a pending request and outstanding privileges can be complex. The complexity derives from the possibility that the ports

to which the outstanding privileges pertain may not be in states which allow transfer. If some ports are in such states, i. e., they have pending requests on them, the system must do whatever is necessary to prepare them for return of the privileges. Outstanding privileges to SELECTRECEIVE ports which themselves have pending requests (and possibly outstanding privileges) pose special problems. The system must clear the pending request on the port to which the privilege pertains so that the donor may resume its role without difficulty, and this may involve other processes and ports. In addition, it is the responsibility of server processes to undo the modifications made on behalf of a request which cannot be completed due to the revocation of privileges that are needed to fulfill the request.

We again consider the case illustrated in figure 4, A transfers privilege for port P2 to process C via P1, and A executes REVOKE(P1), i.e., A desires to revoke the privilege for P2. There are two points at which revocation can occur when a privilege is transferred via a SELECTRECEIVE. REVOKE could be executed before C's GETDETAILS (in response to A's SELECTRECEIVE) or after. If it is executed before, then the system returns the transferred privilege to A. Otherwise, there are three possibilities as discussed below.

C holds the privilege to port P2 and C has no pending request on P2: The port is removed from C's p-list, mention of P2 is removed from the outstanding privilege list of port P1, and P2 is returned to A. The request on P1 is removed.

C holds P2 but has a pending request on P2: Assume P2 connects processes C and B as portrayed in figure 4(c). Process C's pending request on port P2 might be a RECEIVE, a GETDETAILS, an acknowledge-SEND, or a SELECTRECEIVE. In all but the last, the system clears C's request immediately by removing all record of

it from port P2, and returning any port privileges that C may have placed on port P2.

The same action is taken if C's pending request is a SELECTRECEIVE, and B has not removed the request details: The system backs out of C's request. However, if the request is a SELECTRECEIVE and B has taken the request details, then the system has no way of instructing B to abandon tending to C's request, so it ensures that the reply to the current request is ignored. After first revoking any privileges that C passed in the request details, the system sets a flag in port P2 that is used by the kernel to recognize the reply to a dead request when the SEND operation is executed on P2. When B executes the SEND on port P2, the kernel does not place a message on the port.

If process C has received request details of a SELECTRECEIVE executed by process B on P2, any privileges C received in the request details must be returned to B, and the system must execute a REFUSE on port P2 that tells B its request failed. Thus, whether it is C's SELECTRECEIVE or B's SELECTRECEIVE that is pending on port P2, eventually P2 has no pending requests, at which point the situation is similar to the previous case.

C has transferred port P2 to another process on port P3: For revocation to be carried out, there is a need to know which process holds the privilege. The p-lists of processes are utilized for this purpose. (Recall that the p-list of a process is a list of all ports to which the process currently has a privilege.) Also maintained for each port is a servers' stack (a clients' stack) containing the identities of the processes that had server (client) privilege to the port, and the ports along which the processes subsequently transferred the privilege.

The actions taken by the system depend on whether P1 is typed so that revocation of privileges transferred on it is complete-transitive or partial-transitive. If it is the former, system takes actions corresponding to REVOKE(P3). Otherwise, it does the following: after removing P2 from the p-list of the process which holds P2 at the time of revocation, it deletes P2 from the list of outstanding privileges of every port along which P2 was transferred. Once these actions are completed, C holds P2. This situation is handled as in the previous cases.

The form of repossession described here may interfere with the processing of the SELECTRECEIVE requests that included the privilege to port P2 as request details, but it falls upon the processes involved, not the system, to determine whether the requests are now unsatisfiable and to take the appropriate action.

5.2 Revocation of Nonexclusive Privileges

Again, for the sake of concreteness, let us assume that process A has transferred privilege X to process C via port P1, as in Figure 3. C could also be given the privilege to transfer X further as well as the privilege to register the privilege in some subdirectory. Since we are only considering nonexclusive privileges in this section, X cannot be a port privilege.

Suppose A executes REVOKE(P1) after transferring X. X could be the capability for accessing a specific subdirectory in the capability directory or an operation capability giving the privilege to create a port to access an object of a given type via an operation.

To revoke transient privilege X, the c-lists of processes are utilized. In addition, capabilities which are outstanding on ports are linked to the ports on which they were transferred. Since revocation in Gutenberg is transitive, when a process's privilege is revoked it is necessary that the capabilities representing the privilege be deleted not only from the c-list of that process but also from the c-list of every process to which the privilege was transferred.

Since revocation in Gutenberg is designed to revoke privileges derived from a revoked privilege, revocation of privileges by removing capabilities from c-lists of processes is insufficient. Revocation of an operation privilege should not only remove the privilege from the process's c-list but also destroy any port created using the privilege.

When a privilege to access a subdirectory is revoked, it is necessary to revoke all privileges that a process obtained from that subdirectory. To reduce the complexities of such a revocation, we require that when a revocable privilege to a subdirectory is transferred, the privilege should be restricted such that the receiving process does not have access to the subdirectories registered in the transferred subdirectory. Thus, transferring a subdirectory privilege is equivalent to transferring the privileges to access all nodes, except subdirectory nodes, within that subdirectory. Of course, if there is a need to give access to an *inner* subdirectory, the privilege to do so can be transferred separately.

Once a process has used its privilege to change to a subdirectory, revoking that privilege should also result in changing the process's active directory. The question arises as to which directory should be made the new active directory. In Gutenberg, associated with every manager capability is a prime directory; when a process is

instantiated using its capability, the prime directory associated with that capability becomes the active directory of the process. We believe that when access to a process's active directory is revoked, making the process's prime directory the new active directory results in a simple yet workable solution.

When Gutenberg removes a process's ability to access a subdirectory, it does so without undoing any changes the process may have made to the subdirectory. It is for the transferring process to restrict the recipient process's use of the subdirectory. For instance, the transferring process can transfer the subdirectory privilege without the rights necessary to modify it.

In section 5.2.1 we discuss the mechanisms for the revocation of transient nonexclusive privileges. Additional features necessary to handle the revocation of transient nonexclusive privileges that can be made stable by being registered in a subdirectory are described in section 5.2.2.

5.2.1 Revocation of Nonexclusive Transient Privileges

We now consider the revocation of privilege X transferred to C via a SELECTRECEIVE. REVOKE could be executed before C's GETDETAILS (in response to the SELECTRECEIVE), after C's GETDETAILS, or after C's SEND.

Revocation after C's SEND is relevant only for privileges that have been made stable and is discussed in the next section. If revocation is attempted before C's GETDETAILS, then the system returns the transferred privilege to A. The actions taken by the system when revocation is done after C's GETDETAILS but before C's SEND are better described in terms of the following pseudo-code.

```
{A process has transferred nonexclusive privilege X to process C
 on port P1 and has executed a REVOKE(P1).}

begin
 Let Y stand for privilege X as well as any privilege
   transitively derived by C from X;
 For each Y
 DO
 If Y is outstanding from any process to which it was transferred
  then { Suppose  C  has transferred Y via port P3;
         If P1 is typed so that revocation of privileges
            transferred along P1 is complete-transitive
             then REVOKE(P3)
             else {Remove Y
                      from any process to which it was transferred;
                   Delete Y
                     from the list of outstanding privileges
                     of every port along which it was transferred}}
 END;
 Remove X from C's c-list;
 Remove X  from the outstanding privilege list of port P1;
 Return X to A
end;
```

5.2.2 Revocation of Nonexclusive Stable Privileges

Recall that all transferred capabilities are initially transient, i.e., they are represented as capabilities in the recipient process's c-list. If the recipient process has the privilege to do so, then the transient capability may be registered in its active directory. The registered capability then becomes obtainable by other processes in two ways:

1. other processes that can make the subdirectory their active directory can obtain the privilege.

2. a process with the subdirectory as its active directory may create a transient form of the stable privilege and transfer the privilege to another process.

When a privilege is revoked, privileges obtained from its stable form must also be revoked. Though the system ensures that all transient privileges are automatically returned when the SELECTRECEIVE is completed this is not true for registered

privileges. Such privileges have to be explicitly revoked, i. e., removed from their subdirectories.

As we shall see, revocation of stable privileges requires complex mechanisms which incur substantial overheads. One way to avoid this is to prohibit the transfer of privileges along with the privilege to register them. This would mean that all transferred privileges would remain transient: Their revocation could then be handled using the scheme described in section 5.2.1. However, in Gutenberg we have provided for the transfer of revocable privileges that can be registered. The following mechanisms accomplish such a revocation. Experience will show whether the benefits of this mechanism outweigh its cost.

The first set of mechanisms is used to remove subdirectory entries corresponding to transferred privileges that have been registered. A revocable capability in a recipient process's c-list is linked to any subdirectory in which it has been registered. This is used to remove the subdirectory entry if the privilege is revoked before the recipient process (the server of the port on which the privilege was transferred) executes a SEND in response to the SELECTRECEIVE.

However, it should also be possible for a transferring process to revoke transferred privileges that have been registered by a recipient process after the process has executed its SEND. This revocation translates to the removal of entries from the subdirectories in which the privileges have been registered. (This is because the transient capabilities have already been revoked by the system and are no longer in the recipient process's c-list.) The revoking process needs a privilege in order to perform this removal. To facilitate this, when the server executes the SEND, the system also returns a list of subdirectory entries corresponding to the transient

privileges that have been registered. Also returned are privileges to REMOVE these entries from the appropriate subdirectories. (These entries include those created by the SENDing process as well as those created by the processes to which the privileges had been further transferred.)

In order to limit the registration of revocable privileges to those for which the donor has REMOVE rights, a process obtaining its privilege from a revocable subdirectory entry is not permitted to register it or to transfer it with the privilege to register. A process can, however, transfer without the register privilege. This does not preclude the original holder(s) of the transferred (and duplicated) privilege from registering it in different subdirectories if they have the privilege to do so.

A second mechanism is used to revoke privileges from processes that use a revocable subdirectory entry. Once a privilege has been registered in a subdirectory, any process with that subdirectory as its active directory can use that privilege with the restriction just outlined. If registered privileges are revoked, privileges have to be revoked from the processes that used the entry in the subdirectory. To facilitate this, attached to a revocable entry in a subdirectory is a list of (active) processes which used the privilege. If subsequently the subdirectory entry is revoked, then the corresponding privilege is revoked from each of these processes.

A third mechanism is used to revoke privileges from each of the processes to which the privilege corresponding to a revocable subdirectory entry is transferred. Before the privilege in a subdirectory entry is transferred, the transferring process creates a transient form of the privilege which causes an entry to be created in the process's c-list. The entry in the subdirectory points to the entry created in the process's c-list. The needed revocation is then made feasible since the c-list entry for

a privilege that has been transferred identifies the port along which the privilege is transferred.

6. HANDLING EVENTS THAT DISRUPT PRIVILEGE TRANSFER

The events that may disrupt the transfer of privileges are:

1. The privilege is revoked by the transferring process.
2. The port on which the privilege is passed is destroyed.
3. The transferring process terminates.
4. The receiving process terminates.
5. The transferred privilege is a port privilege and the port is destroyed.

One of the requirements on our privilege transfer mechanism is to respond to any of these disruptions in a manner which does not create "lost" objects, i. e., user-defined or kernel objects to which no access is possible. A second requirement is that the appropriate notification of the disruption be given to the processes affected by the disruption. In this section we discuss the Gutenberg responses to these disruptions.

6.1 Disruption during a SEND

The system's response to any of the disrupting events is the same: the transferring process A is informed of the failure of the transmission, the privilege is returned to A, and the privilege revocation, process termination, or port destruction that caused the failure is processed normally. In effect, the system backs out of the transfer before performing the revocation, termination or destruction; the requirement that process A's SEND be an acknowledge-SEND makes it possible to notify A that

the privilege has been returned.

6.2 Disruption during a SELECTRECEIVE

In section 5 we saw how the system handles revocation during privilege transfer through a SELECTRECEIVE. Destruction of the port on which the privilege is transferred, say P1, is handled by first performing the actions of a REVOKE(P1). This returns the port to the state that existed before the SELECTRECEIVE was executed. Then the system takes actions for destroying the port.

Termination of the transferring process is handled by requiring that a process which has a pending SELECTRECEIVE on a port should first revoke privileges transferred via the port and then execute the terminate. If it does not, the kernel acts as if it did, as in the previous case. (Whether or not to revoke stable privileges could be a parameter at TERMINATE time).

We now examine the system responses to the last two disruptive events. As before, assume that A transfers privilege X via port P1 to process C.

The receiving process terminates. Suppose C, the receiving process terminates before its GETDETAILS. If C is the original server of port P1, then the system informs A of C's termination (in the status returned for the SELECTRECEIVE), returns privilege X and destroys port P1. Otherwise, C received server privilege to port P1 from another process in request details of that process's SELECTRECEIVE. The server privilege to port P1 should be returned to the donor. The donor may then be able to field A's request.

Suppose C terminates after its GETDETAILS. If X is the privilege to port P2, then P2 has to be restored to a consistent state before it can be returned. If port P2 is in a state where it can be returned then the system executes a REFUSE that returns the privilege and informs A that the request failed. Otherwise the system restores port P2 to its pristine state before it is returned to A.

When a process terminates, the system creates a table of all the privileges that will be returned to it (due to pending SELECTRECEIVEs), and records whether they must be returned to other processes. When a privilege is returned to the terminated process, if it did not receive the privilege in request details, the port to which the privilege pertains is processed as if the call to TERMINATE was made at that point. Otherwise, it is grouped in a list with other privileges currently held by the terminated process that were part of the same request details. These lists are maintained for each SELECTRECEIVE port which the terminated process was serving, and when the contents of a particular list matches the outstanding privilege list of the SELECTRECEIVE port to which it corresponds, the system executes a REFUSE to return the outstanding privileges and inform the client process that its SELECTRECEIVE request has failed.

The transferred port is destroyed. If P2, the transferred port, is destroyed before C's GETDETAILS, (P2's server privilege must be the privilege transferred in this case), the system rejects A's request on port P1, returns the privilege to A, and effects the port destruction normally. If however, port P2 is destroyed after C's GETDETAILS then the port is destroyed and the privilege to port P2 is removed from the list of outstanding privileges on port P1, so that the system will not try to return the privilege when C performs its SEND. Note that it is up to C to

determine if A's request can be satisfied. C will get a status returned informing it that P2 no longer exists if it executes a primitive on P2. If process A had received the server privilege to port P2 from some other process, the system must trace it back to its source and delete it from the outstanding privileges lists of all the SELECTRECEIVE ports over which it was passed.

7. CONCLUDING REMARKS

We have presented a scheme for transferring and revoking privileges in a port-based system, the Gutenberg operating system. Gutenberg is a port-based, object-oriented kernel in which protection is treated nonuniformly: The protection of objects local to processes is left to techniques outside the kernel, e. g., compilation of strongly typed languages, while the kernel protects objects that are manipulated through interprocess communication. The basic protection principle in Gutenberg is: *A port connecting a process to the manager of an object exists only if the process has the privilege to access the object with a specific operation.* Thus, privileges to create and use ports are privileges to operate on user-defined objects; this, along with an abstract data type approach to all kernel objects, gives Gutenberg its object orientation.

Privilege in Gutenberg is represented by capabilities which can have two levels of persistence, transient for those capabilities which persist only as long as an owning process exists, and stable for those capabilities in a directory structure, the capability directory, whose existence is independent of processes. We have presented mechanisms for achieving transfer and revocation of privileges represented by both transient and stable capabilities. We have highlighted the choices made in the design

of these mechanisms and explained those choices in terms of the tradeoffs among simplicity, flexibility, and efficiency. The choices include:

- Using both unidirectional and bidirectional communication primitives and associating them with permanent (unidirectional) and temporary (bidirectional) granting of privileges in order to provide flexibility in privilege granting while keeping the kernel simple.

- Typing ports with respect to the ability to transfer privileges on them in order to expedite communication in cases where no privilege transfer can be made.

- Restricting ports to connecting one client process with one server process in order to simplify interprocess communication in general and the transfer of privileges in particular.

- Separating capabilities into transient and stable capabilities and allowing stable capabilities to be stored only in a kernel-managed directory in order to permit privileges to exist beyond the lifetimes of processes without overly complicating the management of privileges.

- Requiring the possibility of revocation of a transferred privilege to be declared at transfer time in order to limit the overhead incurred during the transfer of privileges that will not be revoked.

- Including in the revocation mechanism the ability to revoke derived privileges and privileges which have been made stable, so that users can be allowed a wide use of transferred privileges even if there is a possibility of revocation.

Experience in the use of Gutenberg will test these choices. We are currently implementing a centralized version of Gutenberg and are studying its extension to a distributed kernel with an emphasis on the protection issues.

8. REFERENCES

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

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

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

[Gligor 79] Gligor, V., "Review and Revocation of Access Privileges Distributed Through Capabilities," IEEE Transactions on Software Engineering, vol. SE-5, no. 6, November, 1979.

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

[Lampson and Sturgis 76] Lampson, B. W. and Sturgis, H. E., "Reflections on an Operating System Design," Communications of the ACM, vol. 19, no. 5, May, 1976.

[Needham and Walker 77] Needham, R. M. and Walker, R. D. H., "The Cambridge CAP Computer and its Protection System," Proceedings of the 6th ACM Symposium on Operating System Principles, November, 1977.

[Ramamritham et al. 83] 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.

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

[Stemple et al. 83] Stemple, D., Ramamritham, K., and Vinter, S., "Operating System Support for Abstract Database Types," Proceedings, 2nd International Conference on Databases, September, 1983.

[Stemple et al. 84a] Stemple, D., Ramamritham, K., and Vinter, S., "Interprocess Communication Without Process Identification," Submitted for publication, 1984.

[Stemple et al. 84b] Stemple, D., Vinter, S., and Ramamritham, K., "Dynamic Control of Module Interconnections", submitted for publication, 1984.

[Vinter et al. 83] Vinter, S., Ramamritham, K., and Stemple, D. "Protecting Objects through the use of Ports," Proc. of the Phoenix Conference on Computers and Communication, Mar 1983.

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