Capability Managers

RICHARD B. KIEBURTZ, MEMBER, IEEE, AND ABRAHAM SILBERSCHATZ

Abstract—The use of capabilities to control the access of component programs to resources in an operating system is an attractive means by which to provide a uniform protection mechanism. In this paper, a capability is defined as an abstract encapsulation of the data needed to define access to a protected object. We do not assume that capability checking is necessarily concentrated in a protection kernel, nor that capabilities to different types of objects are all of the same degree of complexity. We explore a language-based capability mechanism in which protection environments are established by declaration, enforcement protocols are automatically produced by a compiler, and access control policy is clearly placed in the hands of the system designer. The basic mechanism introduced is a program component called a capability manager that is an extension of the monitor concept. It can be used to realize most of the facilities associated with kernel-based capabilities, including preemptive revocation.

Index Terms—Access control, capability, exception handling, manager, monitor, protection, resource allocation, revocation.

I. INTRODUCTION

The invention of the monitor concept [1], [2] and its inclusion in the programming language Concurrent Pascal [3] illustrate the use of a standardized language-based mechanism to achieve synchronization of processes accessing a shared data base, to encapsulate abstract resource types, and to provide more flexible, dynamic access control. An obvious
advantage of a language-based control mechanism is that data-
independent restrictions can be stated declaratively, and their
enforcement can be guaranteed by a compiler. Validity
checks that depend only on static information known at com-
pile time need not incur overhead during execution. An
equally important, but perhaps less obvious, benefit is that
declaratively stated access control affords greater opportunity
for verification that implementation matches intent than does
a control mechanism which depends on a code sequence
explicitly programmed by a system implementor.

On the other hand, the greatest flexibility in an access con-
control mechanism is achieved by providing interpretation and
validation of access requests at the time of execution. This
strategy, under which control decisions are deferred until they
can no longer be avoided, is best illustrated in the notion of a
capability subsystem [4], [5]. Obviously, some price must be
paid for such flexibility, both in terms of machine overhead
during execution and in the conceptual and logical complexity
of a system that utilizes it.

It is the aim of this paper to show that language-based access
control can be extended, virtually as far as may be desired, in
the direction of greater flexibility. The motivation for this ex-
ercise is to enjoy the benefits of a standardized enforcement
mechanism wherever it is adequate, while retaining the ability
to use data-dependent, procedurally interpreted control when
its power and generality are needed. In a language-based sys-

tem, the programmer can specify the quality and degree of

protection that he wishes for each abstract object that a pro-
gram can manipulate.

Use of a programming language mechanism to assure the
integrity of a data segment that is shared among concurrently
active processes rests upon two assumptions.

1) Compiled code will be executed on a virtual machine
which provides inviolable storage segments. This protection
is needed to assure that the contents of a code or data segment
cannot be modified by the activity of an unauthorized process.

This protection is customarily provided by an operating sys-
tem nucleus, utilizing protection keys, relocation registers,

words containing tag bits, or some equivalently effective
means provided by a lower level virtual machine.

2) Each request by a program to alter the scope of storage
segments that it can address must have been validated by a
language processor, such as a compiler, to ensure that the
agreed-upon access control protocols are observed. This
restriction is to assure that unchecked code cannot bypass the
declared intent of access controls, even when the use of an
access right is not validated by an intervening agent.

Note that the second assumption does not require all code in
a system to be checked by a common compiler. A process
executing unchecked code can be allowed, but any request it
makes to extend its access rights would have to be intercepted
and validated by an interpretive mechanism.

What we are dealing with here are the rights of program com-
ponents to access resources. More specifically, we view a
resource as any abstraction that is manifested in a computer
system by data in a storage segment. What, then, constitutes
an access right? First, there must be knowledge of its exis-
tence, which in a source-language program, means an identifier
having a set of declared attributes for a resource. Second,
there must be a binding to a storage segment containing the
data base of an instance of the declared resource type. We call
this an access binding; it may be either static, that is, estab-
lished permanently at the time of program generation, or

dynamic.

A capability is a logical extension of the notion of access
binding to allow the inclusion of additional data that may be
used to govern access control. These data may include a set of
rights, a validation key, and attributes that identify the pro-
gram component holding the capability.

A right consists of a license to perform a designated opera-
tion on a resource. Rights are introduced into capabilities in
order to differentiate among the actions that can be performed
by different processes upon a shared resource.

A key is a unique number supplied when a capability is
granted. The simplest use of a key attaches it to an instance of
a resource, and subsequently places a copy of the key into
each capability created for use of the resource. Each access
request is then validated by matching the key. Keys are intro-
duced in a system design when one wants to retain the ability
to revoke a capability previously granted.

Identification in some form or other is embedded into cap-
abilities when discrimination among different holders of access
rights to a common resource is made a part of an access con-

trol strategy. One way to implement this identification is by
assignment of a unique key to each newly created capability
[6]. Capabilities may then be transferred among processes
without losing their identities. However, a resource must then
have associated with it not just a single key, but a collection
of all keys allocated to capabilities for its use.

Motivation for the use of capabilities in a system design
arises from the need to implement policies that discriminate
among different users of a shared resource. The dominant
point of view in most of the published work on capability-
based systems has been to introduce the capability concept at
a very primitive level, so that it pervades the entire system
[7]–[9]. This has considerable esthetic appeal as a design
philosophy, but is not inherent in the abstract concept.
Furthermore, the requirements for protection are not uniform
among all system components. Considerable sophistication is
necessary for the implementation of policies of protection on
high-level objects, such as files, whereas much simpler proto-
colls will ordinarily suffice to protect simpler objects, such as
untyped segments in real (physical) main storage. A central-
ized uniform capability system can either provide efficient
protection of objects whose access protocols are simple, or
sophisticated, but expensive protection of all objects, but
usually cannot serve both needs equally well.

In what follows, we develop a very flexible system for defin-
ing capability-based protection in a programming language
environment. This seems to be suitable to meet the protection
needs of all types of objects, save possibly that of untyped
segments of main storage. These we assume to be defined and
access-restricted by a simple operating-system nucleus, utilizing
such hardware support as is available on any given computer.

It is at the point of definition of various customer processes,
and the provision of general forms of resources for them to
use, that the capability concept achieves its greatest utility. We believe that it is at roughly this same conceptual level of abstraction that a software designer will wish to capitalize on the expressive power of a high-level systems programming language to describe the processes, resources, and interaction that he wishes to implement. An alternative to the manipulation of capabilities by a nuclear protection kernel is to introduce capabilities of varying degrees of sophistication into a system design just at the points where they are needed to implement policy. We propose that this can be done most readily by defining the capability concept in a programming language environment [10], [12], [19].

A dynamically bound capability requires data storage in the address space of the program component that holds it, and is therefore itself a form of resource whose integrity must be protected. In order to secure this protection, each capability may be enclosed in an individual storage segment.

In an operating system designed around a protection kernel, only the kernel holds the right of access to the contents of capability segments. Prevention of forgery of capabilities by program components outside the kernel then rests solely on assumption 1) about the inviolability of storage segments. In a language-based implementation of capabilities, the validity of both assumptions 1) and 2) removes the stringent requirement of a nuclear protection kernel to guarantee the integrity of capabilities, and allows its functions to be distributed throughout a system, and even to be intermixed with policy-decision functions. We have named the modules that carry out the allocation of capabilities, capability managers, and in the remainder of this paper, we shall attempt to explain how they work.

II. Capability Managers

In order to protect the integrity of an abstract data type, a data segment that represents an instance of the type must not be subject to direct access by programs that wish to make use of it. Access is instead accomplished by means of a set of procedures and functions defined for the type, as in an access-restricted Simula class, or in the monitor of Concurrent Pascal. For such a resource, the set of rights for its use is taken from the set of exportable procedure and function names defined in the program component implementing operators of the type. The set of rights held by a program component for the use of an instance of an abstract type can be contained in the capability that it holds. These rights sets will play a central role in describing a language-based implementation of capabilities. When a resource type is defined, sets of its exportable rights defining useful views of the resource may be declared. Both the names of the declared sets and the names of individual rights listed as members of these sets are exported from the definition of a resource type. For instance, a resource type might be declared as

```
type Channeltype = monitor;
  rights Receive = {Get};
    Send = {Put};
var

procedure Get ( );

endmonitor.
```

In a declaration of an instance of the type, the maximum set of rights assumed for that instance is declared as a qualifier of the type name. It is not necessary that all declared instances of the type be given the same maximum set of rights; these may be restricted in order to declare instances with different views of the type, as in

```
var Controlchannel : Channeltype (Send);
  Communicationchannel : Channeltype (Send + Receive);
  (* in which the operation + denotes the union of sets *)
```

A program component in which an instance of a resource is declared obtains a static capability for use of the instance. This means that the compiler can make an access binding statically, as well as ensure that the program component attempts to exercise only those rights claimed in the declaration. A resource can be passed as a var parameter to a procedure. The formal parameter list of the procedure must also specify the set of rights assumed for use of the resource within the procedure body. These rights must not exceed those declared for the actual parameter in a call. Again, this restriction can be checked by a compiler.

In an earlier paper [11], we introduced the notion of a program component type called a manager, which is similar to a monitor in structure, but which dynamically allocates to customer processes the access binding to a resource. A process can access a managed instance of a resource only if it holds an access binding previously granted to it by the manager. Whereas a monitor encapsulates an abstract resource, prevents access conflicts by processes that each hold an access binding, and provides synchronization for the actions of processes in their use of the resource, a manager encapsulates a pool of identical interchangeable instances of an abstract resource. With the additional facility of declaring sets of rights associated with an access binding, the manager becomes a distributor of capabilities. The capability allocation and revocation functions usually associated with protection are to be distributed among capability managers in a language-based implementation.

The address space of a manager will contain one or more statically bound instances of the resource type that it manages. To the set of rights defined for the managed type, the manager defines some new rights for manipulation of capabilities to the type. At the least, these must include a right to call for allocation of a capability, for otherwise the manager would have no facility for transmitting instances of the managed type from its own address space to those of customer processes. In addition, it may add other rights to permit voluntary deallocation, to accomplish preemptive revocation of capabilities previously granted, or to otherwise enhance the set of rights originally defined upon the resource type. We call these additions by the manager, enhanced rights of the managed type.

For instance, suppose IPCtype is the name of a manager
type for the resource Channeltype mentioned previously. We have in mind the use of IPCtype to define a pool of interprocess communication buffers. The sets of rights that will apply to instances of the managed type will be those defined within the manager.

```pascal
type IPCtype = manager of IPC : Channeltype;
    rights Receiver = Receive + {Open, Close, Status};
        Sender = Send + {Open, Close, Status};
    const N = 50;
    var Channel : array [1..N] of record
    ID_number : integer;
        EHC : integer;
        (* an entry hold count *)
        Instance : Channeltype
    end;
procedure entry Open (Rendezvous_number : integer);
    (* allocates an access binding to an instance of
    Channeltype with ID_number equal to
    Rendezvous_number *)
procedure entry Close;
    (* deallocates an access binding to Channeltype *)
function entry Status : Boolean;
    (* returns true if an access binding is currently held
    in the capability of the customer process *)
endmanager.
```

An enhanced set of rights may contain rights not held by the manager itself, but provided for the use of customer processes. Conversely, the manager may hold rights to the managed type that are not incorporated into any of the exportable sets of rights. A customer process must declare the rights that it assumes in terms of the enhanced sets of rights defined by the manager, not those defined directly for the resource type. Thus, the abstraction defined by a manager can be protected, even though it is implemented in terms of a more primitive abstract type not hidden from the customer process.

For instance, a customer process, having a globally declared access to a manager,

```pascal
var IPCpool : IPCtype;
```

and locally declaring an instance of the managed resource type,

```pascal
var IPCinput : Channeltype (Receiver) from IPCpool;
```

now holds, under the name IPCinput, a capability to access both the manager and an instance of the resource. Initially, of course, the capability to the resource has no access binding; in order to secure that, the allocation procedure Open, defined by the manager, must first be executed.

Fig. 1 illustrates a capability graph in which the nodes represent program components, arcs represent access binding, and labels on the arcs represent rights. A dashed arc indicates a dynamic access binding, established by invoking the allocation procedure Open. In the diagram, Process 1 might be the customer process whose declarations are given above, and who has made a resource allocation call

```pascal
IPCinput.Open (<number>);
```

Note that the manager in this example holds only an access binding to the resource instances that it manages; it holds no rights to operate upon them.

Up to this point, we have purposely avoided describing a capability in terms of its representation in a data segment. Although the definition of a capability should not mandate a particular implementation, we can suggest a possible one. The data base of a capability can be described as a sequence of records. Each record, save the last, corresponds to a level of management, while the last record defines access to an in-
stance of the managed type itself. The contents of each record consist of an access binding, a set of rights, and any keys or other data needed to identify either the resource itself or the individual capability. Fig. 2 illustrates the representation of a revocable capability with one level of management.

The first record of a capability will always contain static access binding. Since static allocations are made during compilation, it is permissible for the data represented by this record to be distributed among the code and the data segments normally used to establish addressability, and for this reason, the record has been shown in dashed outline in Fig. 2. Records following the first one represent dynamic access bindings, and must be present in a distinct capability segment. When a capability segment is created, its sets of rights can be initialized in each component record, for these never change. Its dynamic access bindings are initialized to a null address, and the contents of its key fields are undefined.

Capabilities can obviously be of differing sizes and complexities, depending on the degree of control defined over the abstract resources to which they afford access. This variability has the advantage that the overhead imposed on the use of a capability is proportional to the sophistication of the resource; simple resource management does not incur overhead for the sake of allowing generality. In the exercise of a capability, no more than a single level of indirection is ever required to invoke a procedure. Since all access bindings currently held are present in the capability at any time, a call upon a procedure may be implemented by a compiler in the following way.

1) From the identifier of the procedure mentioned in the call, determine the type to which it belongs. From the declarations of the capability, determine which record in the capability segment controls access to a resource of this type. Further references to the capability mean references to this specific record.

2) Check that the name of the requested procedure is contained in the rights field of the capability.

Compile code for a calling sequence that will:

3) Use the access binding from the capability to address the data segment of the resource. Provide an exception trap, or the return of error status in case an access binding is not present in the capability.

4) Validate the access by matching the key from the capability against the validation key found in the storage segment. In case of failure, take an exception return as in 3).

5) Perform the procedure call.

In the case of static capabilities, steps 3) and 4) can be performed at compilation time, unless capability revocation has been declared possible.

It should now be clear why a common name has been used as a syntactic reference, both to an instance of a managed resource and to the manager of the resource. All rights are invoked through the use of this common identifier, indicating clearly that the enhancement provided by a manager alters a user's view of an abstract resource type.

III. OWNERSHIP AND REVOCATION OF ACCESS

Ownership in a capability-based system is the concept that distinguishes one process as having the right to establish a policy for distributing capabilities to use a particular resource. In a system using capability managers, the manager is the only program component which has the ability to allocate or modify a capability itself. If a doctrine of private ownership is to be implemented, then the manager must act as an agent in carrying out the policy of an owner. In the absence of such a doctrine, a kind of communal ownership of all resources ensues by default, in which policies governing the distribution of capabilities are those set by the system designer.

If a policy for distribution is to discriminate among those who request capabilities, then there must be some basis for distinguishing among the requestors. We can identify three bases for discrimination.

1) Capacity of a resource. There are many types of resources for which it is appropriate to restrict the number of processes that can simultaneously hold privileges. For the class types of Concurrent Pascal, the capacity limit is one. For an interprocess communications buffer, a reasonable limit might be two. Other capacity limits may be application dependent. A resource capacity may be implemented by embedding an entry-hold-count limit in a capability [11].

2) Resource naming. Two or more processes that need to share a common instance of a managed resource may agree among themselves on an identifier to be used in requesting the resource. This identifier may be subject to inspection and modification by a process that obtains it, and is not made part of a capability. Nevertheless, a manager can implement a policy of inspecting the names submitted by processes to determine whether to allocate access to a resource instance already accessible to another process. An example is the use of rendezvous numbers by processes that wish to exchange messages through an interprocess communications buffer [11].

3) Mandatory process identification. As a basis for discrimination, every process in the system can be given a unique identifier which is embedded as a constant in every capability held by the process. A manager could compare and copy such a constant, although it could not alter it. Protection is provided by the definition of a constant in the programming language. Discrimination can then be made on the basis of attributes associated with the process identifiers in a table maintained by a manager, analogous to the use of access control lists in Multics [13].

Ownership is a policy-driven concept, and it need not be
inherent in the design of a system. Its implementation through capability managers affords a means of limiting the scope of any particular policy without affecting other parts of a system. Close on the heels of a policy-driven mechanism for the distribution of capabilities comes the need to provide for revocation of rights previously granted. The simplest instance of revocation occurs when the owner of a resource instance executes a terminal deallocation of the resource. For instance, this could be the result of a timeout signaling that a process had exceeded its authorization to use the resource.

Since a capability resides in the address space of the process that holds it, there is no way for a manager to regain it preemptively. When a process exercises a right to operate directly upon a resource instance that it has been allocated, the manager is not involved. A manager would only have an opportunity to regain control of a capability when the process invokes one of the enhanced rights defined by the manager itself, and the capability segment is temporarily bound to the manager's address space. Therefore, preemptive revocation of a capability must occur by disabling it, rather than destroying it.

The use of a validation key is the means we suggest for providing revocability of capabilities. A master copy of the key is kept in the storage segment of a managed resource to which the manager has access. A duplicate copy of the key is inserted into a capability, along with an access binding, when a user is granted access by an allocation procedure of the manager. The manager holds an implicit right Setkey to set or reset the master key of each instance of the resource type that it manages, and this right is never exported. When the policy governing use of a resource instance calls for revocation, the manager executes its modification privilege on the master key, thereby disabling all revocable capabilities for use of that instance.

The attribute of being revocable must be declared for each resource type to which it is to apply in order that the compiler will have knowledge of the fact. It is only for revocable capabilities that the validation key needs to be checked in the calling sequence compiled to execute a right. Otherwise, the overhead of capability validation can be avoided.

Lest revocation should sound simpler than it really is, let us point out two difficulties.

1) Revocation of a capability for the use of a class type resource cannot be immediately effective. This is because there is no way to determine whether or not the process holding a capability is or is not exercising it (i.e., is active in the program segment of the class). Therefore, although a revocation action can prevent a process from initiating another operation upon the revoked class resource, the data segment of that resource instance cannot safely be reallocated until the process holding the revoked capability has voluntarily or involuntarily signaled the manager by some action.

2) When a capability for use of a monitor containing condition queues is revoked, deadlock possibilities may arise that were impossible prior to the revocation. Obviously, if a process is suspended on a condition queue, awaiting a signal from another process whose capability to issue the signal has been revoked, the waiting process waits in vain.

Difficulty 1) can be avoided if the resource to be revoked has the mutual exclusion protection of a monitor. Then the right of revocation to be exercised by the manager will be realized by a monitor procedure which cannot be completed while another process is active in the monitor. Revocation becomes effective at the first opportunity at which the resource is in a quiescent state. Since management of classes is also important, we suggest another means of dealing with this problem in Section V. Difficulty 2) is not so easily avoided. We shall return to it in the following section.

IV. Exception Handling

The possibility of preemptive revocation of a capability held by a process increases the number of ways in which exceptional conditions can arise during the activity of the process. Although the primary intent of revocation is to interrupt the course of a runaway process that has exceeded its authorized use of a resource, it will inevitably also affect processes that are not runaways, and which must somehow attempt to recover from the withdrawal of assumed rights.

There are three principal strategies for dealing with exceptional conditions.

1) Handle the exception at the point of detection. When a condition arises that prevents continuation of process execution, place the process on a condition queue to await a change in the state that led to the exception.

2) Return control to the point of invocation of the operation in which the exception was detected. To indicate the exception, a contingency code is also returned in a status variable. It is the responsibility of the calling process to check the returned status and take appropriate action.

3) Transfer control to a trap location designated by the program component that invoked the failed operation.

The first strategy is only satisfactory for temporary conditions. In fact, when preemptive revocation of capabilities is possible, there is no sure way to distinguish between temporary and permanent conditions, for the agent that might normally be expected to change a condition cannot be relied upon. The second strategy is the simplest one to employ, but it places a great burden for exception handling into almost every program component in a system. In practice, therefore, the tendency is to employ very simple, and often unsatisfactory, exception handling, rather than to make every invocation of an operation a detailed special case.

The third strategy for exception handling has much to recommend it if sufficient flexibility can be achieved in designating the trap location. All too often in practice, the trap location is a default provided by a system, and the action taken may prevent the continuation of the process that was in execution when the exception occurred. This need not be the case, however. It seems quite reasonable for a process, upon acquiring an access binding to a resource, to designate an exception handler within its address space to which control is to be returned if the intended use of the resource should
fail. Once this trap location has been designated, it can be embedded within the capability that is held for use of the resource, and no additional code is required within the customer process to check the occurrence of an exceptional condition arising from any use it may make of the resource.

When a capability is exercised, the exceptional return address supplied in the capability is to be saved in the local data segment of the procedure being invoked, just as the normal return address is customarily saved. This trap location thus becomes a part of the process state during its activity within the resource, even though the capability in which it was communicated is not bound to the address space of the resource.

We now return to the problem raised by preemptive revocation of rights to access a monitor that contains condition queues. In such a case, we must assume the worst, namely, that some queues are occupied at the time of revocation. To avoid deadlockoning of processes that may be suspended upon the queues of an inaccessible monitor, the queues should be inspected, and the processes waiting there should be awakened, with control returned at the designated exception trap location. For each monitor type that is declared with the revocable attribute, we assume the existence of an implicitly defined entry procedure called Wakeup, whose function is to empty the condition queues in this way.

All of the mechanisms so far proposed can be illustrated with an example. The resource type that we wish to use is Channeltype, given as an example in Section II. However, we now wish to declare it to be a revocable resource type, and so its declaration is modified to read

```
type Channeltype = monitor revocable;
  rights Receive = {Get};
      Send = {Put};
      Revocate = {Setkey, Wakeup};
  ...
end monitor
```

The procedures Setkey and Wakeup are implicitly defined and supplied by the programming language implementation. The manager of instances of Channeltype that are used to realize an interprocess communications facility is now to implement a policy imposing a time limit on the use of any instance of a communications buffer.

```
type IPCtype = manager of IPC : Channeltype;
  rights Receiver = Receive + {Open, Close, Status};
      Sender = Send + {Open, Close, Status};
      Revoker = {Timeout};
  const N = 50;
      Maxinterval = ...
      Maxint = largest representable integer
  var Currentime, Mintime : integer;
      Channel : array [1..N] of record
        ID_number, EHC, Timelimit : integer;
  ...
end monitor
```

Note that the manager has declared that it holds Revocate rights upon the instance of Channeltype that it manages, but has not incorporated Revocate into any exportable set of rights. Whenever an execution of the Open procedure results in allocating a new instance of Channeltype (i.e., one whose entry hold count field was formerly zero) to a customer process, Timelimit is set to the current clock time plus the constant Maxinterval. The Timeout procedure is given as follows:

```
procedure entry Timeout;
  var Currentime : integer;
      Index : 1..N;
begin
  Timerinterrupt.wait;
  Currentime := clock; Mintime := Maxint;
  for Index := 1 to N do
    with Channel [Index] do
      if EHC > 0 then
        if Currentime > Timelimit then
          begin
            EHC := 0; ID_number := 0;
            Instance.Setkey; Instance.Wakeup
          end
        else if Timelimit < Mintime then
          Mintime := Timelimit
      end
end;
```

Although we have omitted the additional code for the sake of brevity, the Timeout procedure would also signal a process that had been suspended on a queue while executing the procedure Open in the event that no instance of Channeltype was available.

An instance of the manager type is declared in the global environment of all customer processes,

```
var IPCpool : IPCtype;
```

and individual customer processes can declare managed instances as needed:

```
var IPCchannel : Channeltype (Receiver) from IPCpool;
```

A process that wishes to specify its own exception handler in case of resource failure will do so by adding a label constant as an extra argument in a call to the allocation procedure.

```
IPCchannel.Open (Channel_number, Exception_handler);
```

The address bound to this label will be inserted into the capability when an access binding is made.

1 The keywords condition interrupt denote a condition queue whose signal operation is to be provided by an external process when the specified condition becomes true. We introduce it here as notation only.
Another process must be dedicated to activating the Time-
out procedure of the manager,

    process Timer;
    var X : Channeltype (Revoker) from IPCpool;
    begin repeat X.Timeout until False end;
endprocess.

Notice that Timer uses the name X only to access the manager;
it does not hold any right to access an instance of Channel-
type, nor does it even hold a right to call for allocation.

V. SIMULTANEOUS SHARING OF A RESOURCE

A monitor allows resource sharing by concurrent processes in
a highly restrictive way; at most one process can be active in
a monitor at any time. This restriction guarantees the safe use
of a shared resource in that a variable shared within a monitor
cannot be updated by one process while a second process re-
ters to it. However, in doing so, it enforces sequentiality
among all processes that hold capabilities for use of the mon-
tor, even if these capabilities do not allow variables to be
updated. In practice, this often means that a programmer will
forego the protection afforded by the monitor in order to
allow simultaneous sharing of a resource. We wish to inves-
tigate means for allowing a shared resource to be simultaneously
accessed by several processes when a system’s designer is will-
ing to certify that no harmful interference can arise.

To accomplish this, we shall allow access to a shareable
instance to be dispensed by a capability manager. With this
mechanism, we can assure that

1) only those processes authorized to share an instance of a
class resource can hold an access binding to it;
2) no process can exercise a forbidden right, i.e., one that
has not been designated in the set of rights declared for the
instance allocated to the process. This allows one to verify
that mutual interference cannot occur among processes con-
currently accessing the resource.

With the addition of a revocation mechanism similar to the
one we have proposed for monitor types, we can further
ensure that

3) a revocable class resource can be reclaimed at the earliest
possible point in time. This is the time at which all processes
have completed the execution of procedures of the class that
was in progress when revocation occurred.

To obtain the revocable property of a shared class, we must
introduce a prologue and an epilogue into the calling sequence
of every procedure of the class, for it is necessary to be able
to determine when processes are active within it.

The difficulty inherent in class revocation is to determine
when process activity within the class has terminated. In
the case of a revocable monitor, the Setkey procedure could not
be completed by its manager until a process active in the
monitor had exited (or executed a wait) because of the en-
forced mutual exclusion. In a class, an unsynchronized Setkey
could be completed immediately, leaving the manager without
knowledge of whether or not any process remains active in the
class.

A solution is to maintain a count of processes active in the
class, and to embed the counting operations into the calling
sequence. For this purpose, we postulate an implicitly de-
clared monitor type, an instance of which is associated with
each instance of a revocable class. The access validation key
of the class instance is also embedded within this monitor, and
the monitor has access to the contents of a capability record.
Although the declaration of this monitor is not visible to the
programmer, it is equivalent to

    type ClassGuard = monitor;
    var
      Count : Integer;
      Key : Keytype;
      NoActivity : condition;
    rights
      Caller = {Prologue, Epilogue};
      Revoker = {Setkey};
    procedure entry Prologue;
    begin
      if capability.Key = Key then
        Count := Count + 1
      else
        return to capability.exception_trap
    end;
    procedure entry Epilogue;
    begin
      if Count = 0 then
        NoActivity.signal
    end;
    procedure entry Setkey;
    begin
      Key := unique value;
      if Count > 0 then
        NoActivity.wait
    begin (* initialization *)
      Key := unique value;
      Count := 0
    end
endclass.

The ClassGuard monitor is a standard type that accompanies
every revocable class. When an instance K of a revocable class
is declared, an instance of the ClassGuard monitor is also de-
clared implicitly, and is bound to the same name, K. The sets of
rights Caller and Revoker enhance the sets of rights defined
for the revocable class. Caller rights are always appended to
any set of rights declared for use of the class itself; Revoker is
a new set of rights that can be declared for an instance of a
revocable class. Caller rights are invisible to the programmer;
they are used only by the compiler in expanding a call upon an
entry procedure of the class.

The calling sequence to exercise a right P upon an instance K
of a revocable class is then expanded as follows. A call written
by the programmer as

    K.P (parameters)

becomes
[if not P in capability.rights then
  return to capability.exception_trap;
K.Prologue;]
  K.P (parameters); (* the actual call *)
  [K.Epilogue]

where the brackets indicate implicitly generated code inserted by the compiler. The exception_trap address is that furnished by the customer process as a parameter in a call of the allocation procedure, an enhanced right supported by the manager of the class.

While mutual exclusion is necessary within the ClassGuard monitor to prevent interference upon use of the Count and the condition variables, it is enforced upon customer processes only during execution of the short Prologue and Epilogue procedures. Since the single right exercised by the manager, namely, to call Setkey, requires entry only into the ClassGuard monitor, the manager obtains almost immediate access to the validation key of the revocable class. By use of the condition variable, the manager is delayed in returning from Setkey to resume scheduling of the resource until all activity within the now-revoked class instance has ceased. Note that since only the manager of a class instance can ever hold a right to Setkey, the queue associated with NoActivity can never hold more than a single process entry.

We illustrate the use of a revocable shared class to solve the second readers and writers problem [14], [15]. The problem can be succinctly defined. A file is to be shared among several processes, called readers or writers according to the rights they seek to exercise. A writer must have exclusive access to the file, whereas many readers can share access simultaneously without interference. Writers are to be given preemptive priority over readers. The activity of a reader or a writer need not be atomic, that is, when a process acquires access to the file, it may perform repeated operations unless prevented from doing so by the synchronization mechanism.

This problem has served as a tilting-horse for many proposed synchronization schemes. It can be solved with the use of semaphore primitives to achieve synchronization; however, such solutions are of sufficient complexity that subtle errors were discovered in the first published versions [16], [17]. High-level language constructs, such as monitors, have allowed the solutions to be simplified greatly [2], [18] by eliminating the need to explicitly program a complicated sequence of semaphore operations in order to synchronize accesses by several processes to a critical set of shared variables. But even these constructs have not resulted in a solution immune to error in case a customer process is badly behaved. Since the file to be shared by readers and writers cannot reside in a monitor (this would force mutually exclusive access by readers as well as by writers), a customer process holds a static access binding to the shared resource, whether or not it executes the monitor procedures in the proper sequence to achieve the desired synchronization of access.

Although the readers and writers problem has not ordinarily been considered in the context of protection, the control of access afforded by the capability manager can be used to obtain a simple solution. The shared file can be declared to be

```plaintext
type Filetype = class revocable;
  rights
    Reader = {Read}; (* + Caller *)
    Writer = {Write}; (* + Caller *)
    Revoker = {Setkey};
    ...
  endclass.

Both the Reader and Writer sets also implicitly include the Caller set, which defines the extended rights furnished by the ClassGuard monitor to customer processes. The names of the rights in the Caller set are invisible, however.

Since the class has the attribute revocable, key validation is performed on a capability that seeks to exercise any right (other than Setkey) upon an instance of the class. The right to call Setkey is used by a manager to preempt continued activity by a reader process whenever a writer requests access to a shared file. Of course, a reader is not prevented from completing a Read already begun, but no further Read operations will be allowed to start. Because the Epilogue procedure is invoked (by an implicit call) following every Read operation, the manager is assured of regaining control at the earliest possible time after it initiates a Setkey operation to revoke Reader capabilities.

A manager of the shared file may be declared as follows:

type FileMgertype = manager of File : Filetype;
  const
    N_processes;
  rights
    Reader = Reader + {OpenRead};
    Writer = Writer + {OpenWrite, CloseWrite};
  var
    N_writers : 0 .. N_processes;
    SharedFile : Filetype(Revoker);
    Readerproceed, Writerproceed : condition;
  procedure entry OpenWrite (Recovery_pt : label);
    begin
      N_writers := succ(N_writers);
      if N_writers > 1 then
        Writerproceed.wait;
        SharedFile.Setkey; (* revoke access of all readers *)
        File := SharedFile (* also inserts Recovery_pt into
          capability.exception_trap *)
      end;
  end;
  procedure entry CloseWrite;
    begin
      if File <> nil then
        begin
          File := nil;
          N_writers := pred(N_writers);
          if N_writers > Ø then
            Writerproceed.signal
          else
            Readerproceed.signal
        end
    end;
```

Authorized licensed use limited to: Yale University. Downloaded on February 23,2022 at 15:16:46 UTC from IEEE Xplore. Restrictions apply.
A capability diagram for this example is shown in Fig. 3. A writer process might have the structure:

```plaintext
procedure entry OpenRead (Recovery_pt : label);
begin
  while N_writers > ∅ do
    Readerproceed.wait;
    File := SharedFile; (* also inserts Recovery_pt into capability.exception_trap *)
    Readerproceed.signal (* propagates the signal *)
  end;
  begin (* initially *) N_writers := ∅ end
endmanager.
```

A well-behaved reader process must provide a recovery point to which control will be transferred in case an access exception occurs since its Reader rights may be revoked without prior notification. This trap address is provided only once, in the call upon the allocation procedure OpenRead.

```plaintext
process Readerprocess;
label
  ReStart;
var
  InFile : Filetype (Reader) from FileMgr;
begin
  ReStart : initialize transaction;
  InFile.OpenRead (ReStart);
  repeat
    InFile.Read (datum);
    digest datum
    until end of transaction
end process.
```

Notice that the specified restrictions on use of the shared resource cannot be compromised, even if incorrect process code is written. Nor can the resource be made unavailable by a runaway reader process. (A runaway writer is, of course, more dangerous.) Nor can a process misrepresent its intended use of the resource, and thereby gain any unauthorized rights. The amount of dynamic checking of capabilities and the amount of synchronization that is enforced appears to be close to the minimum that is required to have a safe solution.

### VI. Conclusions

We have presented arguments for the use of a language-based capability mechanism for use in the design and implementation of operating systems. A number of programming language facilities have been proposed for the realization of such a mechanism, building upon an assumed base that implements monitors. Those aspects of a capability mechanism necessary for protection have been embedded in a programming language so that they cannot be intentionally or accidentally subverted by a programmer who makes use of the facility. Other aspects having to do with policy, such as the specification of exception handling, the concept of ownership, allocation, and revocation strategies, have been left in the hands of the system designer. While we have not given exhaustive examples, we believe that most of the flexibility achievable with kernel-based capability systems can also be realized in a language-based system. Since the complexity and size of capabilities does not need to be uniform throughout a system, the overhead accompanying the use of sophisticated capabilities can be restricted to the points at which the sophistication is needed.

It is only fair to point out a possible limitation of any language-based capability system. Since responsibility for
the manipulation of capabilities is distributed throughout a system rather than localized in a kernel, the protection afforded by the capabilities subsystem is only as sound as the security of the entire system. We have given the assumptions on which this security rests in a passive system, and these assumptions are not unreasonable. However, in a system that undergoes periodic maintenance, reloading, and regeneration, these assumptions must be extended to include security at program linkages, program loading from secure files, and the correctness of system generation software. Some of these same assumptions must be made to ensure the security of a kernel-based protection system, but not to the same degree.

ACKNOWLEDGMENT

We wish to thank our colleagues, A. J. Bernstein and K. Ekanadham, for reviewing a draft of the manuscript.

REFERENCES


Richard B. Kieburz (S'53-M'61) was born in Spokane, WA, on November 28, 1933. He received the Ph.D. degree in electrical engineering from the University of Washington, Seattle, in 1961.

Until 1968 his research was in microwave propagation and antenna theory. At that time, he accomplished a change in his field of specialty, aided by an NSF Science Faculty Fellowship that enabled him to spend the 1968-1969 academic year at Stanford University, Stanford, CA, studying computer science. Since then he has worked in programming languages, program verification, and compiler design. He is also the author of a series of programming textbooks. He has been a member of the faculty of the State University of New York, Stony Brook, since 1964, where he is currently a Professor of Computer Science. From 1970 to 1975 he was Chairman of the Computer Science Department at Stony Brook. Prior to 1964 he served on the faculty of New York University, New York, NY.

Abraham Silberschatz received the Ph.D. degree in computer science from the State University of New York, Stony Brook, in 1976.

He is currently on the faculty of the Department of Mathematical Sciences at the University of Texas at Dallas. His research interests are in the areas of operating systems, parallel processing, and programming languages.