Key-Lock Mechanisms for Object Protection in Single-Address-Space Systems

Lanfranco Lopriore
Dipartimento di Ingegneria dell'Informazione: Elettronica, Informatica, Telecomunicazioni, Università di Pisa
via G. Caruso 16, 56122 Pisa, Italy
Email: l.lopriore@iet.unipi.it

Abstract— **This paper focuses on memory addressing environments that support the notion of a single address space. We consider the problem of hampering access attempts to the private objects of a given thread, when these attempts are generated by unauthorized threads of different processes. We introduce two different forms of access privilege representation - handles and gates - which are designed to coexist within the boundaries of the same protection system. The handle concept is a generalization of the classical protected pointer concept. A handle associates several keys (passwords) with an object name. Each key grants a specific access right to the named object. A gate is a compact representation of access privileges, which uses a single bit to encode an access right. Handles are protected from forgery by key sparseness. They can be freely mixed in memory with ordinary data. On the other hand, gates are sensitive data that must be kept segregated in private memory regions of the protection system. The dualism of handles and gates makes it possible to take advantage of the simplicity of access right distribution and object sharing between threads, which is characteristic of key-based protection systems, and to avoid the negative impact on overall system performance, which results from the large key size and the high costs of lengthy processing that are connected with key validation.**

Index Terms—**access right, process, protection, revocation, single address space, thread**

I. Introduction

In a classical model of process interaction and cooperation, a *thread* is defined as an elementary, active entity capable of accessing and modifying the passive entities of the system, called *objects*. A *process* is the result of the joint activities of several tightly coupled threads. Interactions between threads of different processes are comparatively rare, and efficiency in these interactions is not a stringent requirement. As far as protection is concerned, mechanisms are required to hamper access attempts to the private objects of a given thread, if these attempts are generated by unauthorized threads of different processes. Protection mechanisms are not required between threads of the same process. A thread of a given process can freely access all the objects of the other threads in this process. These objects are considered as part of a single pool.

In a traditional virtual memory system, each process is executed within the boundaries of its own address space. This address space acts as a repository for all the objects accessible by the process. Private object protection is guaranteed by address space separation [21]; a process cannot even name the private objects of any other process. If threads of two or more different processes need to share access to a given object, and this object is placed at different addresses in the virtual spaces of these processes, complex synonym problems arise. This happens in a virtual-addressed cache [1], [5] as well as for instance in the circuitry for virtual-to physical address translation [14], [15]. One solution is to place the object at a virtual address that is fixed for all the processes involved in object sharing. This solution implies that a consensus has been reached among these processes [13]. On the other hand, within the boundaries of a single process, all threads access the same address space, and consequently, object sharing between these threads is straightforward [37].

In a different, *single-address-space* system, all processes reference a common virtual space [2], [17], [25]. The meaning of a virtual address is unique, and is independent of the process issuing this address. The validity of the name (virtual address) of a given object extends to all processes, and each process uses this name to reference the object. No *ad-hoc* mechanism is required for object sharing between threads of different processes [8], [9], however, the introduction of mechanisms for private object protection is mandatory [7], [23], [33].

With reference to a single-address-space environment, we will consider a classical protection system paradigm that associates threads with *protection domains* [24]. The protection domain of a given thread specifies the objects that the thread can access and the *access rights* that the thread holds to each of these objects. Objects are typed and the definition of the type of a given object states the operations that can be applied to this object, a set of access rights, and the associations between the operations and the access rights. Let X be an object type, R_0, R_1, …, R_{r-1} be the operations defined by X, and AR_0, AR_1, …, AR_{n-1} be the access rights defined by X. A thread can execute operation R_i on object B of type X only if the domain of this thread includes all those access rights to B that are required to execute R_i successfully.

A basic problem when designing protection systems is how to express relationships between objects and access rights. An effective solution is to associate one or more *passwords* with each protected object, one password for each access right defined by the type of this object. A

© 2009 ACADEMY PUBLISHER
doi:10.4304/jcp.4.12.1322-1331

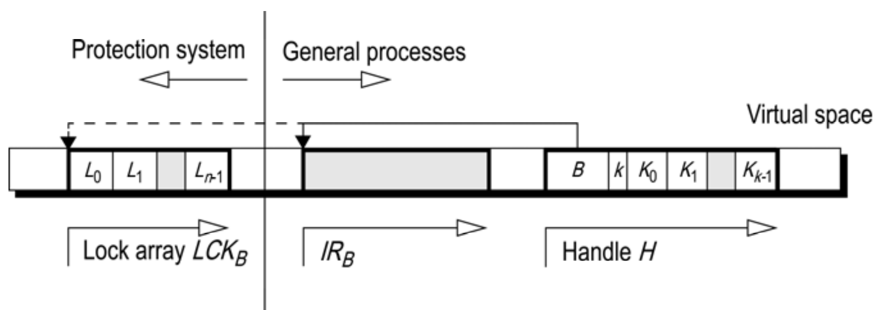

Figure 1. Configuration of handle H, lock array LCK_B and internal representation IR_B of object B.

protected pointer $\{B, W\}$ consists of an object name B and a password W. If the password matches one of the object passwords, the protected pointer grants the corresponding access right to object B.

Protected pointers have a potentially negative effect on the overall performance of the system. Several protected pointers are necessary to certify possession of a complex access privilege expressed in terms of several access rights. This negatively affects memory requirements, especially in a single-address-space environment, on account of the large size of object names ensuing from the large address space. (A large address space is necessary to avoid unacceptable limitations on the amount of virtual space available for each process [6].) Furthermore, one or more passwords must be validated each time an operation is executed on a given object, to ascertain whether the thread issuing this operation holds the required access permissions. Password validation negatively affects execution times, for instance if an operation is executed as part of a loop. Space and time problems are exacerbated by the large password size necessary to prevent processes from forging keys.

This paper proposes solutions to these performance drawbacks. Firstly, we present a relaxation of the close relationship that exists within a protected pointer between an object name and a single password. We propose a variant of the protected pointer concept, called a *handle*, which makes it possible to associate an object name with several passwords (called *keys* [18]). Handles are protected from forgery by key sparseness, and can be freely mixed in memory with ordinary data.

Secondly, we introduce an alternative representation of access privileges. In this new representation, *gates* are used to encode each access right in a single bit. Gates are sensitive data that must be kept segregated in private memory regions of the protection system. Handles and gates are designed to coexist within the boundaries of the same protection system.

Our design satisfies two essential requirements:

- The resulting system should encompass the advantages deriving from the simplicity of access right distribution and object sharing between threads, which characterize password-based protection systems.
- The negative impact on the overall system performance that ensues from large passwords and repeated actions of password validation should be kept to a minimum.

The remainder of this paper is structured as follows. Section II presents our view of handle-based protection. Section III introduces the dualism of gates and handles, and illustrates gate use in object access with special reference to object sharing between threads. Section IV evaluates the results and discusses the relationships existing between our mechanisms for object protection and a number of other mechanisms proposed in the literature. Finally, Section V summarizes the most important features of the approach to object protection proposed in the previous sections. The focus is on the significant advantages of keys and locks for representing the state of a protection system.

II. HANDLE-BASED PROTECTION

Let B be an object of type X, and $AR_0, AR_1, \dots, AR_{n-1}$ be the access rights defined by X. We associate a set of *locks* $L_0, L_1, \dots, L_{n-1}$ with B, one lock for each access right defined by X. A handle $H = \{B, K_0, K_1, \dots, K_{k-1}\}$ consists of object name B and a set of keys $K_0, K_1, \dots, K_{k-1}$ for this object. Key K_i in handle H is *valid* if it matches one of the locks, say lock L_j, associated with B. Let AR_j be the access right corresponding to lock L_j. In such situations, possession of H certifies possession of access right AR_j on object B. The quantity k is handle-specific; different handles may well include a different number of keys.

The locks of a given object are part of the private portion of the internal representation of this object. A form of lock segregation in memory is necessary, so that no thread can access the locks and read or modify them. This result is obtained as follows. For each given object B, we reserve an area within the boundaries of the virtual space region of the protection system. This memory area is called the *lock array* LCK_B. The LCK_B address is a function of object name B. The j-th element of LCK_B contains the j-th lock, L_j. Fig. 1 shows the memory configuration of object B with special reference to the relations between handle H, lock array LCK_B and the internal representation of the object, IR_B.

If the key size is large enough, threads are not able to forge keys nor can they violate the integrity of the protection system. In such situations, if a thread attempts to assemble a handle by associating an object name with an arbitrary key, the probability that this key matches one of the locks associated with the object is negligible. Let us refer to a memory configuration of k contiguous memory cells reserved to store the keys of a given handle. If a thread erroneously attempts to use a non-existing key, say

© 2009 ACADEMY PUBLISHER

the $(k+1)$-th key, key validation will use the contents of a memory cell that does not actually contain a key. Of course, the validation is destined to fail.

An interesting property of handles is the absence of any restriction on the physical position of the handle components in memory. A process that holds handle H is free to store the components of this handle in arbitrary virtual space positions, while keeping track of the associations between these components in the program algorithms. The memory cells reserved for storage of object name B and keys $K_0, K_1, \ldots, K_{k-1}$ of handle H for instance do not need to be contiguous.

A. Using Handles

Let us refer to thread T which is attempting to execute operation R on object B, and let AR_j be the access right that allows the successful execution of R. When T issues the call to R, it exhibits a handle $H = \{B, K_0, K_1, \ldots, K_{k-1}\}$ referencing B as an argument of the call. The execution of R ascertains whether H includes access right AR_j. To do this, the internal representation of B is accessed and lock L_j corresponding to AR_j is compared with keys $K_0, K_1, \ldots, K_{k-1}$. If a match is found, execution is allowed, otherwise a protection exception is raised and execution fails.

High costs in terms of execution times are associated with these iterated key-lock comparisons. We will now introduce gates as an alternative representation of access privileges for an effective solution to this performance problem.

III. GATES

A *gate* G is a pair $\{B, \mathcal{AR}\}$, where B is the name of an object and the access right field $\mathcal{AR}$ contains the specification of a set of access rights to B. We hypothesize that an upper limit exists for the number of access rights that can be defined by any given object type. In this hypothesis, the length of $\mathcal{AR}$ is fixed and type-independent. Let X be the type of object B. In gate G, the j-th bit of $\mathcal{AR}$ is associated with the j-th access right, AR_j, defined by X. If this bit is set, then G includes AR_j.

Let $H = \{B, K_0, K_1, \ldots, K_{k-1}\}$ be a handle for object B. Gate G is *equivalent* to H if the access rights specified by the $\mathcal{AR}$ field are those and only those specified by the keys in H. This means that (i) if the j-th bit of $\mathcal{AR}$ is set, then a key in H matches lock L_j corresponding to access right AR_j; and (ii) if the j-th bit of $\mathcal{AR}$ is clear, then no key in H matches L_j.

The concept of equivalency of gates and handles can be extended to two or more handles referencing the same object. Let $H' = \{B, K'_0, K'_1, \ldots, K'_{k'-1}\}$ and $H'' = \{B, K''_0, K''_1, \ldots, K''_{k''-1}\}$ be two handles referencing object B, for instance. Gate $G = \{B, \mathcal{AR}\}$ is equivalent to the H', H'' pair if the access rights specified by the $\mathcal{AR}$ field are those and only those specified by the keys in H' and H''. This means that (i) if the j-th bit of $\mathcal{AR}$ is set, then at least one key in $\{K'_0, K'_1, \ldots, K'_{k'-1}, K''_0, K''_1, \ldots, K''_{k''-1}\}$ matches lock L_j corresponding to access right AR_j; and (ii) if the j-th bit of $\mathcal{AR}$ is clear, then no key in $\{K'_0, K'_1, \ldots, K'_{k'-1}, K''_0, K''_1, \ldots, K''_{k''-1}\}$ matches L_j.

A. Protection Tables

As pointed out in Section II, handles are protected from forgery since it is practically impossible to guess a valid key. As a consequence they can be freely stored in unprotected memory regions. On the other hand, if we allow a process to gain unrestricted access to a given gate, this process will be in a position to modify the access right field of this gate and add new access rights, or even cause the gate to reference a different object. Thus, the enforcement of a form of gate segregation in memory is mandatory. This result is obtained by restricting the storage of gates to reserved memory areas that are part of the private memory space of the protection system.

The protection system associates a *protection table* PT_P with each given process P. This table aims to contain the gates held by P, which are shared by all the threads that form this process. Each entry of the protection table can store a gate. Two actions are permitted on a protection table, i.e. to write a gate into a given protection table entry and to read the gate contained in a given entry. The write of a gate occurs as a result of a conversion of a handle to gate form. This effect can be obtained by executing a protection system primitive called *Convert*() (the actions involved in the execution of this primitive are analyzed below). A gate read occurs as part of access privilege verification, in the execution of a protection system primitive called *Verify*(). The protection table entries can only be accessed using *Convert*() and *Verify*().

At any given time, a register of the protection system, the *running process register RPR*, contains the name P of the *running process*, i.e. the process of thread T being executed at this time. Thus, the contents of RPR identify the protection table PT_P that is used each time T performs an access attempt to a protected object. This is done to verify whether T holds the required access privilege. When thread T relinquishes the processor and another thread T' is assigned the processor, if T and T' are part of the same process the contents of RPR do not change. On the other hand, if the two threads belong to different processes, say P and P', then the name P' of the new process is written into RPR.

Thus, the threads of a given process P can only access the entries of the protection table PT_P associated with this process. These threads cannot take advantage of the access privileges held by a different process, which are specified by the gates stored in the protection table of this process. This condition is essential for private object protection across process boundaries.

B. Handle to Gate Conversions

Let P be the name of the running process, specified by the contents of the running process register RPR, and let PT_P be the protection table associated with P. The *Convert*() primitive of the protection system makes it possible to convert handle H into an equivalent gate G and store this gate in an entry, say entry E, of PT_P. This primitive has the form

Convert(*addrH*, *n*)

© 2009 ACADEMY PUBLISHER

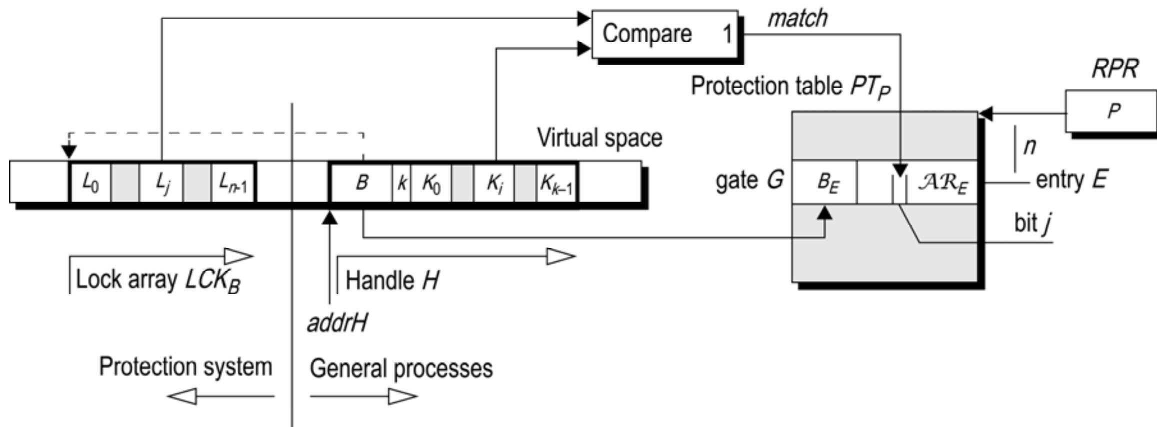

Figure 2. Actions involved in the execution of the *Convert*(*addrH*, *n*) primitive, in the iteration for key K_i.

Argument *addrH* is the address of the memory area containing the components B, K_0, K_1, …, K_{k-1} of handle H. Argument n is the index of entry E in PT_P. Let B_E and $\mathcal{AR}_E$ denote the object name field and the access right field of E. Execution of *Convert*() compares quantity B with the contents of B_E; if no match is found, quantity B is inserted into B_E and $\mathcal{AR}_E$ is cleared. Then, each key K_i, $i = 0, 1, \ldots, k - 1$, is compared with the locks associated with object B, which are stored in lock array LCK_B. If a match is found and L_j is the matching lock, then the j-th bit of $\mathcal{AR}_E$ is set. Fig. 2 shows the actions caused by execution of *Convert*() with special reference to the activities involved in the generic iteration for key K_i, if a match is found between this key and lock L_j. As a consequence of the match, the protection table entry reserved for gate G is accessed and bit j of the access right field of this entry is set.

It should be noted that if entry E already contains a gate for object B, execution of *Convert*() adds the access rights specified by the handle at address *addrH* to those already included in E. This additive behavior of *Convert*() allows us to give the form of a single gate to several, distinct handles that reference the same object. Let H' and H'' be two handles that reference object B, for instance, and let *addrH'* and *addrH''* be the addresses of the memory areas containing the components of these handles. Two subsequent executions of *Convert*() can be used to produce a gate equivalent to the H', H'' pair, as follows:

Convert(*addrH'*, *n*)
Convert(*addrH''*, *n*)

The first call to *Convert*() produces a gate equivalent to H' and stores this gate in the n-th entry of PT_P. The second call modifies the access right field of this entry to include the access rights specified by H''.

C. Access Right Verification

The *Verify*() protection system primitive allows us to ascertain the presence of a given access privilege in a given gate. This primitive has the form

Verify(*n*, *B*, *m*)

Argument n is the index of an entry E of protection table PT_P associated with the running process P. Argument m is a *mask* whose size is equal to the size of the access right field of a gate. Execution of *Verify*() returns *true* if both the following conditions are met: (i) the object name field B_E of entry E contains quantity B; and (ii) for each bit in mask m, if this bit is set, then the corresponding bit of the access right field $\mathcal{AR}_E$ of E is set. The mask field allows us to ascertain the availability of complex access privileges expressed in terms of several access rights by a single execution of *Verify*(). To this aim, we will assemble a mask featuring several bits set, i.e. the bits corresponding to each of these access rights.

D. Using Gates

In a protection environment supporting a duality of handles and gates, let us refer to thread T holding handle H for object B, and let us consider the actions needed to execute operation R on this object. Thread T reserves an entry E of protection table PT_P for the storage of a gate G equivalent to H. Then, the *Convert*() primitive is executed to generate this gate. Thread T is now in a position to certify possession of access right AR_j which allow successful execution of operation R by transmitting the index n of E in PT_P as an argument of the call to R. In the execution of this operation, quantity n is used to call the *Verify*() primitive and perform the required access right checks.

As seen in Subsection II.A, in the absence of gates, a sequence of key-lock comparisons is necessary each time an operation is executed on a protected object to validate the object access. In contrast, in an environment featuring a dualism of handles and gates, key-lock comparisons are only necessary once for each protected object, when *Convert*() is executed to generate a gate for this object. An inline expansion of *Verify*() can be significantly faster than several key-lock comparisons. The resulting reduction in processing time costs is especially significant for an object access that requires the possession of a complex access privilege expressed in terms of several, distinct access rights. As seen in Subsection III.C, the presence of a privilege of this type can be ascertained by a single execution of *Verify*(), by choosing an appropriate configuration for the mask argument, m.

Convert() execution times are negatively affected by the fact that the validation of each given key K_i of handle H requires a comparison of this key with each lock asso-

© 2009 ACADEMY PUBLISHER

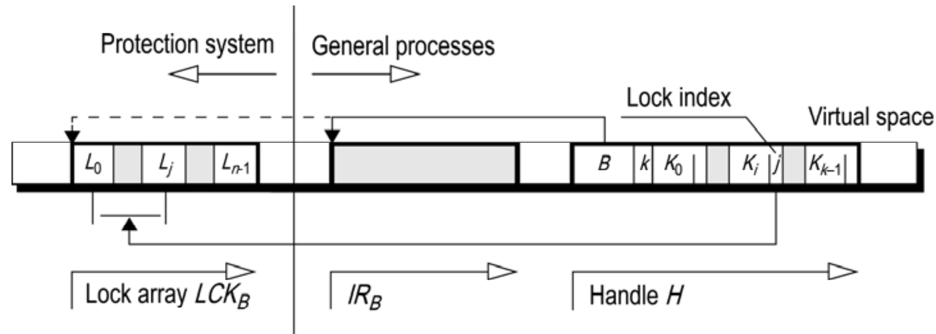

Figure 3. Configuration of handle H, lock array LCK_B and internal representation IR_B of object B in the presence of lock indexes.

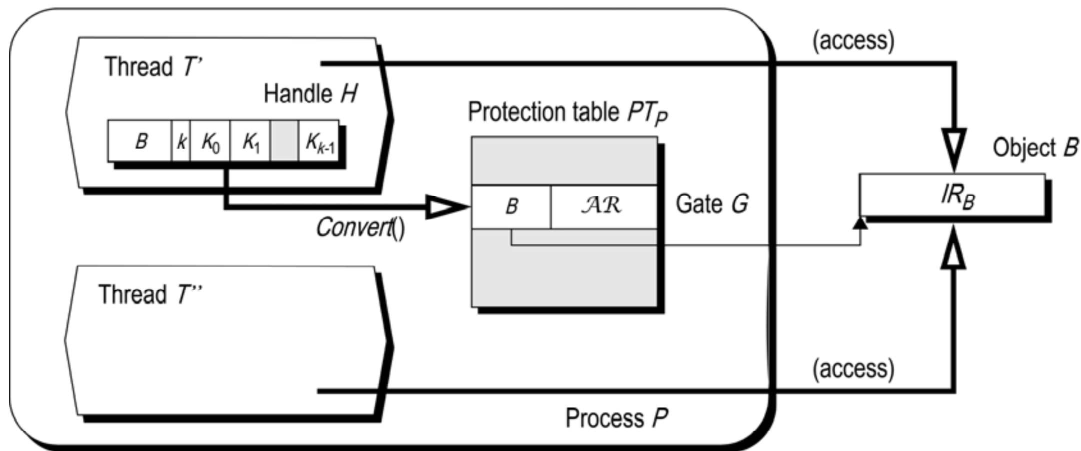

Figure 4. Sharing objects between threads of the same process.

ciated with B, iteratively, until a matching lock L_j is found. One solution is to extend K_i to include a *lock index field* containing the index j of L_j in lock array LCK_B. In this case, a single key-lock comparison is sufficient to validate K_i (if K_i and L_j do not match, validation fails). Protection system integrity is guaranteed by the fact that any erroneous configuration of the lock index field invalidates the key. Fig. 3 shows an extension of the configuration of Fig. 1 obtained by adding lock indexes to keys.

E. Sharing Objects

All the threads of a given process share the same protection table. It follows that, if one of these threads converts a handle into gate form, all the other threads can use the gate resulting from the conversion. Let *T'* and *T"* be two threads of process *P*, let PT_P be the protection table of this process, and let us suppose that thread *T'* holds a handle *H* referencing object *B*. Suppose that *T'* executes the *Convert*() primitive to generate a gate *G* equivalent to *H* and stores this gate in an entry of PT_P. Thread *T"* will now be able to take advantage of the contents of this entry and access *B*. Fig. 4 shows the two threads *T'* and *T"*. Thread *T"* accesses object *B* by taking advantage of a gate *G* for this object which was generated by thread *T'*.

Thus, the protection table mechanism supports the sharing of an object between threads of the same process. In such situations, a single handle is sufficient to encode the access privileges of all these threads, provided that one of them converts the handle into a gate form. The cost in terms of execution times is that of a single *Convert*(), irrespectively of the number of threads involved in the object sharing activity. Access privilege validation requires execution of the *Verify*() primitive on each object access. As pointed out previously, the cost of *Verify*() in terms of execution times is negligible.

On the other hand, the sharing of an object between processes explicitly requires handle transmission. Let *T'* be a thread of process *P'* and *T"* be a thread of process *P"*, and let us suppose that *T'* holds a handle *H* for object *B*. The sharing of *B* between *T'* and *T"* requires that *T'* transmits a copy of *H* to *T"*. This thread can now convert *H* into a gate and store this gate in an entry of its own protection table, $PT_{P''}$. Thus, *T"* gains access to *B*. Fig. 5 shows thread *T"* of process *P"* accessing object *B* after converting a handle *H* for this object into the form of a gate, *G"*. Thread *T"* cannot take advantage of gate *G'* generated by thread *T'*, as this thread belongs to a different process, *P'*.

In summary, the protection system enforces no degree of protection on the private objects of a given thread against accesses generated by the other threads of the same process. On the other hand, the protection system prevents a thread of a given process from accessing the private objects of the threads of a different process, unless access is permitted by an explicit handle copy.

IV. Discussion, and Relation to Previous Work

A. Capability-Based Protection Systems

Several methods have been proposed in the literature

© 2009 ACADEMY PUBLISHER

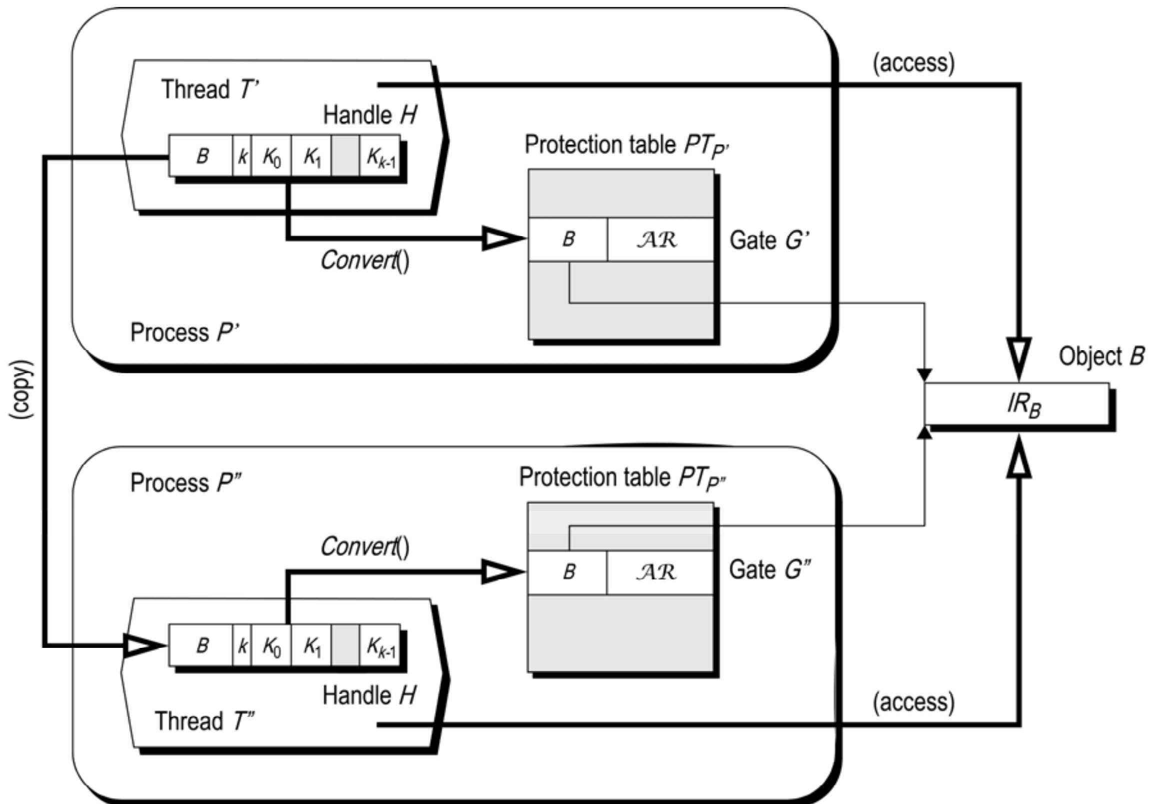

Figure 5. Sharing objects between threads of different processes.

to represent the state of the protection system in memory. A classical approach is based on the concept of a *capability* [20], [22]. This is a pair {*B*, $\mathcal{AR}$}, where *B* is the name of a protected object and $\mathcal{AR}$ is the specification of a set of access rights on *B*. Capability possession makes it possible to access *B* and accomplish the operations permitted by the access rights in $\mathcal{AR}$. A protection domain is specified in terms of a collection of capabilities. When a thread attempts to access object *B* and execute operation *R*, it must present a capability for this object, and the access right field of this capability must include the access rights permitting the execution of *R*. Several computer systems using forms of capability-based protection have been developed, e.g. the Cambridge CAP Computer [36], the Plessey System 250 [11], the Intel iAPX 432 [27], the EROS system [30] and the IBM AS/400 [31].

Capabilities can be freely copied, and the protection system does not mediate capability transmission between domains. Access privileges tend to migrate throughout protection domains [32], and the protection system has no means of keeping track of the domains that received a given privilege [29]. Consequently, the review and revocation of access privileges is hard if not impossible.

To represent a capability in memory we have to assign an identifier to each given object, and this identifier must be unique throughout the system. In a single-address-space environment, this can be obtained by using the virtual address of the first memory location reserved for object storage. Let us refer to object *B* of type *T*, for instance. We shall impose an upper limit to the number *n* of access rights that can be defined by any given object type. The access right field of a capability for *B* will be encoded in *n* bits. If the *j*-th bit is set, then the capability includes the *j*-th access right, AR_j, defined by type *T*.

Capability segregation

In capability-based protection systems, we must prevent unauthorized modifications of capabilities [28]. We must preclude a process from tampering with the internal representation of a capability and amplifying the access privileges granted by this capability, for instance by adding new access rights.

Capability segregation can be obtained by reserving specific memory areas for capability storage [11], [36]. These areas can be modeled as objects of a machine type that we shall call the *CapabilitySegment* type. The operations of this type are implemented by *ad-hoc* machine instructions, the *capability instructions*. These instructions make it possible to write capability values into capability segments and to read capability values from capability segments, for instance.

A different approach relies on special memory devices that associate a 1-bit *tag* with each memory cell. If set, the tag of a given cell specifies that this cell contains a capability [3], [35]. When a machine instruction is executed this produces the tag checks required for memory access validation. For instance, if a capability is accessed in memory and the tag of the storage cell involved in the access is clear, a protection exception is generated and execution fails

Both the capability segment approach and the tagged memory approach require the support of specialized processors. This is in sharp contrast to the current trend of

© 2009 ACADEMY PUBLISHER

processor standardization. If capability segments are used, significant drawbacks arise from the dualism of data segments and capability segments in terms of the complexity of object layout in memory. In such situations, the internal representation of an object is often structured as a tree of other objects. The root of this tree is a capability segment which we call the *object descriptor*. The capabilities in the object descriptor reference one or more data segments, which are reserved for storing the values of the data members. For an object whose internal representation includes other member objects, the object descriptor also includes the capabilities for the descriptors of these member objects. This complex structure complicates programming as well as compiler writing. Of course, the problem is mitigated if capability segregation is obtained by taking advantage of tagged storage, at the expense of using specialized memory devices and increasing hardware complexity. Furthermore, in a tagged memory environment, *ad-hoc* solutions must be devised for capability and data storage in the secondary memory, because of the need to save cell tags.

Of course, there is some similarity between the concepts of a capability and a gate. However, gates need to be stored in reserved memory areas, i.e. the protection tables, which are part of the protection system. On the other hand capabilities must be segregated from the data, but can be freely stored in the memory regions of general processes. For instance, a thread is free to move a capability from capability segment *S'* to capability segment *S"*, provided that this thread holds a capability for *S'* permitting to read capabilities, and a capability for *S"* permitting to write capabilities.

On the other hand, handles do not need to be segregated in memory at all. They are ordinary data values that can be stored in any storage region, in the primary memory as well as in the secondary memory. Significant advantages follow in terms of the complexity of object representation. The descriptor of a given object can now contain both the data members and the handles for the member objects, for instance.

In fact, handles and gates are two different methods of access right representation that are designed to coexist within the boundaries of the same protection environment. Handles are aimed at distribution of access privilege between processes, and gates are aimed at improving efficiency in access right verification and object sharing between threads of the same process.

B. Protected Pointers

Protected pointers (also called *password capabilities*) have been proposed as a solution to the problems connected with the use of capability segregation techniques based on capability segments or tagged memory banks [4], [26]. As pointed out in Section I, protected pointers do not need to be segregated in memory. They are protected from alteration and forgery by the practical impossibility of successfully guessing valid passwords.

Revocation

Protected pointers are an effective solution to the problem of access privilege revocation. By eliminating a given password from the list of valid passwords associated with the given object, we exercise a form of *partial* revocation [12] involving only those access rights that correspond to this password. Revocation is *transitive*, as its effects automatically extend to all threads holding a protected pointer expressed in terms of this password.

In our handle-based environment, similar effects can be obtained by changing the locks associated with a given object. Let L_j be the lock corresponding to access right AR_j. By modifying L_j we invalidate all handles that specify AR_j in terms of a key matching this lock. We can even associate several locks with the same access right. In this case, if we modify one of these locks, we produce a *selective* revocation of access rights. Let L_j' and L_j'' be two locks for access right AR_j. By changing one of these locks, say L_j', we invalidate only those handles that specify AR_j in terms of a key matching L_j', whereas the validity of all handles whose keys match L_j'' is not affected by the revocation.

Memory requirements

Protected pointers have high costs in terms of memory requirements. This is mainly a consequence of the close correspondence between passwords and access rights, so that a protected pointer grants a single access right and several protected pointers are necessary to certify possession of a complex access privilege expressed in terms of several access rights.

Let g be the size (in bytes) of an object name, w be the size of a password, and $p = g + w$ denote the size of a protected pointer. The possession of a complex access privilege defined in terms of several access rights, say n access rights, implies possession of as many protected pointers, with a total memory requirement of $n \cdot p$ bytes. In the presence of a 64-bit processor, g is 8 bytes. For 64-bit passwords and two access rights, the memory requirement of the protection information is 32 bytes. For three access rights, the memory requirement is 48 bytes.

In our key-lock environment, there are significant reductions in memory costs since a single handle can contain several distinct keys. An access privilege defined in terms of n access rights can be specified using a handle featuring n keys. The total memory requirement of this handle is $g + n \cdot k$ bytes, where g denotes the size of an object name and k denotes the key size. In the presence of a 64-bit processor and 64-bit locks, the memory requirement for two access rights is 24 bytes, with a memory space saving of 25 per cent compared to a solution using protected pointers. For three access rights, the memory requirement is 32 bytes, and the memory space saving increases to 33 per cent.

Space costs can be further reduced since the components of a given handle can be placed in arbitrary positions in the memory space. A thread holding several handles for the same object does not need to replicate the identifier of this object for each of these handles. The thread will rebuild the association between the identifier and the lock before calling the *Convert*() primitive. In such situations, the thread gathers these quantities into a suitable memory area and transmits the address of this area to *Convert*() using the *addrH* argument.

© 2009 ACADEMY PUBLISHER

C. Active Protection Domains

Mungi [10], [19] is a single-address-space operating system relying on password capabilities for object protection. In Mungi, an object is a collection of memory pages, and the access rights defined for each object include *read*, *write*, *execute* and *destroy*. The object address and length are stored together with the *owner password* in a system table, called the *object table*. An owner password grants full access rights. A password derivation scheme applied to the owner password makes it possible to obtain weaker passwords.

Each user process is asked to organize its own password capabilities into capability lists conforming to a standard format. The resulting data structure is called the *active protection domain* of that user process. This data structure is traversed by the kernel to validate each object access. When a process attempts to execute an operation on a given object, the kernel reads the passwords associated with this object from the object table and then searches the active protection domain of this process for a capability whose password matches one of the object passwords. If this search is successful and the password has sufficient strength, the access is validated; otherwise a protection fault is raised. The rationale behind active protection domains is that a thread should not have to deal with protection explicitly as long as it is accessing its own private objects.

The designers of the Mungi system argue that the costs of active protection domains in terms of execution times and complication of the overall system architecture are justified by the need to relieve the programmer from the burden of explicitly managing access privileges for private objects. In fact, the Mungi protection model forces user programs to adhere to fixed pervasive forms of access privilege organization. The need to arrange password capabilities into capability lists of a standard format within the boundaries of an active protection domain places undue restrictions on the programmer. These restrictions tend to hamper the principal advantage of password capabilities, i.e. the absence of any need to segregate password capabilities in memory.

In contrast, in our system if a thread wishes to execute an operation on a given object it must convert a handle for this object into a gate form. Then, the thread invokes the operation by specifying the index of the protection table entry that stores the gate resulting from the conversion. This explicit action of access privilege presentation is needed for private objects as well as for shared objects. Our protection paradigm does not distinguish private object accesses from shared object accesses.

In our opinion, the advantages in terms of overall system performance and system simplicity that ensue from our form of explicit access privilege management largely exceed the disadvantages, especially with the appropriate help from the compiler. Handle management and handle-to-gate conversions are easy compiler tasks. Few modifications are necessary to add access privilege management to a program written in the absence of protection, to make this program suitable to operate in an environment featuring handles and gates. Essentially, these modifications take place in the definition of the operations of the protected types. From the point of view of a process using a given object, handle usage can be assimilated to pointer usage. The compiler will be able to make the actions connected to access privilege presentation largely transparent to the programmer. Placing new burdens on the compiler is a current trend, which is exploited in many different aspects of computer design, including translation lookaside buffer management, cache control and data prefetching [16], [34], [38].

V. Concluding Remarks

Focusing on a memory addressing environment supporting the notion of a single address space, we have considered the problem of hampering access attempts to the private objects of a given thread, when these attempts are generated by unauthorized threads of different processes. A classical solution uses protected pointers to certify possession of access privileges. A protected pointer consists of an object name and a password. If the password is valid, the protected pointer grants the corresponding access right to the named object. This solution is prone to heavy performance drawbacks, in terms of both memory requirements and execution times. This is a consequence of the large password size that is necessary to prevent forgery and the repeated actions of password validation. Our main design goal was to reduce the impact of these drawbacks.

We have presented two different methods for access right representation, handles and gates, which were designed to coexist within the boundaries of the same protection environment. Handles are mainly aimed at access privilege distribution between threads of different processes; gates are aimed at improving efficiency in access right verification and object sharing between threads of the same process.

- Handles are a generalization of the protected pointer concept, obtained by associating the name of an object with several keys. Handles are protected from forgery by key sparseness.
- Gates are an alternative, compact representation of access privileges reserving a single bit for each access right. Gates are sensitive data that must be protected from alteration and forgery. This is done by segregating them into private memory regions of the protection system and restricting their manipulation to the execution of the primitives of the protection system. The *Convert*() primitive makes it possible to transform a handle into an equivalent gate. Equivalency between gates and handles is defined in terms of inclusion of access rights. The *Verify*() primitive makes it possible to check a gate for the presence of a given set of access rights.

We have obtained the following results:

- Handles can be freely mixed in memory with ordinary data. Significant advantages arise in terms of the ease of programming and simplification of the memory layout of protected objects. Distribution of access privileges between threads of different processes is obtained by means of simple handle copy actions .

© 2009 ACADEMY PUBLISHER

These actions require no intervention by the protection system. The components of a given handle do not need to be stored in contiguous memory cells. The key granting a given access right on a given object can be stored separately from the object name, for example in the data area reserved for the execution of an operation permitted by this access right.

- With respect to solutions using protected pointers, handles reduce the space requirements of the information for memory management. By associating an object name with several keys, a single handle is necessary to certify possession of a complex access privilege expressed in terms of several access rights. The resulting improvements in memory costs are especially significant in a single-address-space environment, owing to the large size of object names that arise from the large size of memory addresses.
- Gates are an effective solution to the time performance problems that ensue in the execution of an operation on a given object, if a handle is used to certify possession of the access privilege necessary to accomplish this operation. High costs in terms of execution times are connected with the iterated actions of key-lock comparison that are necessary to validate the handle. In contrast, if a gate is used, the compact representation of the access right field permits a fast verification of access privilege possession.
- The additive behaviour of the *Convert*() primitive makes it possible to convert several handles into a single gate, if all these handles reference the same object. The resulting gate can be effectively used for object access by all the threads of the same process. In this way, gates effectively support object sharing within process boundaries.

The results presented in this paper have been evaluated from a number of points of view, including memory costs and the execution time overheads connected with protection. Owing to a lack of experimental results, the discussion has been largely based on a comparison with previous work.

Significant advantages are associated with the use of keys and locks to represent a protection system. These advantages are especially valuable as far as ease of access right management and object sharing between threads are concerned. Protected pointers have a negative impact on overall system performance. An effective implementation of a dualism of handles and gates could be an effective solution. In our opinion, access right representation techniques relying on forms of key-lock protection deserve fresh consideration. This especially applies to single-address-space environments, where the introduction of mechanisms for private object protection is mandatory. We hope that our work will have a significant impact in this direction.

REFERENCES

[1] R. Ashok, S. Chheda, C. A. Moritz, "Cool-Mem: combining statically speculative memory accessing with selective address translation for energy efficiency," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 2002, pp. 133–143.

[2] A. Bartoli, G. Dini, L. Lopriore, "Single address space implementation in distributed systems," *Concurrency: Practice and Experience*, vol. 12, no. 4 (April 2000), pp. 251–280.

[3] N. P. Carter, S. W. Keckler, W. J. Dally, "Hardware support for fast capability-based addressing," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1994, pp. 319–327.

[4] M. D. Castro, R. D. Pose, C. Kopp, "Password-capabilities and the Walnut kernel," *The Computer Journal*, vol. 51, no. 5 (September 2008), pp. 595–607.

[5] M. Cekleov, M. Dubois, "Virtual-address caches. Part 1: problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, no. 5 (September/October 1997), pp. 64–71.

[6] J. Chase, M. Feeley, H. Levy, "Some issues for single address space systems," *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, Napa, CA, October 1993, pp. 150–154.

[7] J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 4 (November 1994), pp. 271–307.

[8] L. Deller, G. Heiser, "Linking programs in a single address space," *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, USA, June 1999, pp. 283–294.

[9] G. Dini, L. Lopriore, "Sharing objects in a distributed, single address space environment," *Future Generation Computer Systems*, vol. 17, no. 3 (November 2000), pp. 247–264.

[10] A. Edwards, G. Heiser, "Components + security = OS extensibility," *Proceedings of the Sixth Australasian Computer Systems Architecture Conference*, Gold Coast, Australia, January 2001, pp. 27–34.

[11] D. M. England, "Capability concept mechanisms and structure in System 250," *Proceedings of the International Workshop on Protection in Operating Systems*, IRIA, Paris, 1974, pp. 63–82.

[12] V. D. Gligor, "Review and revocation of access privileges distributed through capabilities," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6 (November 1979), pp. 575–586.

[13] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, J. Liedtke, "The Mungi single-address-space operating system," *Software — Practice and Experience*, vol. 28, no. 9 (July 1998), pp. 901–928.

[14] B. Jacob, T. Mudge, "Software-managed address translation," *Proceedings of the Third International Symposium on High Performance Computer Architecture*, San Antonio, Texas, USA, February 1997, pp. 156–167.

[15] B. Jacob, T. Mudge, "Virtual memory: issues of implementation," *Computer*, vol. 31, no. 6 (June 1998), pp. 33–43.

[16] I. Kadayif, P. Nath, M. Kandemir, A. Sivasubramaniam, "Reducing data TLB power via compiler-directed address generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2 (February 2007), pp. 312–324.

[17] E. J. Koldinger, J. S. Chase, S. J. Eggers, "Architectural support for single address space operating systems," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, October 1992; in *SIGARCH Computer Architecture News*, vol. 20, special issue (October 1992), pp. 175–186.

© 2009 ACADEMY PUBLISHER

[18] C. S. Laih, L. Harn, J.-Y. Lee, “On the design of a single-key-lock mechanism based on Newton’s interpolating polynomial,” *IEEE Transactions on Software Engineering*, vol. 15, no. 9 (September 1989), pp. 1135–1137.

[19] B. Leslie, N. FitzRoy-Dale, G. Heiser, “Encapsulated user-level device drivers in the Mungi operating system,” *Proceedings of the Workshop on Object Systems and Software Architectures*, Victor Harbor, South Australia, Australia, January 2004, pp. 16–30.

[20] H. M. Levy, *Capability-Based Computer Systems*. Bedford, Mass.: Digital Press, 1984.

[21] A. Lindström, J. Rosenberg, A. Dearle, “The grand unified theory of address spaces,” *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995, pp. 66–71.

[22] L. Lopriore, “Capability based tagged architectures,” *IEEE Transactions on Computers*, vol. C-33, no. 9 (September 1984), pp. 786–803.

[23] L. Lopriore, “Protection in a single-address-space environment,” *Information Processing Letters*, vol. 76, no. 1–2 (November 2000), pp. 25–32.

[24] L. Lopriore, “Access control mechanisms in a distributed, persistent memory system,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 10 (October 2002), pp. 1066–1083.

[25] D. S. Miller, D. B. White, A. C. Skousen, R. Tcherepov, “Lower level architecture of the Sombrero single address space distributed operating system,” *Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems*, Dallas, TX, USA, November 2006, pp. 484–489.

[26] D. Mossop, R. Pose, “Security models in the Password-Capability System,” *Proceedings of the Tencon 2005 IEEE Region 10 Conference*, Melbourne, Australia, November 2005, pp. 1–6.

[27] E. I. Organick, *A Programmer’s View of the Intel 432 System*. New York: McGraw-Hill, 1983.

[28] J. Rosenberg, J. L. Keedy, D. Abramson, “Addressing mechanisms for large virtual memories,” *The Computer Journal*, vol. 35, no. 4 (August 1992), pp. 369–375.

[29] L. Rousseau, S. Natkin, “A framework of secure object system architecture,” *Proceedings of the Third Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, CA, February 1997, pp. 108–115.

[30] J. S. Shapiro, J. Vanderburgh, E. Northup, D. Chizmadia, “Design of the EROS trusted window system,” *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, 2004, pp. 165–178.

[31] F. G. Soltis, P. Conte, *Inside the AS/400: Featuring the AS/400E Series*, Second Edition. Loveland, CO: Duke Press, 1997.

[32] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, J. Lepreau, “The Flask security architecture: system support for diverse security policies,” *Proceedings of the Eighth USENIX Security Symposium*, Washington, DC, USA, August 1999, pp. 123–139.

[33] M. M. Swift, B. N. Bershad, H. M. Levy, “Improving the reliability of commodity operating systems,” *ACM Transactions on Computer Systems*, vol. 23, no. 1 (February 2005), pp. 77–110.

[34] O. S. Unsal, R. Ashok, I. Koren, C. M. Krishna, C. A. Moritz, “Cool-Cache: a compiler-enabled energy efficient data caching framework for embedded/multimedia processors,” *ACM Transactions on Embedded Computing Systems*, vol. 2, no. 3 (August 2003), pp. 373–392.

[35] P. Vasek, K. Ghose, “A comparison of two context allocation approaches for fast protected calls,” *Proceedings of the Fourth International Conference on High-Performance Computing*, Bangalore, India, December 1997, pp. 16–21.

[36] M. V. Wilkes, R. M. Needham, *The Cambridge CAP Computer and Its Operating System*. New York: North Holland, 1979.

[37] E. Witchel, J. Cates, K. Asanović, “Mondrian memory protection,” *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 2002, pp. 304–316.

[38] E. Witchel, S. Larsen, C. S. Ananian, K. Asanović, “Direct addressed caches for reduced power consumption,” *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, Austin, Texas, December 2001, pp. 124–133.

© 2009 ACADEMY PUBLISHER