Protection in Grasshopper: A Persistent Operating System

[†]Alan Dearle, [*]Rex di Bona, [*]James Farrow, [*]Frans Henskens,
[†]David Hulse, [*]Anders Lindström, [*]Stephen Norris,
[*]John Rosenberg and [†]Francis Vaughan

[*]Department of Computer Science
University of Sydney
N.S.W., 2006, Australia
{rex,matty,frans,anders,srn,johnr}
@cs.su.oz.au

[†]Department of Computer Science
University of Adelaide
S.A., 5001, Australia
(al,dave,francis}
@cs.adelaide.edu.au

Abstract

Persistent systems support a single storage abstraction in which all data may be created and manipulated in a uniform manner, regardless of its longevity. In such systems a protection mechanism is required to ensure that programs can access precisely those objects they are supposed to access and no others. In a monolingual system this protection can be provided by the type system of the programming language; in systems which support multiple persistent languages a separate protection mechanism must be supported. This paper describes the capability-based protection mechanism employed in Grasshopper, a new operating system specifically designed to support persistent systems on a conventional workstation platform. We show that this mechanism provides sufficient power and flexibility to handle a wide variety of protection scenarios.

1. Introduction

In this paper we describe the protection mechanism in Grasshopper, an operating system designed to support orthogonal persistence. The two basic principles of orthogonal persistence are that any object may persist (exist) for as long, or as short, a period as the object is required, and that objects may be manipulated in the same manner regardless of their longevity [3]. Persistent systems provide a fundamentally different computation paradigm to conventional systems. In a conventional system different mechanisms are provided for creating and accessing temporary data and permanent data (e.g. virtual memory and a file system). A persistent system, on the other hand, supports a single storage abstraction in which all data may be created and manipulated in a uniform manner, regardless of its longevity. Thus, programs may create data structures which outlive their execution and there is no need to write code to "flatten" data structures in order to store them in files.

A number of persistent systems have been constructed, most of which have been built above conventional operating systems, usually Unix [1,4,25,26,32]. Although these systems have been successful in terms of demonstrating the feasibility of persistence as a programming paradigm, efficiency has been a major problem. This is not surprising since they are being constructed above operating systems with a model that is fundamentally different from the persistence model. Other groups have developed specialised hardware in order to provide a more appropriate environment [9,17,33]. These groups have encountered difficulties because of the cost of building hardware using the latest technology and the problems associated with making the results of the research available to other research groups.

In Grasshopper [12,13] we have adopted a third approach which is to develop a new operating system on a conventional workstation platform. Some other research groups have also taken this route [5,6,10]. We see the advantages of this approach as:

- workstations are cheap and readily available,
- their performance is improving rapidly, and
- most research groups have access to these machines and so the results of our work can be easily made available.

Unfortunately, the use of workstations designed to support Unix does place some constraints on our design, particularly related to addressing issues. These problems have been discussed elsewhere [13]. Despite these difficulties we believe that it is possible to provide an efficient environment for persistent systems on conventional hardware.

An important issue in the design of a persistent system is the protection mechanism. Some form of protection is necessary for two reasons:

1. to ensure that programs can access precisely those objects they are supposed to access and no others, and
2. to restrict access to certain operating system functions.

In a monolingual persistent system built using a type-safe language the first category of protection can be provided by the programming language and type system [27]. However, Grasshopper is intended to be language independent. It is expected that persistent application systems employing different languages will run concurrently above the kernel. We therefore cannot rely on a single type system.

In most conventional operating systems access to operating system functions is controlled by a separate mechanism from that used to control access to data. Each user is given an access level and this determines the operations which may be performed. In Unix there are effectively only two such access levels, normal user and super-user. Just as we have argued that there should be a single data creation and manipulation mechanism for all data, it is sensible to have a single protection mechanism which provides all access controls. Such a uniform approach has been adopted on some object-based operating systems (e.g. Monads [22]) and is

employed in Grasshopper, which has a single protection mechanism controlling access to both data and operating system functions.

This paper concentrates on the issue of protection in the Grasshopper operating system. We begin with a description of the basic abstractions over storage and execution in Grasshopper and the protection requirements for these abstractions. We then provide some background on capabilities as a protection mechanism for persistent systems in general. This is followed by a description of the structure of capabilities in Grasshopper and a discussion of access rights and revocation. Finally we describe the operations supported for the manipulation of capabilities in Grasshopper.

2. Grasshopper Basic Abstractions

In this section we describe the two basic abstractions in Grasshopper. The abstraction over storage is the *container* and the abstraction over execution is the *locus*. Conceptually, each locus executes within a single container, its *host container*. Containers are not virtual address spaces. They may be of any size, including larger than the virtual address range supported by the hardware. The data stored in a container is supplied by a *manager*. Managers are responsible for maintaining a consistent and recoverable stable copy of the data represented by the container. The use of managers, which is vital to the removal of the distinction between persistent and volatile storage, is beyond the scope of this paper and is discussed in [12].

2.1 Containers

Containers are the only storage abstraction provided by Grasshopper; they are persistent entities which replace both address spaces and file systems. In most operating systems, the notion of a virtual address space is associated with an ephemeral entity, a process, which accesses data within that address space. In contrast, containers and loci are orthogonal concepts. A Grasshopper system consists of a number of containers which may have loci executing within them. At any moment in time, a locus can only address the data visible in the container in which it is executing. Of course, there must be facilities which allow the transfer of data between containers. The mechanisms provided in Grasshopper are *mapping* and *invocation*.

The purpose of container mapping is to allow data to be shared between containers. This is achieved by allowing data in a region of one container to appear (either as read-only or read-write) in another container. In its simplest form, this mechanism provides shared memory and shared libraries similar to that provided by conventional operating systems. However, conventional operating systems restrict the mapping of memory to a single level. Both VMS [24] and variants of Unix (such as SunOS) provide the ability to share memory segments between process address spaces, and a separate ability to map from disk storage into a process address space. Several other systems [7, 8, 28, 31] provide the notion of a *memory object*, which provides an abstraction over data. In these systems, memory

objects can be mapped into a process address space, however memory objects and processes are separate abstractions. It is therefore impossible to directly address a memory object, or to compose a memory object from other memory objects.

By contrast, the single abstraction over data provided by Grasshopper may be arbitrarily (possibly recursively) composed. Since any container can have another mapped onto it, it is possible to construct a hierarchy of container mappings as shown in Figure 1. The hierarchy of container mappings forms a directed acyclic graph. The restriction that mappings cannot contain circular dependencies is imposed to ensure that one container is always ultimately responsible for data. In Figure 1, container *C2* is mapped onto container *C1* at location *a1*. In turn, *C2* has regions of containers *C3* and *C4* mapped onto it. The data from *C3* is visible in *C1* at address *a3*, which is equal to *a1 + a2*.

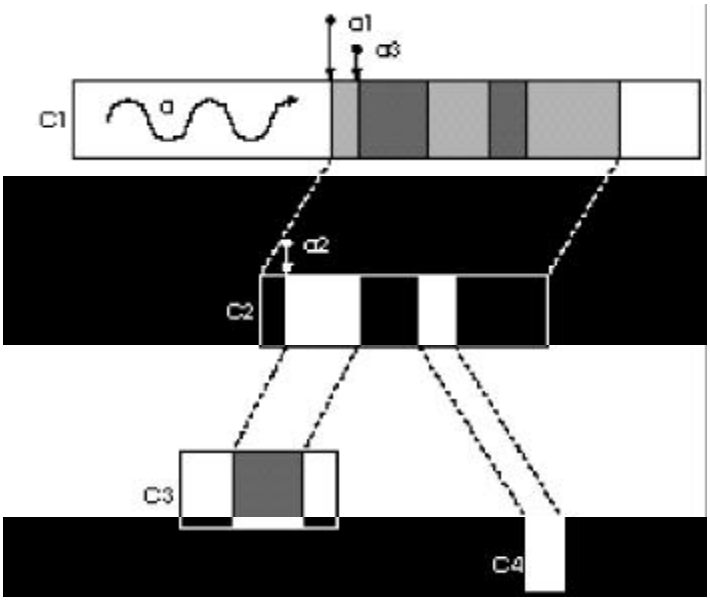

Figure 1: A container mapping hierarchy

Loci perceive the address space of their host container. Therefore, all loci executing within a container share the same address space. However, a locus may require private data, which is visible to it, yet invisible to other loci that inhabit the same container. To satisfy this need, Grasshopper provides the notion of a *locus private mapping*.

Locus private mappings are visible only to the locus which created them and take precedence over host container mappings. This allows, for example, each locus to have its own stack space with the stacks of all loci occupying the same address range within the host container. This technique both simplifies multi-threaded programming and provides a useful security mechanism that is unavailable with conventional addressing mechanisms.

2.2 Loci

In its simplest form, a locus is simply the contents of the registers of the machine on which it is executing. Like containers, loci are maintained by the Grasshopper kernel and are inherently persistent. Making loci persistent is a departure from other operating system designs and frees the programmer from much complexity [22].

A locus is associated with a container, its host container. The locus perceives the host container's contents plus any containers mapped by locus private mappings within its own address space. Virtual addresses generated by the locus map directly onto addresses within the host container and the locus private mapped containers. A container comprising program code, mutable data and a locus forms a basic running program. Loci are an orthogonal abstraction to containers. Any number of loci may execute within a given container; this allows Grasshopper to support multi-threaded programming paradigms.

2.3 Inter-Container Communication

An operating system is largely responsible for the control and management of two entities: objects, which contain data (containers); and processes (loci), the active elements which manipulate these objects. One of the most important considerations in the design of an operating system is the model of interaction between these entities. Grasshopper uses the procedure-oriented model in which communication is achieved via procedure calls and processes move between entities [23]. Thus a locus may *invoke* a container thereby changing its host container.

Any container may include, as one of its attributes, a single entry point known as an *invocation point*. When a locus invokes a container, it begins executing code at the invocation point. The single invocation point is important for security; it is the invoked container that controls the execution of the invoking locus by providing the code that will be executed.

A locus may invoke and return through many containers in a manner similar to conventional procedure calls. The Grasshopper kernel maintains a call chain of invocations between containers. Implicitly each locus appears to be rooted in the container representing the kernel: when a locus returns to this point it is deleted. However some loci may never need to return to the container from which they were invoked; such a locus may meander from container to container. In such circumstances, a parameter to the invoke system call allows the locus to inform the kernel that no call chain need be maintained.

Access to operating system functions is also achieved by invocation. This provides a uniform interface for applications and blurs the distinction between system and user functions.

2.4 Protection Requirements

In the previous sections we have described the basic abstractions in Grasshopper and the operations over these abstractions. Given that containers are the *only* abstraction over storage, some access control mechanisms are required. These include control over:

- which containers may be invoked
- the setting of an invocation point
- which containers may be mapped and the type of access one has to the mapped region, e.g. read/write
- the creation and destruction of containers

Similarly, it is desirable to have control over loci. The control required includes:

- control over the creation of locus private mappings
- the ability to block and unblock loci
- management of locus exceptions
- control over the creation and destruction of loci.

In a conventional operating system many of these controls are provided by the file system, which maintains access lists, usually on a hierarchical basis. This is not appropriate in Grasshopper since there is no file system. Some persistent systems use the type system to provide control over access, however, as we have stated earlier, we propose to support multiple languages with different type systems, and so this is not an alternative. For these reasons we believe that it is essential for Grasshopper to support a third abstraction: a protection mechanism. That abstraction is capabilities and in the following section we provide some background and justification for this choice.

3. Capabilities as a Protection Mechanism

Capabilities were first proposed by Dennis and Van Horn [14] as a technique for describing the semantics of controlled access to data. The idea was extended by Fabry who proposed a computer system based on capabilities [15]. There have been several capability-based systems constructed. Some of these enlisted hardware support [30,33,35], others were purely software implementations [29,36]. Although these systems differ greatly, the fundamental principles of capability-based access control are the same.

The basic idea is that access to objects is controlled by the ownership and presentation of capabilities. That is, in order for a program to access an object it must produce a capability for that object. In this sense capabilities may be viewed as keys which unlock the object to which they refer. Since the possession of a

capability gives an undeniable right to access an object it is important that programs are unable to access data for which no authorisation is held. A capability for an object can only be obtained by creating a new object or by being passed a capability by another program holding that capability.

There are three well-known techniques for achieving this requirement:

tagging:	in which extra bits are provided by the hardware to indicate memory regions representing capabilities and to restrict access to them,
passwords:	in which a key, embedded in a sparse address space, is stored with the entity and a matching key must be presented to gain access to that entity, and
segregation:	in which capabilities are stored in a protected area of memory.

In all of the above methods, capabilities have three components: a unique name identifying some entity, a set of access rights related to that entity, and rights pertaining to the capability itself, for example, whether the capability can be copied. Capability systems use entity names which are unique for the life of the system, that is, the name given to an entity will never be re-used, even if the entity is deleted. This avoids aliasing problems and provides a means of trapping dangling references. Such unique names may be generated by using a structured naming technique where each machine is given a unique name and each entity created on that machine has a unique name [18,19].

Although the ownership of a capability guarantees the right to access the corresponding entity, the access rights field may restrict the level of access allowed. The facilities provided by access rights vary greatly between different capability systems. They may be as simple as read, write and execute, or they may be based on the semantics of the different objects, for example a list of procedures for accessing an abstract data type. When a capability is presented in order to access an object, the system checks that the type of access does not conflict with that allowed in the capability. There is usually an operation which allows a new capability to be created from an existing one with a subset of the access rights. This allows for the construction of restricted views.

The third component of a capability contains status bits which indicate which operations can be performed on the capability itself. Again, these vary greatly. The minimum usually provided is a *no copy* bit which restricts the copying of the capability, perhaps on a per user basis. This may be used to stop some user from passing a capability on to other users, i.e. to limit propagation. Other status bits may include a *delete* bit which allows the holder of the capability to delete the object.

A further facility provided on some capability systems is the ability to revoke access. That is, after giving a program a capability it may be desirable at a later time to revoke this capability. Implementation of revocation is not easy. The

simplest technique is to change the unique name of the object. This will effectively invalidate all existing capabilities. Selective revocation may be supported by using indirection through an owner-controlled table of access rights or by providing multiple names for the object which can be individually invalidated.

Capabilities provide a uniform model for controlling access of data. However, entry to the system itself, by logging on, must in the end be based on some form of password. An advantage of capability-based systems is that, even if the password system is broken, there need not be any single password which provides access to all data of the system. That is, there need not be a super-user.

In summary we see the major advantages of capabilities as a protection mechanism as being:

- unique naming of entities, avoiding aliasing problems
- flexibility, in that a number of different protection paradigms may be implemented
- restricted access to entities may be supported
- revocation allows lifetime control over access
- avoidance of the need for a super-user

For all of these reasons capabilities are supported as the basic protection abstraction in Grasshopper.

4. Capabilities in Grasshopper

The basic access and protection mechanism in Grasshopper is the capability. In order to perform any operation on a container or locus an appropriate capability must be presented. From an abstract point of view, capabilities in Grasshopper have five fields as shown in Figure 2. The unique name identifies the entity to which this capability grants access. The category defines the kind of the entity represented. The categories supported include containers, loci and devices. However, it is anticipated that there will be additional categories supported in the final system.

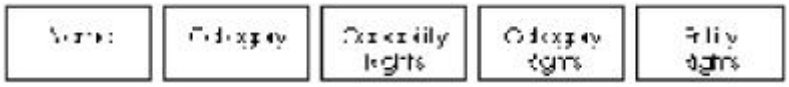

Figure 2: The logical structure of a capability in Grasshopper

The next three fields define the access granted by the capability. The capability rights indicate rights relating to the capability itself while the category rights relate to operations on a particular entity category. The capability and category rights are described in section 4.4.

The entity rights are uninterpreted by the system and are meaningful only to the particular entity referenced by the capability. They are stored in a secure manner by the kernel with the other access rights and are passed as an implicit parameter on invocation. They may be used for a variety of purposes; for example, they could be used as tags to represent a primitive type system allowing a container to implement a domain specific protection mechanism. Alternatively, they may be used as an identifier; a device manager may represent different physical devices using capabilities that differ only in their entity rights field. In this case the entity rights field represents the physical device. They may also be used to describe the level of service available from an invoked container using this capability. This could be implemented as a bit list, with one bit for each operation provided by the container. Such an arrangement would allow the construction of capabilities with different views of a container. Alternatively, the entity rights could be used as a tag to identify the capability used to effect an invoke. This would allow some form of accounting for the service provided. In both of these case the kernel does not interpret the meaning of the entity rights; it simply stores them and makes them available as part of the invoke mechanism.

The two major issues in the design of a capability system are the naming scheme and the method used to protect the capabilities themselves. The allocation of unique names to entities and the method used to locate these entities is beyond the scope of this paper. However, as will be shown in the following sections, these unique entity names are not directly visible to users of the system and so we have considerable flexibility in the design of the low-level naming scheme.

As we have described in Section 3 there are three basic techniques for protecting capabilities, namely tagging, passwords and segregation. The merits of each of these have been well discussed in the literature [2,16,21,34]. Given that Grasshopper is to be implemented on conventional hardware, tagging is not an option. Password capabilities have the advantage that they may be embedded within applications and require no special software to protect them. However, it is precisely this feature which makes them less appealing. Since the kernel cannot know how many (if any) capabilities for an entity exist at any point in time, it cannot perform garbage collection and must rely on explicit destruction of entities or some form of aging [2].

In a segregated system the kernel always knows how many capabilities exist for an entity. Using segregated capabilities allows garbage collection to be performed in association with explicit destruction of entities by loci. Reference counts may be maintained and, when the reference count on a capability falls to zero (i.e., there are no more extant references to the corresponding entity) the entity may be deleted. For these reasons segregated capabilities are used in Grasshopper.

4.1 Association of Capabilities with Entities

In Grasshopper, capabilities provide access to entities and also control the level of access to those entities. Since the Grasshopper system uses a segregated capability system, capabilities are protected from direct manipulation by programs.

Associated with each entity, but protected from user access, are two tables known as the *permission group table* and the *capability table*. Capabilities owned by an entity reside in its capability table whilst control over access to an entity is effected by entries within its permission group table. Capability table entries are indexed by fixed length keys and refer to permission group entries which contain sets of permissions; together these tables implement the Grasshopper capability regime.

Loci may only use capabilities owned by themselves or their host container; these capabilities are accessed via the presentation of a *capability reference* or *capref*. Caprefs are the way in which capabilities appear to the application programmer. A capref comprises a pair consisting of a flag and a key. The flag specifies whether the key should be interpreted with respect to the locus or its host container. The key refers to an entry in the selected entity's capability table. Caprefs are not protected by the system and may be constructed arbitrarily. This does not constitute a security risk since a useful capref can only refer to those capabilities legitimately held by the host container or locus.

On presentation of a capref, the key is looked up in the selected capability table. Assuming a match is found, the permission group field of the selected capability table entry is used to identify the entity being referenced and the permission group containing the permissions associated with that capability.

In Figure 3 the container *C1* is the host container of locus *L1*. Two caprefs, *CR1* and *CR2* are stored within Container *C1* and are therefore addressable by locus *L1*. When presented by *L1*, capref *CR1* refers to a capability in *L1*'s capability table whereas *CR2* refers to the capability in the *L1*'s host container, *C1*. In both cases the capabilities refer to the entity C2. However, since they refer to different permission group entries, they may have different protections associated with them.

On creation of an entity, a single permission group called the *master permission group*, granting all access, is also created and a capability referring to it is returned. Appropriate operations are provided for creating new permission groups with reduced access; these are discussed later in the paper. Such groups are called *derived permission groups* and capabilities referring to these are called *derived capabilities.*

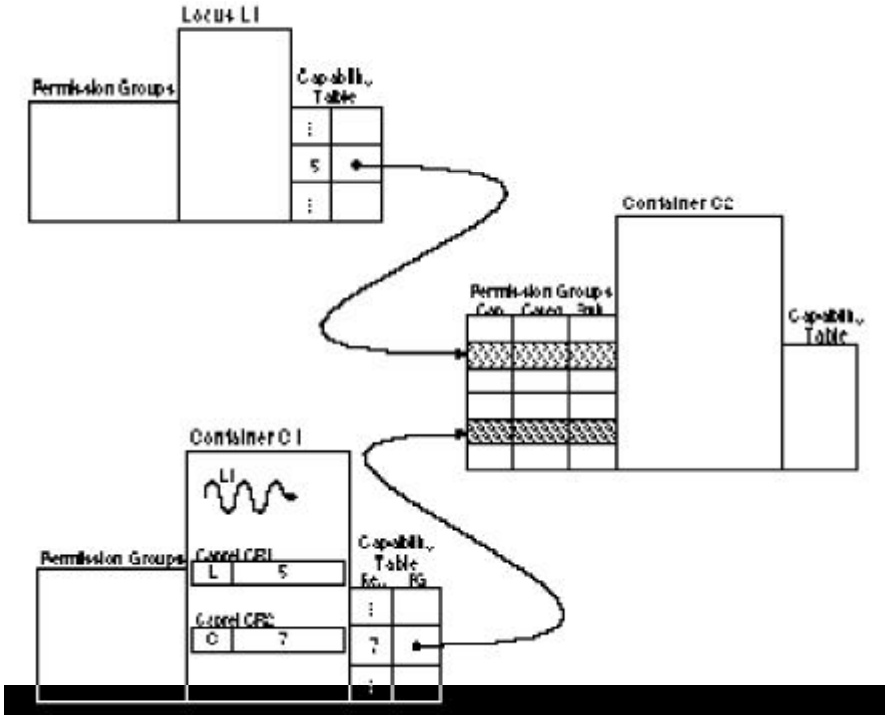

Figure 3: Permission groups, capability tables and caprefs

There are two distinct advantages of this structure. First, it makes no assumptions about the relationships between capabilities. The kernel simply maintains the capabilities in a table in a secure manner. Arbitrary naming schemes and structures can be constructed above the kernel. For example, it would be possible to build a hierarchical naming scheme similar to that provided by Unix. Alternatively, more flexible naming schemes such as those described in [11, 20] could be implemented. Indeed, alternative naming schemes may coexist within a running Grasshopper system. Second, the separation of the capabilities from the permission groups provides considerable flexibility, particularly in relation to revocation of access. This is discussed in Section 4.3.

4.2 Access Rights

In this section we summarise the access rights supported by the Grasshopper capability system. These access rights are stored in permission groups and define the operations that may be performed using a corresponding capability. All system functions (i.e. mapping, invocation, etc.) are controlled by capabilities and require the presentation of capabilities with appropriate access permissions.

As we have indicated above, there are three sets of rights: capability rights, category rights and entity rights. The first two groups of rights, capability and category rights, are defined by the Grasshopper kernel and are collectively referred

to as kernel rights. The last group, entity rights, are not interpreted by the kernel, but are held in a secure manner in the permission groups.

4.2.1 Capability Rights

The capability rights apply to all categories of entities and control the operations on the capability tables and permission groups. The capability rights are:

- *destroy* – the right to destroy the corresponding entity
- *copy permission group* – the right to create a copy of a permission group for an entity
- *delete permission group* – the right to delete a permission group
- *reduce kernel rights* – the right to reduce the kernel rights in an existing permission group
- *modify entity rights* – the right to modify the entity rights in an existing permission group
- *derive kernel rights* – the right to derive a new permission group from an existing permission group with equal or reduced kernel rights
- *derive entity rights* – the right to derive a new permission group from an existing permission group, possibly with modified entity rights
- *inject* – the right to insert new capabilities into an entity's capability table

The first right allows the destruction of an entity. This is effectively achieved by deleting all of the permission groups for the entity. The next two rights control copying and deletion of permission groups. The following four rights permit the modification of the rights in a permission group and the creation of new permission groups with modified access rights. The separation of control over manipulation of kernel rights and entity rights reflects the fact that entity rights are uninterpreted by the kernel. The meaning of kernel rights is universally known. On the other hand, only programs which understand the format of the entity rights for a particular entity can sensibly modify these. It is thus necessary to be able to separately restrict the ability to manipulate entity rights to appropriate programs.

There are no rights which control the manipulation of capability tables. Loci can always refer to their own capability table and the capability table of their current host container. The *inject* right controls access to other entity's capability tables. This is further discussed in Section 4.4.

4.2.2 Category Rights

The category rights apply to specific categories of entities. However, there are some common category rights which apply to both containers and loci. These are:

- *alter read-only mappings* – the right to define and remove read-only mappings of containers into this entity
- *alter read-write mappings* – the right to define and remove read-write mappings of containers into this entity
- *alter exception handler* – the right to define and remove an exception handler for this entity

The container category rights are:

- *map read-only* - the right to map this container into another entity with read-only access
- *map read-write* - the right to map this container into another entity with read-write access
- *invoke* - the right to invoke this container
- *set invocation point* - the right to modify the invocation point for this container

The locus category rights are:

- *raise exception* - the right to raise an exception for this locus
- *block/unblock* - the right to control the scheduling of this locus

There are two sets of mapping rights. The first set, alter read-only/read-write mappings, defines whether the holder of the capability is allowed to map containers into the corresponding entity. The second set applies to containers and indicate whether the holder of the capability is permitted to map the specified container into other containers and loci. For example, in order to map container A into container B with read-write access, a locus must have a capability for A with at least *map read-write* rights and a capability for container B with at least *alter read-write mappings* right.

Grasshopper supports a concept of exception handlers. There are several rights associated with these, however they are beyond the scope of this paper. They are included here for completeness.

As described earlier, Grasshopper provides an invocation mechanism which permits loci executing in different host containers to communicate. A capability with the *invoke* right is required in order to invoke another container. Invocation causes the locus to begin execution at the invocation point of the invoked container. The invocation point may be changed by the holder of a capability with the *set invocation point* right.

Finally, the *block/unblock* right allows the holder of such a capability to control the execution of a locus. This may be used to implement higher level synchronisation and scheduling mechanisms in a controlled manner.

4.3 Permission Groups and Revocation

One of the most powerful features of Grasshopper protection system is support for revocation. It is possible to grant access to some entity by the provision of a capability and then to revoke this access at a later stage. Revocation is achieved in Grasshopper via the permission group mechanism.

Recall that a capability effectively consists of the name of an entity and the identification of one of the permission groups associated with that entity. Access may be revoked by deleting the permission group with which the capability is associated. Any future use of the capability will result in an exception because the permission group no longer exists. Such draconian measures are not always required; Grasshopper therefore also supports the ability to reduce the access rights for the permission group, thereby reducing the operations allowed.

The mechanism described above, combined with the ability to copy permission groups and to derive new permission groups, provides a powerful and flexible protection paradigm. The set of permissions groups associated with an entity form a tree. Copying a permission group creates a new sibling, whilst derivation creates a new child. This is similar to a scheme for password capabilities described by Anderson [2].

The properties of permission group trees are simple:

- The access rights available in any permission group are always greater than or equal to those available from any descendant of that permission group.
- If any permission group is deleted then all of its descendants are also deleted.
- If the access rights in a permission group are reduced then all its descendants are similarly reduced.

Consider as an example a class of students to whom we wish to give access to some entity required for an assignment in such a way that it is possible to revoke access for any individual student, or for all students, e.g. when the assignment is due. This can be achieved by creating a new permission group and then deriving a permission group below this for each student. Providing a capability for each permission group is held by the person in charge, then an individual student's access may be revoked by deleting that student's permission group and access for all students may be revoked by deleting the permission group from which they were derived.

4.4 Operations on Capabilities

In the previous sections we have referred to various operations for manipulating and copying capabilities and permission groups. In this section we provide type definitions for the various data structures along with a description of each of the

operations. The structure of the types *entity_rights_type* and *cap_key_type* is implementation dependent and is not important to the discussion which follows. The notation *capability*(*x*) is used to refer to the capability found in the capability table indicated by *x.cr_flag* in the location indicated by *x.cr_key*.

Types

permissions **is structure** (capability_rights_type *capability_rights*;
category_rights_type *category_rights*;
entity_rights_type *entity_rights*)

permission_group **is structure** (permissions *pg_permissions*)

capability **is structure** (cap_key_type *cap_key*;
permission_group *cap_pg*)

caplist_selector **is enum** (locus, host_container)

capref **is structure** (caplist_selector *cr_flag*;
cap_key_type *cr_key*)

Operations

copy_cap (capref *source, destination*)

delete_cap (capref *target*)

copy_pg (capref *source, destination*)

reduce_pg (capref *target*; permissions *new_permissions*)

delete_pg (capref *target*)

derive_pg (capref *source, destination*; permissions *new_permissions*)

delete_entity (capref *target*)

inject_cap (capref *source, destination*; cap_key_type *new_cap_key*)

The **copy_cap** operation allows capabilities to be copied between and within the currently accessible capability tables (i.e. current locus table and current host container table). The capability referred to by *source* is copied to the capability table indicated by *destination.cr_flag* in the location indicated by *destination.cr_key*. Note that this operation does not affect the permission groups. After a **copy_cap** operation, both *source* and *destination* refer to the same permission group.

The **delete_cap** operation removes the entry referred to by *target.cr_key* in the table indicated by *target.cr_flag*. Again, permission groups are not affected by this operation.

The **copy_pg** operation creates a new permission group for the entity indicated by *capability*(*source*). The new permission group has the same permissions as the source. A new capability is created and is stored in *capability*(*destination*) . This capability refers to the new permission group. The new permission group is a sibling to *capability*(*source*).*cap_pg*, in the permission group tree; revocation of *capability*(*source*) does not revoke *capability*(*destination*). Similarly, revocation of *capability*(*destination*) does not revoke *capability*(*source*).*cap_pg*. *Capability*(*source*).*cap_pg*.*pg_permissions* must include *copy permission group* for this operation to take place.

The capability copy operations are used for two main purposes. First, they allow the re-organisation of the capabilities in a capability table. Second, they may be used to pass and return capability parameters on invocations, i.e. capabilities to be passed to a container on an invocation are copied to the locus' capability table prior to the invocation. These capabilities may then be accessed by the locus in the invoked container. Capabilities may be returned by the same mechanism. Notice that the allocation and management of keys is the responsibility of the code executing in the container. Appropriate library routines are provided for this purpose.

The **reduce_pg** operation replaces *capability*(*target*).*cap_pg*.*pg_permissions*, with *new_permissions*. All permission groups below *capability*(*target*).*cap_pg* in the permission group tree are reduced in the same way. *Capability*(*target*).*cap_pg*.*pg_permissions* must include appropriate permissions (*reduce kernel rights* and/or *modify entity rights*) for this operation to take place.

Delete_pg deletes the permission group *capability*(*target*).*cap_pg*, and any permission groups below this one in the permission group tree. Any capability referring to any of the deleted permission groups will be invalid following this operation. *Capability*(*target*).*cap_pg*.*permissions* must include *delete permission group* for this operation to take place.

The **derive_pg** operation is similar to **copy_pg**, but creates the new permission group as a child in the permission group tree, possibly with reduced permissions. The new permission group is created below *capability*(*source*).*cap_pg* in the permission tree, and with permissions *new_permissions*. A new capability is created as *capability*(*destination*) and this capability points at the new permission group. Since the new permission group is a child of *capability*(*source*).*cap_pg*, revocation of *capability*(*source*) will cause revocation of *capability*(*destination*). *Capability*(*source*).*cap_pg*.*pg_permissions* must include *derive kernel rights* and *derive entity rights* (assuming both are modified) for this operation to take place.

The permission group operations provide control over the construction of the permission group tree in order to allow revocation as discussed in Section 4.3. They also allow for the construction of restricted views of an entity by appropriate use of the entity permissions.

Delete_entity deletes all permission groups relating to the entity referred to by *capability*(*target*). As a result, all capabilities referring to the entity are invalidated, making the entity inaccessible and effectively deleting it. *Capability*(*target*).*cap_pg*.*permissions* must include *destroy* for this operation to take place.

The final operation, **inject_cap**, is the only operation which can access a capability table other than the current locus and current host container tables. *Capability*(*source*) is copied to the capability table of the entity referenced by *capability*(*destination*) in the location indicated by *new_cap_key*. This operation does not affect the permission groups; both source and the new capability point at the same permission group. *Capability*(*destination*).*cap_pg*.*permissions* must include *inject* for this operation to take place.

The inject operation is particular useful for populating a new entity with some initial capabilities. For example, a new locus may be created and given some capabilities for basic system functions such as input-output.

5. Conclusions

In this paper we have described the protection mechanism for the Grasshopper operating system. The fact that this mechanism is based on capabilities results in a number of advantages:

1. The system does not enforce any particular naming or protection paradigm. The mapping from meaningful names to capabilities is managed outside the kernel. Thus it is possible to construct arbitrary naming schemes.
2. The creator of an entity has full control over the level of access provided to other users.
3. Arbitrarily restricted views of entities may be constructed using the entity rights field of permission groups.
4. Access to entities may be selectively revoked.
5. By the use of unique names and explicit deletion the need for garbage collection across the entire store is avoided.

A secondary advantage of our approach to capabilities relates to the scheme used to provide unique names for capabilities. This naming scheme is not visible outside the kernel; applications always use caprefs to refer to entities. This leaves considerable flexibility in the design of the kernel entity naming scheme and also permits distribution to be completely transparent.

Although it has been argued in the past that capabilities are an expensive mechanism, this has been in an environment where capabilities are used for all addressing. It will be noted that in Grasshopper, capabilities are only used for validating course-grain operations such as invocation and mapping. Normal

memory accesses are directly handled by the conventional virtual memory hardware. It is therefore expected that the proposed scheme will be no more expensive than protection mechanisms provided by existing operating systems and may well be more efficient.

The scheme described in this paper has been implemented in a prototype version of Grasshopper on DEC Alpha machines. This prototype system is already capable of executing simple user programs and it is expected that a fully usable version of the system will be available later this year.

Acknowledgments

The work described in this paper is supported by Australian Research Council grant A49130439 and by an equipment grant under the Alpha Innovators Program from Digital Equipment Corporation.

References

1. Albano, A., Cardelli, L. and Orsini, R. "Galileo: A Strongly Typed, Interactive Conceptual Language", ACM Transactions on Database Systems, 10(2), pp. 230-260, 1985.

2. Anderson, M., Pose, R. and Wallace, C. S. "A Password-Capability System", *The Computer Journal*, vol 29, 1, pp. 1-8, 1986.

3. Atkinson, M. P., Bailey, P., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, 26(4), pp. 360-365, 1983.

4. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, 17(7), pp. 24-31, 1981.

5. Campbell, R. H., Johnston, G. M. and Russo, V. F. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems", *ACM Operating Systems Review*, 21(3), pp. 9-17, 1987.

6. Chase, J. S., Levy, H. M., Baker-Harvey, M. and Lazowska, E. D. "Opal: A Single Address Space System for 64-Bit Architectures", *Third IEEE Workshop on Workstation Operating Systems*, IEEE, 1992.

7. Cheriton, D. R. "The V Kernel: A Software Base for Distributed Systems", *Software*, 1(2), pp. 9-42, 1984.

8. Chorus Systems "Overview of the CHORUS Distributed Operating Systems", *Computer Systems - The Journal of the Usenix Association*, 1(4), 1990.

9. Cockshott, W. P. "Design of POMP - a Persistent Object Management Processor", *Proceedings of the Third International Workshop on Persistent Object Systems*, (ed J. Rosenberg and D. M. Koch), Springer-Verlag, pp. 367-376, 1989.

10. Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. "The Clouds Distributed Operating System", *Proceedings, 8th International Conference on Distributed Computing Systems*, 1988.

11. Dearle, A. "Environments: A Flexible Binding Mechanism to Support System Evolution", *Proc. 22nd Hawaii International Conference on System Sciences*, vol II, pp. 46-55, 1989.

12. Dearle, A., di Bona, R., Farrow, J. M., Henskens, F. A., Lindström, A., Rosenberg, J. and Vaughan, F. "Grasshopper: An Orthogonally Persistent Operating System", *Computer Systems (to appear)*, 1994.

13. Dearle, A., Rosenberg, J., Henskens, F. A., Vaughan, F. and Maciunas, K. "An Examination of Operating System Support for Persistent Object Systems", *Proceedings of the 25th Hawaii International Conference on System Sciences*, vol 1, (ed V. Milutinovic and B. D. Shriver), IEEE Computer Society Press, Hawaii, U. S. A., pp. 779-789, 1992.

14. Dennis, J. B. and Van Horn, E. C. "Programming Semantics for Multiprogrammed Computations", *Communications of the A.C.M.*, 9(3), pp. 143-145, 1966.

15. Fabry, R. S. "Capability-Based Addressing", *Communications of the A.C.M.*, 17(7), pp. 403-412, 1974.

16. Gehringer, E. F. and Keedy, J. L. "Tagged Architecture: How Compelling are its Advantages?", *Twelth International Symposium on Computer Architecture*, pp. 162-170, 1985.

17. Harland, D. M. "REKURSIV: Object-oriented Computer Architecture", Ellis-Horwood Limited, 1988.

18. Henskens, F. A. "A Capability-based Persistent Distributed Shared Memory", PhD Thesis, University of Newcastle, N.S.W., Australia, ISBN 0 86758 668 0, 1991.

19. Henskens, F. A. "Addressing Moved Modules in a Capability-based Distributed Shared Memory", *Proceedings of the 25th Hawaii International Conference on System Sciences*, vol 1, (ed V. Milutinovic and B. D. Shriver), IEEE Computer Society Press, Hawaii, U. S. A., pp. 769-778, 1992.

20. Hitchens, M. and Rosenberg, J. "Binding between Names and Objects in a Persistent System", *Proceedings of 2nd International Workshop on Object Orientation in Operating Systems*, IEEE, Dourdan, France, pp. 26-37, 1992.

21. Keedy, J. L. "An Implementation of Capabilities without a Central Mapping Table", *Proc. 17th Hawaii International Conference on System Sciences*, pp. 180-185, 1984.

22. Keedy, J. L. and Vosseberg, K. "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System", *Proceedings of the 25th Hawaii International Conference on Systems Sciences*, vol 1, IEEE, Hawaii, USA, pp. 747-756, 1992.

23. Lauer, H. C. and Needham, R. M. "On the Duality of Operating System Structures", *Operating Systems Review*, 13(2), pp. 3-19, 1979.

24. Levy, H. M. and Lipman, P. H. "Virtual Memory Management in the VAX/VMS Operating System", *Computer*, 15(3), pp. 35-41, 1982.

25. Matthes, F. and Schmidt, J. W. "The Type System of DBPL", *Proceedings of the Second International Workshop on Database Programming Languages*, Morgan Kaufmann, pp. 219-225, 1989.

26. Morrison, R., Brown, A. L., Conner, R. C. H. and Dearle, A. "Napier88 Reference Manual", Universities of Glasgow and St. Andrews, Persistent Programming Research Report PPRR-77-89, 1989.

27. Morrison, R., Brown, A. L., Connor, R. C. H., Cutts, Q. I., Dearle, A., Kirby, G., Rosenberg, J. and Stemple, D. "Protection in Persistent Object Systems", *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, Springer-Verlag, Bremen, Germany, pp. 48-66, 1990.

28 Moss, J. E. B. "Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach", *Proceedings of the Second International Workshop on Database Programming Languages*, Gleneden Beach, Oregon, Morgan Kaufmann, pp. 358-374, 1989.

29. Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R. and van Staveren, H. "Amoeba: A Distributed Operating System for the 1990s", *Computer*, 23(5), pp. 44-53, 1990.

30. Pose, R. D. "Capability Based, Tightly Coupled Multiprocessor Hardware to Support a Persistent Global Virtual Memory", *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, (ed B. D. Shriver), pp. 36-45, 1989.

31. Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, ACM Order Number 556870, pp. 31-39, 1987.

32. Richardson, J. E. and Carey, M. J. "Implementing Persistence in E", *Proceedings of the Third International Workshop on Persistent Object Systems*, (ed J. Rosenberg and D. M. Koch), Springer-Verlag, pp. 175-199, 1989.

33. Rosenberg, J. and Abramson, D. A. "MONADS-PC: A Capability Based Workstation to Support Software Engineering", *Proc, 18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

34. Tanenbaum, A. S. "Experiences with the Amoeba Distributed System", *Communications of the ACM*, 33(12), pp. 46-63, 1990.

35. Wilkes, M. V. and Needham, R. M. "The Cambridge CAP Computer and its Operating System", North Holland, Oxford, 1979.

36. Wulf, W. A., Levin, R. and Harbison, S. P. "HYDRA/C.mmp: An Experimental Computer System", McGraw-Hill, New York, 1981.