Protection Domain Extensions in Mungi[*]

Jerry Vochteloo, Kevin Elphinstone, Stephen Russell, and Gernot Heiser[†]

School of Computer Science and Engineering
The University of New South Wales, Sydney, Australia 2052

Abstract

The Mungi single address space operating system provides a protected procedure call mechanism named protection domain extension *(PDX). The PDX call executes in a protection domain which is the union of (a subset of) the caller's domain, and a fixed domain associated with the procedure. On return, the caller's original protection domain is re-established. Extensive caching of validation data allows amortisation of setup costs over a possibly large number of invocations. The PDX mechanism forms the basis for object support in Mungi, particularly encapsulation. It is also used for accessing devices, and to implement user-level page fault handlers and other services.*

1. Introduction

One of the most attractive features of object-oriented operating systems is the ability of users to transparently extend the OS. Such extensibility is of particular interest if users can access methods provided by other users without compromising security. Hence, the system should efficiently support object encapsulation and safe method invocation.

Capability systems are particularly well-suited to support extensibility [Lev84]. Safe method invocation in these systems is made possible by the provision of a *protected procedure call* mechanism, which allows the callee to perform operations the system would not permit the caller to do directly.

Mungi [HERV94] is a 64-bit single address space operating system (SASOS) based on password capabilities. Mungi's protected procedure mechanism is called *protection domain extension* (PDX). This paper describes Mungi's PDX mechanism and its implementation.

[*]This work was supported by Australian Reseach Council grant A49330285.

[†]Phone: +61-2-9385-5156, fax: +61-2-9385-5995, e-mail: disy@cse.unsw.edu.au, www: http://www.cse.unsw.edu.au/˜disy

2. Protection Domains in Mungi

Mungi's basic protection model has been described in [VRH93]. In short, each *task* (which consists of one or more *threads*) has associated with it a *protection domain*, which is the set of objects accessible to the task. The protection domain is implemented as a set of pointers to *capability lists*, which are arrays of capabilities. Contrary to classical software-based capability systems, Mungi's capability lists are not system objects but are user-maintained. Object accesses are validated by matching the list of valid capabilities (and corresponding access rights) recorded in the central *object table* against the capabilities found in the protection domain. If the validation succeeds an entry is made in a per-task *segment list*, which caches validations.

Capabilities in Mungi refer to "objects" which are contiguous ranges of virtual memory pages. No internal object structure is assumed by the system. The search of the protection domain implies that capabilities need not be explicitly presented to the system on the first (or any subsequent) access to an object. If, however, a capability is presented explicitly, it is immediately validated, and the segment list is updated as appropriate.

On a page fault, the segment list is first consulted, and if a matching entry is found, the corresponding page is mapped, otherwise the access is validated as above.

3. Protection Domain Extension

Validation of object access in the Mungi system requires the searching of two large data structures (object table and the capability lists). To amortize some of the validation costs much of the validation information is cached. Implementing protected procedure calls based on an extension of the caller's protection domain has two main benefits: firstly we can re-use the cached validation information from the caller's protection domain, and secondly the extension allows for the implicit sharing of large numbers of objects between the caller and the protected procedure.

Mungi's PDX mechanism allows the extension of a thread's protection domain for the duration of a procedure call. The kernel call

PdxCall(Cap pdx, Cap clist, uint method, uint nargs, ...)

invokes, with the given arguments, the entry-point designated by method of the object addressed by pdx. The kernel verifies that the method number falls into the range recorded in the object table. The invocation executes in a protection domain which is the union of the supplied clist and the protection domain registered (in the object table) for the PDX object (Fig. 1).

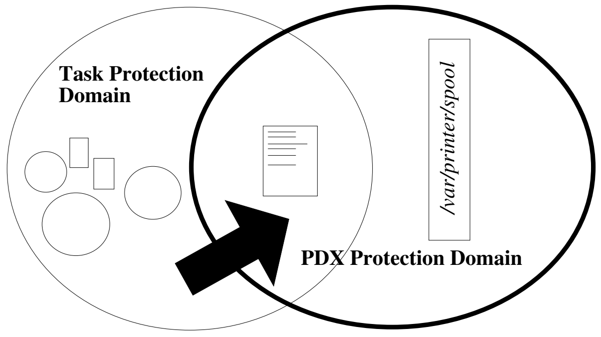

Figure 1. The PdxCall

If a null capability is passed to the clist parameter (as opposed to a capability to an empty clist, which indicates an empty protection domain) the caller's whole protection domain is merged with the protection domain registered for the PDX, effectively *extending* the caller's protection domain (hence the name PDX).

To return to the caller, the procedure executes a PdxReturn kernel call, which restores the caller's protection domain.

3.1. Implementation

Setting up a new protection domain is not a lightweight operation. However, the setup cost can be amortised by caching the PDX procedure's protection domain, in particular its access validations, between calls. When a task t calls a PDX procedure p for the first time, a new task t_p is created. The PDX call essentially becomes a blocking RPC call, which spawns a new thread in t_p for the duration of the PDX execution. The operation of creating a new thread and transferring control to it is very lightweight in Mungi, as it maps directly onto the corresponding operations of the underlying, very efficient, L4 microkernel [Lie95].

In the case of a proper protection domain extension (i.e. a null clist parameter is passed), t_p can inherit t's cached validations by having t_p reference t's segment list. If, during p's execution, new objects are validated, these validations are prepended to t_p's segment list, without affecting t's part of the segment list. If t validates further accesses between calls to p, these are inserted into t's part of the segment list. On the next call to p, t_p will then inherit these further validations as well.[1] This is shown in Fig 2. The kernel's data structure describing a task contains references to all PDX tasks belonging to it, so these can be cleaned up when the task exits.

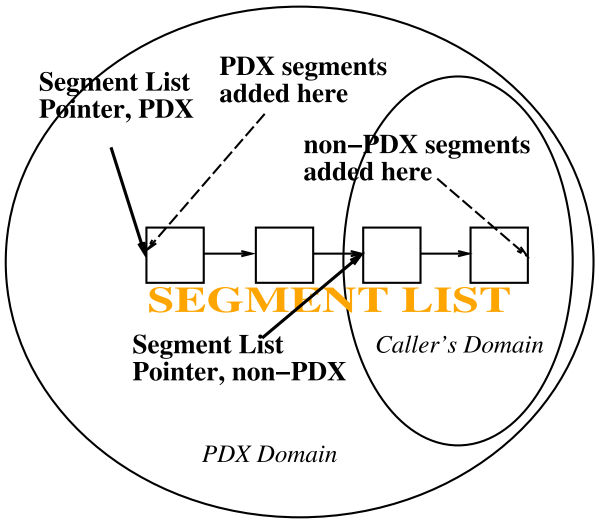
Segment List Pointer, PDX
PDX segments added here
non−PDX segments added here
SEGMENT LIST
Segment List Pointer, non−PDX
Caller's Domain
PDX Domain

Figure 2. Mungi segment lists

In the case of PDX calls with an non-null clist parameter, caching can also be used, provided the clist has not changed since the last call. The kernel can verify this by storing a hash of the clist with the cached protection domain.

3.2. Implicit PDX calls

PDX calls can also be made implicitly, i.e. without explicit presentation of the PDX capability. This happens when a task jumps to a PDX object, to which it holds (in its protection domain) no execute, but a PDX capability. If such a call is to a valid entry-point, it is equivalent to performing a PdxCall with the appropriate pdx and method parameters, and a null clist. However, such an implicit call is much more expensive than an explicit one, as a full validation of the access to the PDX object (i.e. matching the object table entry against the protection domain and verifying the entry-point) needs to be performed on each invocation.

[1]The same happens if, while one of t's threads executes p, another one of t's threads adds further validations — these become immediately visible to t_p.

4. Supporting Objects

The *NOM* object system on the IBM AS/400 [MM96] has demonstrated that it is possible to build an object oriented system on top of abstractions like those provided by Mungi (see Sec. 6). Here we show how Mungi can enforce encapsulation and support inheritance.

4.1. Encapsulation

Encapsulation can be enforced by the protection system if the provider of an object never hands out read, write, or execute capabilities to the object. Instead a PDX procedure is provided which, when invoked, extends the caller's protection domain by the appropriate capabilities to the object. Clients can thus only operate on the object by invoking this procedure. The PDX procedure code can actually be part of the object, or it can be separate.

4.2. Inheritance

To implement inheritance, jump tables used to access virtual methods are associated with the PDX objects. Potentially, these jumps are further PDX calls to methods of other classes. This can lead to a proliferation of cached PDX invocations.

A reduction of this overhead is possible if there is some trust between the classes (as there is likely to be if they are part of the same library). The derived class can then be given the capability to execute the superclass methods directly, i.e. by a normal procedure call.

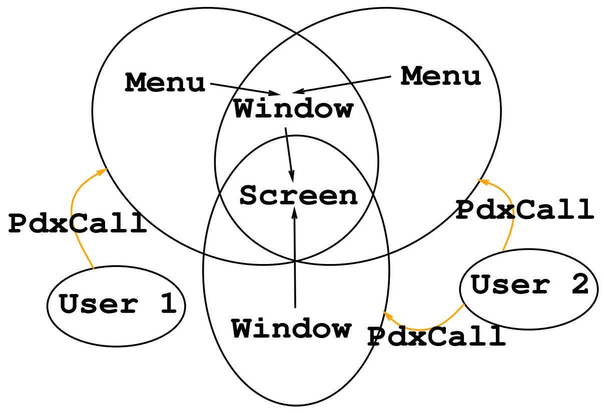

Figure 3. Inheritance

An example of this is given in Figure 3. There are three classes in this example: Menu, which is derived from Window which is derived from Screen. If user 1 invokes Menu and user 2 invokes Menu as well as Window, a total of eight PDX protection domains would need to be cached. However, if the various classes in the hierarchy trust each other, invocation of a superclass method by a subclass is by a normal procedure call, and only 3 PDX protection domains need to be cached.

5. Other Uses of PDX

5.1. Device drivers

Mungi takes the single address space concept seriously, by keeping out of the model anything which would introduce other forms of address space. For example, there is no disk model; clients, such as database systems, which require explicit control over I/O, can achieve this via virtual memory operations [ERHL96].

Similarly, other devices are mapped into VM. The device driver is given capability to the appropriate memory region. Users can then safely perform operations on the device by invoking the driver via a PDX call.

5.2. Services

Object oriented systems traditionally implement services using active servers, which interact with clients via IPC. In Mungi we use PDX objects instead. These are just passive entities which become active only when they are invoked via PdxCall. The "server" has access to the client's objects without having to pass them explicitly as IPC parameters. Clients still have the option of limiting the propagation of their protection domains, by specifying a reduced protection domain in the PDX call, if they do not trust the called PDX "library function".

5.3. User-level page fault handlers

User-level page fault handlers (ULPs) are essential for efficiently supporting databases and implementing persistence in Mungi [ERHL96]. A ULP is a PDX procedure which is invoked by the kernel when a page fault occurs on an object for which that ULP had been registered. ULP invocation uses an empty clist parameter, hence the ULP runs just within the protection domain which was registered for it. As the ULP has no access to either the kernel's or the faulting thread's protection domain, it does not need to be trusted. This is important, as every memory access can potentially lead to a page fault and a thread could otherwise not rely on keeping any secrets, as soon as it accesses another user's object.[2]

[2]Note, however, that the ULP can still interfere with the client's operation by denying service.

A single ULP can handle a large number of objects. Furthermore, as the ULP is invoked by the kernel and is passed an empty protection domain, all clients of a particular ULP can share the same cached PDX protection domain. This limits the number of ULP protection domains that need to be cached to one per ULP in actual use.

6. Relation to Other Work

PDX is conceptually very similar to the *profile adoption* mechanism used on the IBM System/38 [Ber80], and its successor, the AS/400. This mechanism allows invocation of a procedure with an amplified protection domain.[3] The caller can also restrict the part of its protection domain that is available to the callee (this is called "profile propagation" by IBM). The main difference to Mungi, in this context, is that System/38's implementation makes extensive use of specific hardware support, e.g. for tagging capabilities. Mungi's protection system is software based and can be implemented on standard hardware.

Opal [Cha95] is also a SASOS based on password capabilities. Opal uses a different form of protected procedure mechanism: Each protection domain can have one *portal*, which is an entry-point for cross-domain calls. When a call is made to the portal, control is transferred to a place specified by the domain. Any thread that knows a portal id can transfer to the portal's domain, so access control is left to the called domain. As portal invocations switch protection domains, rather than extending the caller's, the caller's and callee's protection domains may be disjoint. In order to make some of its objects accessible to the callee, the caller needs to pass capabilities for such objects explicitly to the portal. This is probably not a significant disadvantage in the context of Opal, as it is normal to present capabilities explicitly in that system. In Mungi, however, capabilities are normally presented implicitly (by storing them in a clist), which makes the protection system much less intrusive. Our PDX mechanism is consistent with this approach.

Angel [MSS+93] is another SASOS. Its approach to protection is to use upcalls to a protection server, which can implement any protection model.

The Grasshopper system [DdBF+94] is not a SASOS, but presents a generalised approach to address spaces, including the ability to emulate a SASOS. In Grasshopper, a protection domain is the union of the protection domains associated with the *locus* (execution abstraction) and the *container* (storage abstraction). When a locus enters a different container, its protection domain automatically changes accordingly.

[3]IBM's term "user profile" essentially refers to a protection domain.

7. Conclusion

In this paper we have presented the mechanism of protection domain extension and described its implementation in the Mungi single address space operating system. The mechanism is relatively expensive on a first call, comparable to the creation of a process, but extensive caching of validation information across calls allows this cost to be amortised over many invocations. The cost of a call other than the first one is essentially that of two cross-domain IPC operations and a thread creation. These are extremely fast in the underlying L4 microkernel, we therefore expect good performance of PDX in Mungi.

The PDX mechanism presents the basis of object support, particularly encapsulation, in Mungi and is also used to access devices and for implementing user-level page fault handlers.

References

[Ber80] Viktors Berstis. Security and protection in the IBM System/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–250. ACM/IEEE, May 1980.

[Cha95] Jeffrey S. Chase. *An Operating System Structure for Wide-Address Architectures*. PhD thesis, University of Washington, 1995.

[DdBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.

[ERHL96] Kevin Elphinstone, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Supporting persistent object systems in a single address space. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, NJ, USA, May 1996. To be published.

[HERV94] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. In *Proceedings of the 17th Australasian Computer Science Conference*, pages 271–80, Christchurch, New Zealand, January 1994.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[MM96] Ashok Malhotra and Steven J. Munroe. Schema evolution in persistent object systems. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, NJ, USA, May 1996. To be published.

[MSS+93] Kevin Murray, Ashley Saulsbury, Tom Stiemerling, Tim Wilkinson, Paul Kelly, and Peter Osmon. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proceedings of the 2nd USENIX Symposium on Microkernels and other Kernel Architectures*, pages 31–43, September 1993.

[VRH93] Jerry Vochteloo, Stephen Russell, and Gernot Heiser. Capability-based protection in the Mungi operating system. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, pages 108–15, Asheville, NC, USA, December 1993. IEEE.