SINGLE ADDRESS SPACE OR PRIVATE ADDRESS SPACES ?

Jacques Mossière, Xavier Rousset de Pina
Bull-IMAG/systèmes, 2 avenue de Vignate, F 38610 Gières
{Jacques.Mossiere, Xavier.Rousset}@imag.fr

1. INTRODUCTION

The appearance of 64 bit address space architectures (DEC Alpha, HP P/A RISC and MIPS R4000), has fostered interest in single address space operating systems. Examples of such implementations are Opal[Chase 1994], Angel[Murray 1993] and Mungi[Heiser 1993]. They all offer the following advantages:

- a virtual address is the universal means to reference any object in the system; therefore, there is no need for pointer swizzling,
- when an object is deleted, and because of the size of the address space, there is no need for address reutilization, with the expensive problems it implies (address ambiguities, dangling pointers),
- and, last but not least, any application program or data in the system may be statically linked.

The main challenge is probably the design and implementation of a protection scheme to limit the access rights of each thread or application running in the single space. Radical proponents of “pure” single address space claim that protection is orthogonal to addressing, which is true. However, due to the limitations of hardware mechanisms to enforce protection rules, and to the distributed environment, the single space is in fact implemented by the co-operation of multiple virtual address spaces. Each protection domain is implemented by a virtual space. Another difficulty is the management of private data which implies some kind of base register addressing.

Our claim in this position paper is that the management of data private to domains or threads is an intrinsic function of any distributed virtual memory system. Rather than hiding them in a single space, it is more efficient and simpler to implement these private areas at fixed locations in each virtual address space which will be dynamically bound to distinct physical locations. Hence, each address space is statically divided in two areas: a common area which is managed as a single address space, and a private area. Such a system combines the advantages of

Opal-like systems concerning static addressing and linking, and those of classical virtual memory systems concerning the selection of local data.

The remainder of this paper is organized as follows. In section 2 are discussed two different schemes for protection, both relying upon the capability concept. Section 3 gives an argument for the use of private data, and shows that they can be implemented at low cost. Section 4 outlines a proposal in which a shared single address space for persistent data coexists with private segments for temporary data. In conclusion, this proposal is compared to previous systems.

2. A COARSE GRAIN PROTECTION MODEL USING CAPABILITIES

2.1. Motivation

In a single address space operating system, we need to determine which parts of the space are accessible by a given thread and to dynamically enforce isolation between executions.

For the first point, we have to introduce some kind of protection model allowing the description of the address space in terms of protection units and access rights. The protection unit is often a logical segment, and rights are expressed either by access control lists or by capabilities. The two models we briefly explore below share the concepts of segment and capability. In the OPAL solution, the concept of portal is added to control transfers from one translation domain to another. In the second one, the concept of protection domain, a set of segments which are to be accessed in the same translation domain, is introduced.

Concerning the second point, isolation is enforced by running different threads in different translation domains[1]. When several threads are running in the same domain they share the segment attached in the domain with the same access rights.

2.2. The model "à la OPAL"

In OPAL, a translation domain may contain several threads. If a thread owns a capability for a segment, it may map it in its current translation domain with the rights defined in the capability. It is also possible to define a new translation domain for the thread, to map the new segment in this domain, and to transfer control to a valid entry point of the segment by the use of the *Portal* concept.

The definition of higher level units for protection is left to the application or preferably to the run time of the compilers.

[1] A translation domain has in this paper the same meaning as a domain in the classical SASOS litterature. We reserve protection domain to denote a structured unit for protection, as described in 2.3

2.3. A structured model

Conversely, systems like Hydra [Wulf 81] define at a basic level a compound protection unit called a protection domain. A protection domain is defined by a list of capabilities, or C-list, containing the capabilities for the segments to be executed in the domain. In this case, a right to call a new domain may be inserted in a capability pointing to a C-list. A translation domain is associated to each protection domain. When a thread addresses a new domain by a capability with the call right, there is an implicit transfer of control from the old domain to the new one. A capability with a call right access has to define the new protection domain as well as a valid entry point for it in order to ensure a valid execution in the new domain.

2.4. Comparison

Clearly, the second solution is an extension of the first one. If implemented at the most basic level, protection domains and transfers of control between them will certainly be more efficient than if they are implemented in a user level application or run time system. The question, which remains open in our opinion, is to determine whether it is realistic to define a protection model providing the requirements of any application.

3. THE NEED FOR PRIVATE DATA

Consider the implementation of an object-oriented runtime support on top of a single address space. For this application we need a protection model based on fine grain access control to objects such as [Coulouris 1994] or [Hagimont 1994]. In these models objects are small and grouped in segments. The access rights on an object in a particular protection domain are expressed in terms of an object view. An *object view* is the subset of the methods of the object's class that can be invoked by the threads running in a particular protection domain. An object may be shared by different protection domains with different views. The implementation of views is generally achieved by means of specially tailored dispatch vector whose entries point only to the methods permitted by the rights of the threads running in the domain (Figure 1). If an invocation refers to a method that is not permitted in the domain, a run time exception is raised. Dispatch vectors associated with objects are data private to a domain.

So shared code segments may not only be concerned with private static data (i.e. global variables declared in a module or dispatch vector use for late binding) [Chase 1992] but also with data segments.

If the system provides data local to a domain, the private static data can be addressed using their absolute virtual addresses. In particular, the dispatch vector is at the same addresses in all the protection domains where the object can be accessed. The dispatch vector address and the object address are both computed at the object creation and will remain the same during the lifetime of the object. A pointer to the dispatch table is simply located in the object state.

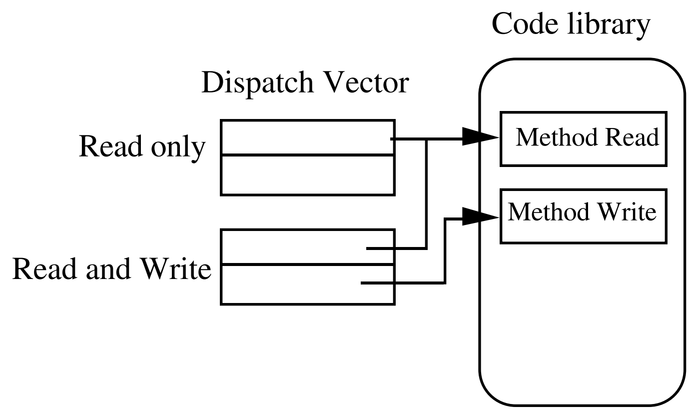

Figure 1: Dispatch Vector

If the system does not provide any local data, a static data reference must be expressed as an offset from a base register. In particular, one way to implement the dispatch vectors associated with objects in a domain is to use linkage segments à la Multics. A segment *SL(D)*, called linkage segment, is associated to a segment *S* in each domain *D* in which it can be attached. The dispatch vector associated with an object contained in *S* is allocated in *SL(D)* when the object is created. The dispatch vector must be at the same offset in all the linkage segments *SL(D)*. This offset is stored in the object state. The management of the consistency of the different linkage segments associated with a segment in the different domains in which it can be attached may be too much expensive. Furthermore, base pointers used to address data in the linkage segments must be saved and restored whenever control crosses segment boundaries.

4. A PROPOSAL FOR SIRAC

Project SIRAC (Système Informatique Réparti pour Applications Coopératives)[2] aims at providing a distributed environment for co-operative work. The applications are written in a high level object oriented language such as C++ or Guide. The objects are persistent and distributed. The system is partitioned into cells connected by a metropolitan or wide area network, each cell is composed of a set of homogeneous stations connected by a high speed (generally local) network. We require that the machines in a cell have the same 64 bit address architecture.

The lowest layer of SIRAC implements a distributed virtual memory including a shared area managed as a single address space common to all threads and protection domains in the cell, and a set of private areas, one for each domain. All these private areas are accessed in the same address range, the private area for a domain being selected by the dynamic address translation

[2]Sirac is also the name of a summit in the higher Alps of Dauphiné

facility. A lock manager and a checkpointing facility are also provided at this level. These facilities are intended to support a database management system.

On top of this memory is implemented a coarse grain segment model with a protection scheme based on software capabilities (cf. section 2). For efficiency purposes, the access rights for a segment in a domain are only checked at the first reference to the segment in that domain; afterwards, a processor accesses the segment by its virtual address.

5. CONCLUSION

In the late sixties, some of the ideas developed here were explored in the context of small address spaces (see for example [Bétourné 1970] or [Watson 1970]). In Esope, a virtual space was associated with a multi-thread process and divided in two parts for private and public data. The size of the space dictated that the shared part contained one program at a time, which resolved the linking problems. This approach became obsolete with the appearance of segmented machines (Multics) and later of capability machines (Hydra). It gains a renewed interest with wide address space architectures, in which it is realistic to manage the shared part as a single address space with all of its advantages. Within the Sirac project, we intend to provide this kind of distributed virtual space at the lowest level and to implement on top of it a fine grain protection scheme as outlined in section 2.

6. ACKNOWLEDGEMENTS

We would like to thank George Coulouris, Jean Dollimore and Sacha Krakowiak for their helpful comments and careful editing of the paper.

7. REFERENCES

Bétourné, C., Ferrié, J., Kaiser, C., Krakowiak, S. and Mossière, J. (1970). "Process Management and Resource Sharing in the Multiaccess System ESOPE." Communications of the ACM **13**(12)

Chase, J. S., Levy, H. M., Baker-Harvey, M. and Lazowska, E. D. (1992). How to Use a 64-Bit Virtual Address Space, Technical Report 92-03-02, University of Washington, Seattle 98195, March 1992.

Chase, J. S., Levy, H. M., Feeley, M. J. and Lazowska, E. D. (1994). Sharing and Protection in a Single Address Space Operating System. University of Washington.

Coulouris, G. and Dollimore, J. (1994). Security requirements for cooperative work: a model and its system implications. Department of Computer Science, Queen Mary and Westfield College, University of London (position paper submitted to Sixth European SIGOPS Workshop)

Hagimont, D. (1994). Protection in the Guide object-oriented distributed system. ECOOP94, Bologna, Italia

Heiser, G., Elphinstone, K., Russell, S. and Hellestrand, G. R. (1993). A Distributed Single Address-Space Operating System Supporting Persistence. School of Computer Science and Engineering, The University of New South Wales.

Murray, K., Wilkinson, T., Osmon, P., Saulsbury, A., Stiemerling, T. and Kelly, P. (1993). Design and Implementation of an Object-Oriented 64-bit Single Address Space Microkernel. Microkernels and Other Kernel Architectures, San Diego, California, Usenix association.

Watson, R. W. (1970). Timesharing system design principles. Mc Graw-Hill.

Wulf, W. A., Levin, R. and Harbison S. P. (1981), Hydra/C.mmp An experimental Computer System, Mc Graw-Hill