The Grand Unified Theory of Address Spaces

*Anders Lindström, *John Rosenberg and †Alan Dearle

*Department of Computer Science
University of Sydney, Australia
{anders, johnr}@cs.su.oz.au

†Department of Computing Science
University of Stirling, Scotland
al@cs.stir.ac.uk

Abstract

A key decision in the design of an operating system is which facilities to provide for the management and composition of the address space. A wide spectrum of schemes exist, ranging from the private process address spaces of Unix through to the recently revived single address space approach. This paper proposes a new model which provides a unified and generalised approach to address space management. The model presents a single abstraction of address spaces which are orthogonal to processes and may be composed in arbitrary ways. The power of the model is demonstrated by case studies which show how shared libraries, a Unix system and a single address space system may be implemented.

1. Introduction

The structure and facilities provided for the manipulation of the address space constitute a fundamental element in the design of an operating system. The approach taken effectively dictates the computational model for programs executing on the system and can limit the scope of the sharing and protection paradigms that are possible. A wide spectrum of approaches to the provision of the address space have been proposed and implemented, each with inherent advantages and limitations.

Probably the best known approach, representing one end of the spectrum, is the model originally supported by Unix. In this model each process has its own address space, containing (possibly shared) code and private data. A major advantage of this approach is that it provides automatic, hardware supported protection between processes. However, the cost has been severe restrictions on the modes of sharing that can be supported. The use of separate process address spaces, combined with multitasking, has lead to a sharing paradigm based on a number of small applications executing in separate processes and communicating via pipes. This approach has proven extremely effective where the information being shared maps easily to a byte stream but is inadequate for more complex data types such as trees or graphs. It has been argued that this design was very much the result of the limited size of address spaces supported by machines. Certainly this was the case for Unix which was designed to run on machines with a 64Kb address space.

The recent advent of machines with 64 bit addresses has caused system designers to rethink the segregated model and to revive an idea first proposed by the MULTICS designers [13] and implemented on dedicated hardware by systems such as Monads [14] . Such systems may be viewed as being at the opposite end of the spectrum from the traditional Unix model and are called *single address space operating systems (SASOS)* [3] . In a SASOS, all processes operate in the same address space. Therefore all data and code must reside within this address space in some unique location. The arguments for this approach include improved sharing, flexible protection models, and support for transparent persistence and distribution.

While the basic idea of a SASOS is appealing, there are a number of problems that have not yet been adequately addressed by their designers. These problems relate to address space reuse and the allocation of addresses in a distributed system. In addition, most SASOS designs rely on the availability of, at least, a 64 bit address space. While 64 bit *processors* exist, none of them actually provides a true 64-bit *address space* and it is not clear when they will.

Between the two ends of the spectrum, there are a number of schemes that are based on a segregated model but that provide improved sharing facilities. In modern Unix environments these include shared, dynamically linked libraries and memory mapped files. In micro-kernel systems, such as Mach and Chorus, these facilities are generalised to a model based on memory objects. While these models provide more flexibility than traditional operating systems, actual experience shows that inadequacies remain for some applications [15] .

We believe that each of these approaches involves trade-offs that are best evaluated on an application-by-application basis. Therefore, the operating system should not dictate which one is used. In this paper we propose a model of address space management that is general enough to implement any of these schemes described above with no significant loss of efficiency. It also provides for some novel approaches unavailable in any of them. The paper first describes the model and then demonstrates the power of the model through several case studies.

2. A General Model of Address Spaces

In this section we describe a new multiple address space model which provides a set of abstractions that allow arbitrary models of sharing. The abstraction over data access and storage is the *container*. The abstraction over execution is the *locus* (from *locus of execution*); a term used to avoid any preconceptions attached to the terms *process* and *thread*. Protection is achieved using a capability system.

2.1 Containers

Containers serve a number of purposes in our model. First, they provide a means for data storage. The data in a container is persistent, meaning that its lifetime is orthogonal to the lifetime of the creating locus and that it survives system shutdowns. The details of container data management are further discussed in [7] . Second, containers provide address spaces for loci which execute within them. A single locus may move between containers by the invocation mechanism explained below. The container in which a locus currently executes is called its *host container*. During execution, all virtual addresses issued by a locus correspond directly to the container addresses of the host container. Containers differ from virtual address spaces on conventional systems because they may be bigger than the address space provided by the hardware and they are persistent. The third purpose of containers is to provide a mechanism for address space composition through the mapping facility described below. Finally, containers serve as a generic protection mechanism both by providing hardware protected abstract data types through the invocation mechanism and by providing 'views' of data using the mapping mechanism.

2.2 Loci and invocation

Our computational model is procedure-oriented\[11] in that communication is achieved by loci moving between containers by *invoking* them. 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. This facility 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. As a consequence of this, a particular container may have any number (including zero) of loci executing within it. Parameters consisting of data and capabilities for other containers (see below) may be passed on an invoke operation. This mechanism is very similar to those provided in other object-based systems such as Monads [14] , Clouds [6] , Alpha [5] and Spring [9] .

One of the key features to the invocation mechanism is that it is potentially very fast in comparison to message-passing systems[2, 10] and for this reason a variant of this model has been incorporated into Mach[8] .

2.3 Mapping and address space composition

The purpose of *container mapping* is to allow data to be efficiently shared between containers. This is achieved by allowing data in a region of one container to be made accessible (either as read-only or read-write) at a range of addresses in another container. Importantly, any mappings made into the source container of the mapping are also visible in the destination container. We call this *recursive* or *transitive* mapping. This provides a powerful mechanism for *composing* address spaces and follows naturally from the model because there is only one abstraction of data storage and access. Recursive mapping is a departure from other systems, such as Mach, which provide separate abstractions for data storage (memory objects) and address spaces, thereby enforcing a single level of composition.

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, the model provides the notion of *locus private mappings*, which are always created relative to a locus and take precedence over container mappings. Only the locus for which they are created perceives them. This allows, for example, each locus to have its own stack occupying the same address range as the stacks of other loci. This technique is used in some native Unix kernel implementations to simplify stack management.

An interesting feature of locus private maps is that they remain in effect even while the locus is not executing in the destination container of the mapping. Thus, if that container is visible through a mapping or the locus returns to it, the locus private mapping will still be taken into account during address translation.

Finally, in addition to the data in a container being persistent, any mappings made into it are also persistent, i.e. they may outlast the creating locus.

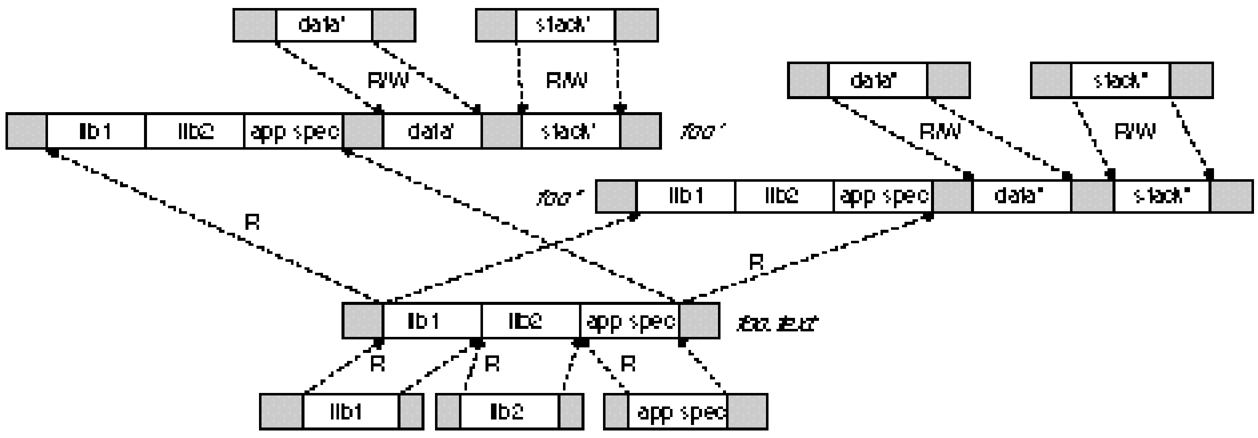

Figure 1. Statically linked shared libraries using recursive mapping.

2.4 Access control

Access control is enforced via a capability-system, which provides a location-independent means of referencing entities within the system. A capability must be presented to invoke a container or to execute any mapping operation. This, combined with the fact that a locus can only address data within its host container and those mapped into it, provides a flexible protection environment.

3. Case Studies Using the New Model

3.1 Statically-linked shared libraries

Many conventional operating systems use shared libraries to reduce the disk and physical memory requirements of programs. Additionally shared libraries may be updated without requiring programs to be relinked. One of the drawbacks of current implementations is that all processes using shared libraries, including instances of the same program, must incur the cost of dynamic linking.

In our model, it is possible to implement shared libraries that are *statically* linked. This means that the text segment required by a program can be created *before* any instances of that program come into existence but that many different programs may still share one copy of the library. This approach can therefore deliver all the benefits of shared libraries without incurring the cost of dynamic linking.

Consider an application call *foo* that uses two libraries *lib1* and *lib2*. Figure 1 illustrates how *foo*'s text segment can be created and shared between different processes. In this example, the data representing the code for the libraries and the application-specific code of *foo*, reside in separate containers and the text segment of *foo* has been created by mapping the libraries and application code into another container called *foo.text*. Notice that, because container creation is independent of loci, this operation can be performed by the linking system before any instances of *foo* are created.

In the example, there are two instances of *foo* called *foo'* and *foo"*. Each instance is implemented as a container that maps in three containers, a private data segment, a private stack segment and the shared text segment. Since the text segment has already been composed, there is no need to perform dynamic linking. The efficacy of avoiding dynamic linking has been shown in the Spring system[12] . The Spring model differs somewhat from ours because it does not allow recursive mapping. Therefore, even though the libraries may have already have had their external references resolved, each library must be separately mapped into each instance of the program. In our model, an intermediate container, such as *foo.text* in figure 1, can be used to reduce this operation to a single mapping.

The above example serves as an illustration of address space *composition* using recursive mapping. Clearly, there are other issues such as how changes to libraries are effected and how the actual linking is performed but these issues are relevant to *any* system that uses shared libraries. Further, the example shows how libraries can be shared between difference instances of the same program. In cases where the libraries are comprised entirely of position independent code, a single copy of the library may be shared between all programs that use it. Such considerations are beyond the scope of this paper.

3.2 Unix processes

One of the implicit benchmarks when designing and implementing new operating systems is whether or not it is possible to implement a Unix emulation. The purported benefits of this are that one can then port familiar programs without having to completely rewrite them and that it somehow validates the new operating system. In this section we describe how locus-private maps can be used in this context.

One of properties of Unix is that a process in kernel-mode can directly access it own user-level address space. While alternative implementations of

Unix on micro-kernels such as Mach[1] and Chorus[4] are possible, the native Unix approach is tried and proven and is clearly efficient.

One of the problems with implementing Unix as a user-level program in, for example, Mach is that the ‘kernel’ resides in a different address space from the processes. This makes it difficult to directly access the process’ address spaces. While the implementors of the Mach Unix server[9] show that techniques such as memory mapped files and running ‘system code’ at user-level can ameliorate this problem, it seems clear that giving the Unix server direct access to processes’ address spaces would have been beneficial.

The solution to this problem in our model hinges on locus private mappings. Consider figure 2a and 2b. Here, the Unix server resides in a separate container, *US*, that is divided into two large regions called the *process region* and the *server region*, corresponding to user and kernel regions in a native implementation. There are two Unix processes, *P1* and *P2*, that are each implemented as separate containers with text, data and stack segments in the process region of the container. The active part of a process is simply a locus (or many loci if threads are supported) running in the container. When the server creates each process, it also performs a read/write locus private map between the process regions of both the process and the server containers. This map is made relative to the process’ locus. In the example, these locus private maps are denoted as *lpm1* and *lpm2* and are made relative to locus *l1* and *l2* respectively.

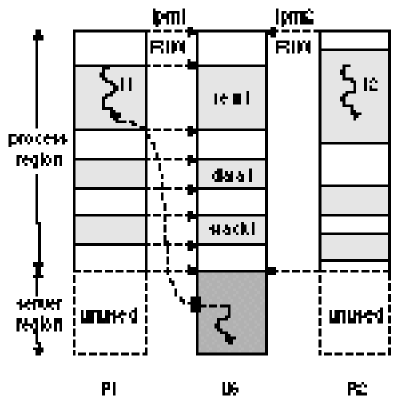

Figure 2a. P1 in kernel-mode.

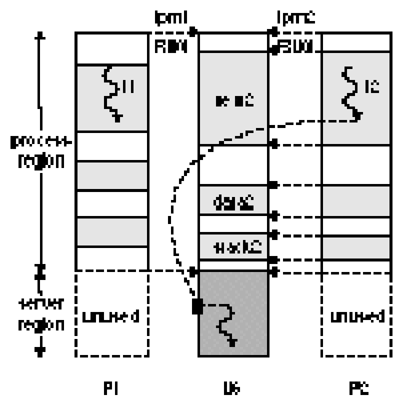

Figure 2b. P2 in kernel-mode.

A system call is implemented as an invocation to the server. Since it is a process’ locus that is actually in the kernel during a system call, any accesses to the process region in the server are translated into accesses to the process’ address space through the locus private map. In figure 2a, *P1*’s locus, *l1*, has performed a system call by invoking *US*. It can be seen from the diagram that accesses to *US*’s process region by *l1* are translated into accesses to *P1*’s container through *lpm1*. Figure 2b shows *l2*’s view of the process region while in *US*.

Since the mappings into the server’s process region are private to a process’ locus, each process can only access its own address space while in the kernel. This is true even if many processes are in kernel-mode simultaneously. Note that, due to the orthogonality of mappings and loci, this mapping only need be performed when the process is created.

3.3 A single address space environment

Having shown how the traditional segregated model of address spaces can be efficiently implemented in our model we move to the other end of the spectrum, namely single address space operating systems (SASOSs). Since our system does not *enforce* a single address space for the entire system the following example is not a single address space *operating system*. Rather, it is a single address space *environment*. While we believe that the former is not really a viable proposition for reasons outlined in section 1, we do believe that the main benefit of a single address space, i.e. meaningful pointers between protection domains and machines, may usefully be exploited in applications that exist in the larger multiple address space environment.

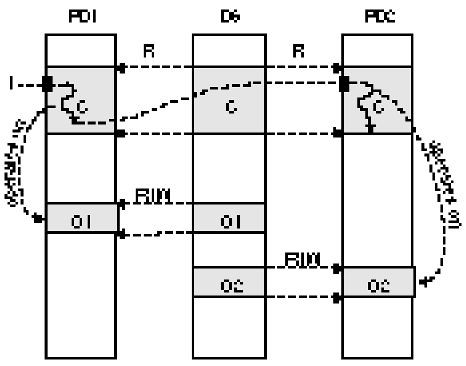

Figure 3. Two protection domains in a single address space environment.

One of the best know SASOS designs is the Opal system[3] . There are two key aspects of Opal in terms of address space management. First, there is a single address space in which all data resides. Second, processes execute in *protection domains* that are a collection of protected (read/write or read) windows onto the data space. Processes may move between protection domains thereby changing the portions of the data space which they can access. Access to protection domains is governed by a capability system. Figure 3 shows how an Opal-like environment may be implemented in our model.

The data space itself is implemented as a single container, *DS*, in which all the data resides. There are two protection domains *PD1* and *PD2* which share a common code segment *C* but operate on two different objects *O1* and *O2*. Each domain is implemented as a separate container with mappings from the appropriate parts of the data space. Protection is based on the fact that a locus, while in a protection domain, can only access those parts of the data space that are mapped into that domain.

The entry point of a domain is set to the appropriate address in the code segment. Thus, when a locus wants to move to a new domain it simply invokes the container representing the domain and starts executing at the correct location. In figure 3, for example, the locus *l* moves from PD1 to PD2. Access to protection domains is controlled by the capability system and is therefore secure.

4. Conclusions

The provision of flexible facilities for the management and composition of the address space has long been an important topic in operating system design. It has taken on more significance with increase in demands being made of the operating system by developments such as shared libraries, distributed shared memory, persistence, etc. A wide spectrum of address space schemes has been developed in an attempt to satisfy these needs.

In this paper we have presented a new address space model which unifies all of these ideas. Our model allows both arbitrary composition of address spaces and arbitrary relationships between address spaces and execution entities, as well as flexible control over access to the address space. We have demonstrated the generality of the scheme by showing how shared libraries, the Unix model and a single address space environment may be implemented above the new model. Although space has not permitted us to discuss persistence, the model also includes mechanisms to support user-level control over the lifetime of data [7] .

The model described has been implemented as part of the Grasshopper operating system project at the Universities of Sydney and Adelaide. The first implementation is on the DEC Alpha platform and is operational in a limited form. A prototype of the Unix server also exists. Initial timing experiments indicate that the overheads associated with maintaining the model are, on average, no higher than those in other operating systems. It is expected that a fully functional version of the kernel will be completed by early 1995. We intend to implement a number of other more novel servers in order to experiment further with the model.

Acknowledgements

This work was supported by Australian Research Council grant A49130439 and by an equipment grant from Digital Equipment Corporation. The authors also wish to express their appreciation to Rex di Bona, Matty Farrow, Frans Henskens, David Hulse, Stephen Norris and Francis Vaughan who have made valuable contributions to this work.

References

[1] Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation for Unix Development", *Proceedings, Summer Usenix Conference*, pp. 93-112, 1986.

[2] Bershad, B. N., Anderson, T. E., Lozowska, E. D. and Levy, H. M. "Lightweight Remote Procedure Call", *Transactions on Computer Systems*, vol 8, 1, pp. 37-55, 1990.

[3] Chase, J. S., Levy, H. M., Feeley, M. J. and Lazowska, E. D. "Sharing and Protection in a Single Address Space Operating System", *ACM Transactions on Computer Systems*, May, 1994.

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

[5] Clark, R. K., Jensen, E. D. and Reynolds, F. D. "An Architectural Overview of the Alpha Real-Time Distributed Kernel", *Usenix Workshop on Microkernels and Other Kernel Architectures*, 1992.

[6] Dasgupta, P., et al "The Design and Implementation of the Clouds Distributed Operating System", *Computing Systems Journal*, vol 3, pp. 11-46, 1990.

[7] 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*, vol 7, 3, pp. 289-312, 1994.

[8] Ford, B. and Lepreau, J. "Evolving Mach 3.0 to a Migrating Thread Model", University of Utah, UUCS-93-022, 1993.

[9] Golub, D., Dean, R., Forin, A. and Rashid, R. "Unix as an Application Program", included with Mach distribution..

[10] Hamilton, G. and Kougiouris, P. "The Spring Nucleus: A Microkernel for Objects", Sun Microsystems Laboratories, TR-93-14, 1993.

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

[12] Nelson, M. N. and Hamilton, G. "High Performance Dynamic Linking Through Caching", Sun Microsystems Laboratories, TR-93-15, 1993.

[13] Organick, E. I. "The Multics System: An Examination of its Structure", MIT Press, Cambridge, Mass., 1972.

[14] 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.

[15] Vauhgan, F., Basso, T., Dearle, A., Marlin, C. and Barter, C. "Casper: A Cached Architecture Supporting Persistence", *Computer Systems*, vol 5, pp. 337-359, 1992.