Wide-Address Spaces — Exploring the Design Space

Alberto Bartoli[1] **Sape J. Mullender**[2] **Martijn van der Valk**[2]

November 16, 1992

Abstract

In a recent issue of Operating System Review, Hayter and McAuley [1991] argue that future high-performance systems trade a traditional, bus-based organization for one where all components are linked together by network switches (the Desk-Area Network). In this issue of Operating System Review, Leslie, McAuley and Mullender conclude that DAN-based architectures allow the exploitation of shared memory on a wider scale than just a single (multi)processor. In this paper, we will explore how emerging 64-bit processors can be used to implement shared address spaces spanning multiple machines.

1 Introduction

In a recent issue of Operating System Review, Hayter and McAuley [1991] argue that future high-performance systems trade a traditional, bus-based organization for one where all components are linked together by network switches (the Desk-Area Network). In this issue of Operating System Review, Leslie, McAuley and Mullender conclude that DAN-based architectures allow the exploitation of shared memory on a wider scale than just a single (multi)processor. In this paper, we will explore how emerging 64-bit processors can be used to implement shared address spaces spanning multiple machines.

The major problem of information sharing is this: For processes to share some information, they have to agree upon *what* to share, i.e. they have to agree upon the *name* of that information. From the performance point of view, the most efficient naming scheme is, of course, the use of *memory addresses*: A pointer in memory can be dereferenced in only one machine instruction; parsing of any other kind of name — especially a string-based one — is far more expensive. In virtually all of today's systems naming schemes based on memory addresses can be only used by threads in one address space, or processes sharing a memory segment and running on one machine. Apart from any other considerations, the inability to share addresses more widely is motivated by the fact that with current 32-bit architectures the size of the address space is not large enough to encompass multiple processes *and* multiple machines. Therefore, pointers crossing machine boundaries must always be translated to some other sort of name.

An approach that seems promising is the use of the emerging *64-bit address spaces* (e.g., the DEC Alpha architecture, or the MIPS R-4000 architecure). The huge size of the virtual address space of these new architectures now allows designing systems in which an address space can be *shared* across a set of machines having the same architecture. In such a system it would be

[1]. University of Pisa, Dept. of Information Engineering, Faculty of Engineering, Via Diotisalvi 2, 56126 Pisa, Italy
[2]. University of Twente, Faculty of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands

possible to have processes able to name objects by means of their addresses, even if they were running on different machines. Wide-address-space architectures may thus strongly reduce the performance problems related to marshalling and naming, which potentially make them a platform for supporting information sharing much more efficiently. In the rest of this document, we will discuss the issues raised by these new architectures, and we will try to highlight the corresponding problems.

2 Shared Wide-Address Spaces

2.1 Scale of the Address Space

Sharing an address space among processes running on the same machine provides by itself some potential for better performance (Bershad et al. [1990]; Koldinger et al. [1991]). According to the previous section, however, far bigger advantages should come by sharing an address space across *multiple* machines. The question that arises is deciding the size and kind of the group of machines sharing the address space. Obviously, a major requirement is that they must be of the same architecture, otherwise there would be no point in exchanging pointers among processes running on different machines, and the contents of each piece of shared memory would have to be repeatedly marshalled and unmarshalled.

Assuming, thus, that an address space is shared among a group of like machines, the problem turns into defining how big that group can be, and how much its components can be physically dispersed. The number of machines that can share an address space is basically limited by the fact that even in a 64-bit space, virtual addresses do not completely come for free: If ten machines create objects at a rate of a ten gigabytes a minute, the address space will last 300 years, but if ten-thousand or a million machines do it, the address space is used up rather quickly. Therefore, it seems reasonable to assume that an address space will be shared by a few tens of machines at most. About the physical distribution of those machines, problems of trust arise: Addresses are a shared resource in a shared address space. Processes must be able to count on certain objects to be reachable at certain addresses. If a number of machines manage an address space together efficiently, they must trust each other to respect the allocation policies for virtual addresses.

To sum up, it seems reasonable to assume that an address space will be shared across a *small, local* group of *like* machines. The physical memory of each machine will act as a *cache* for this shared address space, under control of the local kernel.

2.2 Persistence of the Address Space

Another issue, somewhat independent from the facilitation of information sharing, that wide-address-space architectures make worth exploring, is that of the *lifetime* of an address space. In today's systems, an address space is associated with a *process*. The lifetime of an address space is the same as the lifetime of the corresponding process: They are created and cease to exist together. In today's systems virtual addresses have to be heavily reused, i.e., their number is not enough for an address space to outlive the lifetime of even a single process. With 64-bit architectures that fundamental constraint is removed, and it becomes possible to start designing systems in which

an address spaces is *long lived* in that it may even outlive the uptime of individual machines. The basic idea here is that of backing up on secondary storage information about the layout of the address space — which segment is mapped at which address — together with the actual contents of "segments" in the form of "files".

The interesting issue is to define precisely what "long-lived" means, and from this point of view several design choices are possible. One possibility is that every object is given a range of addresses at creation time that lasts for its whole lifetime. Another possibility is that of focusing on the validity of *pointers*. For instance, the system could guarantee that once a pointer has been assigned the memory address of an object, this pointer will remain valid until the pointed-to object is deleted. Alternatively, objects can be deleted automatically when no pointers to them exist anymore. This will require a garbage-collection mechanism. Note that when a segment is deleted that is pointed-to from other segments, the system must guarantee that when attempting to dereference those pointers, these will be recognized as being invalid. This complicates the *reuse* of virtual addresses — if the system needs to.

If address spaces are long lived and objects keep their address for their lifetime, objects containing internal pointers and pointers to other objects can be stored directly in secondary storage — when they are retrieved they are retrieved to the same virtual address (so internal pointers will keep their meaning) and pointers to other objects will still be valid.

If, contrariwise, objects are stored on secondary storage in marshalled form (which would be useful in any case when objects are shared across address-space boundaries), the need for permanence of object addresses can be somewhat relaxed. Applications must "attach" an object explicitly before it uses pointers to or into it, and "detach" it when it no longer uses those pointers.

2.3 Access to the Address Space

The fact that all processes see the same address space obviously does not imply that they all have equal access to it: Segments have to be associated with a set of protection attributes — read, write, execute — separately mantained per process or group of processes. Differently stated, in such a system protection of memory must be managed in a way independent from the virtual-to-physical mapping, unlike what happens in contemporary systems. Reflecting this separation of concerns at the hardware level may also give substantial performance improvement (Koldinger et al. [1991]).[3]

When sharing an address space across multiple processes, virtual addresses form a *shared* resource. If the address space encompasses also multiple machines, the shared nature of virtual addresses raises some problems that usually are not present in traditional systems. For instance, allocation and deallocation of addresses must be performed in a "consistent" way; that is, kernels of different machines must never create different segments whose address ranges partly overlap, or map the same segment to different addresses for different processes. Satisfying this requirement is not difficult, however, and several efficient solutions may be devised.

What is really important about the shared nature of virtual addresses is that, whenever the kernel has to set up the virtual-to-physical mapping for a segment, it has to fetch the necessary information from a *trusted* part of the system: A malicious entity could reply to a request for the

[3]. Note, however, that the separation above can be implemented also on architectures conceived to support the traditional notion of process as an entity associated with its own private address space. From this point of view, what is needed is basically structuring the kernel in such a way that page tables of all the processes running on a given machine are mantained identical with respect to the virtual-to-physical mapping.

addresses associated with some segment S, by specifying a range that overlaps with addresses already allocated to some other segment T; this would affect not only the processes using S, but also those using T. This problem of trust is perhaps the major one when attempting to associate the address space with some notion of permanence (Section 3). Its most obvious implication is that a user-level file server providing *contents* of a segment cannot decide on the *address* of that segment.

2.4 Addressing Objects

Every address space needs to use some sort of *names* for segments. From this point of view, there are widely differing possible design choices, especially regarding to how an address space can utter names meaningful to other spaces. It is likely that such names will be variable-length, human-readable strings. We also expect that they will be *hierarchically* structured, which allows scaling of the name space to arbitrary size. According to Section 1, what is interesting in a wide-address space about naming, is exploring the possibility of using memory addresses as names. It is straightforward to realize, however, that these cannot be the *only* names used by the system: A memory address only has meaning in one address space.

Most importantly, addresses act as *pure names* to some extent (Needham [1989]). Pure names only identify; they do not help one in finding *where* an object is. Accordingly, we may reasonably assume that whenever a segment has to be *located*, it cannot be named by means of its address, so another kind of name has to be used, for instance the ones outlined above. We may thus devise a style of interaction between processes and the system like the following: Before starting using a segment, a process has to tell the system that is going to use that segment, and names it by means of a hierarchical name — we say that the segment is being *attached* to the process. That name is used by the system to physically locate some administrative information about the segment, including its virtual address and access rights and, most importantly, *where* the contents of the segment are stored.

Another important consequence of the fact that addresses behave like pure names, is that resolving an *address fault* is not obvious. Even without entering in too many details about the possible structure of the kernel, we may think of a faulting address as one that does not lie within any of the segments currently attached — in the sense mentioned above — to the faulting process. Upon such a fault, the kernel would have to figure out whether that address is really an invalid address, or one of a segment that exists and is not currently attached to the process. In the latter case the actual contents of the segment have to be located. That means that given an address, the system must always be able to find out the hierarchical name of the segment containing that address, if it exists. Differently stated, the system must have the capability of performing a complete reverse mapping of the address space. Apart from the fact that in a 64-bit address space this might be computationally expensive, the real problem here is related to security. Namely, since tables for performing reverse mapping in such a space are likely to be too large to be always kept in memory, we expect that at least part of those tables will need to be stored on some secondary storage. Due to the problems raised by the shared nature of virtual addresses (Section 2.3), for a system to be able to do a reverse mapping of the address space, we thus expect that the need of equipping that system with some *trusted* piece of secondary storage will arise.

The problems raised by address fault resolution could be tackled by simply removing the assumption of reverse mapping capability: Whenever a process takes a fault on an address does not

lie within any of the segments currently attached to it, the system simply raises an exception. In this way, processes would know that segments cannot be attached upon faults, therefore applications should have to be structured in such a way that segments are attached before their actual use. This would probably be reasonable, but it would make it much more difficult to think of an address as a name: If process P_1 passes a pointer to process P_2, there is no guarantee that that pointer is meaningful to P_2, i.e. that it refers to a segment attached to P_2.

2.5 Sharing across Address Spaces and Secondary Storage

A wide-address-space architecture by itself, seems not to provide any feature that may facilitate information sharing across processes running in *different* address spaces. Both the performance bottlenecks identified in Section 1 cannot be circumvented: Marshalling, because different spaces can be associated with different hardware architectures; naming, because address spaces allocate addresses independently from each other[4] The same piece of information will be placed at different addresses in different address spaces.

Apart from any motivations dictated by performance, however, it would be interesting to see whether wide-address-space architectures make it easier to share information on secondary storage. Today's applications have marshalling code that was written explicitly by the programmer. Data is often stored in ASCII form and converted to another form when read in (a file of integers, for instance). Although we may reasonably assume to reproduce such a situation even in systems based on 64-bit architectures, it would be nicer to achieve a tighter integration between the virtual memory system and the secondary storage, i.e. to allow several address spaces to think of a given "file" as of a temporarily inactive "segment".

From this point of view, the problem is deciding on whether the secondary storage has to be conceived as part of an address space or not. In the former case, files would be basically dumped images of memory segments, i.e., they would contain data in the format that is understood by the architecture of the machines composing the address space; in the latter case files would be instead stored in a marshalled form. Hybrid organizations are possible, for instance some files could be kept marshalled — text files and those that can be always understood by the kernel itself — and some others unmarshalled.

Given the technological trends concerning relative speeds of CPUs and disks, performance hardly suffers from the overhead imposed by the marshalling/unmarshalling process. The system itself would unavoidably become more complicated, however: Whereas we can assume that the system is somehow able to understand the format of a few basic segment "types", definitely we cannot assume that the system knows about the format of *every* segment. We have thus to devise an organization in which the system may need to rely on some other entity to marshal/unmarshal the contents of a given segment, for instance a process executing some code provided by the entity that created that segment. On the other hand, if every address space had its own secondary storage, marshalling would still be necessary for sharing across address-space boundaries.

Supporting some notion of *persistency* makes things even more complicated. Apart from the need of determining how pointers should be represented on secondary storage, there is some additional complexity demanded of the naming system. For a "file" to be thought of as a temporarily inactive "segment" that exists in several address spaces under the same name, the naming system

---

[4]. This influences also marshalling, because the contents of a segment containing pointers would have no meaning when crossing an address space boundary, even if the corresponding machines had the same architecture.

should be able to associate *different* attributes — i.e. addresses — with the *same* name, depending on where the entity that uttered that name lives. Furthermore, for scalability reasons those attributes cannot be kept "all together": If a file that has been created at the University of Twente is being used all over the world, it makes no sense to give the University itself the responsibility for "remembering" the addresses associated with all the corresponding copies in far-away address spaces.

3 Discussion

According to what has been discussed so far, the major issues raised by the emerging 64-bit architectures seem to be basically related to the problem of *where* to keep information about virtual addresses: In the setting outlined in Sections 2.1 and 2.2, virtual addresses are a resource *shared* across multiple machines, therefore they have to be managed by fully *trusted* entities. This need may be present even in a shared address space that does not have any notion of permanence: To support a complete reverse mapping capability, it seems very likely that at least part of the corresponding data structures will have to be swapped out, which make them potentially vulnerable to several kinds of attack (Section 2.4). On the other hand, without such a capability thinking of an address as a name becomes really difficult. Permanence makes things only harder, the main problem being the fact that a user-level file server providing *contents* of a segment cannot decide also on the *address* of that segment (Section 2.4).

Any security troubles related to virtual addresses would disappear by simply associating some piece of *trusted secondary storage* with every address space. Although quite reasonable, this is still a strong design choice, and it may thus be worth exploring different solutions. An alternative approach to the design of a shared address space — either permanent or not — is that of giving *applications* themselves the responsibility for performing, in a sense, reverse mapping. Every process would know the names of a set of segments — that may be called *segment tables* — containing mappings between ranges of virtual addresses into segment names, for the parts of the address space it may be interested in. Segment tables are created, modified and destroyed by user processes, and are shared by co-operating processes. Whenever a process takes an exception because of an address fault, it looks up in the segment tables attached to it, to see whether it knows which is the segment containing the faulting address; if it does, it can attach that segment and go on, otherwise it aborts execution. The key point is that the system decides on addresses of segments *without* being constrained at all by the contents of segment tables. Therefore, if a malicious server deliberately modifies the contents of some tables, this would affect only the processes that trusted that server enough to give it their tables, all the other processes would not have any trouble.

To support some notion of permanence without any trusted secondary storage, an alternative solution might be the use of *certificates*. A certificate is basically a statement signed by a trusted authority that cannot be forged (Abadí et al. [1991]). In the setting of our interest, certificates might be associated with segments to keep track of their addresses, and they would be signed by the authority in charge of the address space. Certificates can be stored together with segments themselves, i.e., on *untrusted* secondary storage: If a server attempted to maliciously modify a certificate, the system would be always able to detect that and would not accept the proposed mapping. The major disadvantage of certificates is of course performance, because decrypting a certificate takes time.

To sum up, 64-bit architectures are potentially an application platform for improving performance of information sharing and for simplifying management of complex data structures: In the former case, by using address spaces that can be shared across multiple processes and machines (Section 2.1); in the latter, by associating an address space with some notion of persistence (Section 2.2). However, 64-bit architectures provide only the *basic* support for the possibilities above, namely a huge number of virtual addresses. To figure out the best way to make use of these new architectures, other major problems remain to be solved, which are ultimately related to the shared nature of virtual addresses. Furthermore, permanence itself accounts for a great deal of complication of the system, therefore looking for a satisfactory trade-off between the possibilities above seems to be a fundamental issue.

Within the Pegasus project at the University of Twente, the design now focuses on sharing 64-bit address spaces across local groups of like machines, naming (permanent) files by hierarchical names and associating (temporal) segments with files through a so-called *attach* operation. If a file is multiply attached by different processes, the system guarantees that all resulting segments share the same address range. System-provided or user-defined *segment servers* marshal data between file and segment and have access to mechanisms for keeping copies of virtual segments in different physical locations consistent. When the last process that has a segment attached terminates, the segment can be cleaned up by its segment server and the address range deallocated. The system does, however, guarantee not to reuse the address range for a sufficiently long time.

4 References

M. Abadí, M. Burrows, B. Lampson and T. Wobber [October 1991], *Authentication in Distributed Systems: Theory and Practice*, Proceedings of the 13th Symposium on Operating System Principles, Pacific Grove, CA.

B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy [July 1990], *User-Level Interprocess Communication for Shared-Memory Multiprocessors*, University of Washington, Dept. of Computer Science & Engineering, Technical Report 90-05-07, Seattle, WA.

M. Hayter and D. McAuley [October 1991], *The Desk-Area Network*, ACM Operating System Review **25**, 14—21.

E. J. Koldinger, H. M. Levy, J. S. Chase and S. J. Eggers [November 1991], *The Protection Lookaside Buffer: Efficient Protection for Single Address-Space Computers*, Dept. of Computer Science and Engineering, University of Washington, Technical Report 91-11-05, Seattle, WA 98195.

R. M. Needham [1989], *Naming*, in "Distributed Systems", S. J. Mullender, ed., ACM Press, NYC, ISBN 0–201–41660–3.