[24] M. Herlihy, and B. Liskov, "A value transmission method for abstract data types," *ACM Transactions on Programming Languages and Systems,* October 182, pp. 527-551.

[25] A. Birrell, M. Jones, and E. Wobber, " A simple and efficient implementation for small databases," *Proceedings of the Eleventh Symposium on Operating System Principles,* November 1987, pp. 149-154.

[26] J. E. B. Moss, "The Mneme persistent object store," COINS Technical Report 89-107, University of Massachusetts at Amherst, 1989.

[27] P. R. Wilson, "Pointer swizzling at page fault time," *ACM SIGARCH Computer Architecture News,* 1991.

[28] P. J. Teller, "Translation-lookaside buffer consistency," *IEEE Computer,* June 1990, pp. 26-36.

[29] A. Silberschatz, J. L. Peterson, and P. B. Galvin, *Operating System Concepts,* Addison-Wesley Publishing Company, 1990.

[30] E. J. Koldinger, H. M. Levy, J. S. Chase, and S. J. Eggers, "The protection lookaside buffer: efficient protection for single-address space computers," Department of Computer Science and Engineering, University of Washington, Seattle, TR 91-11-05.

[31] G. E. Peterson, *Tutorial, Object-Oriented Computing,* IEEE Computer Society Press, Los Angeles, CA., 1987.

[32] R. M. Needham, and A. J. Herbert, *The Cambridge Distributed Operating System,* Addison Wesley, Reading, Mass., 1982.

[33] J. L. Hennesy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, Inc. San Mateo, CA, 1990.

[12] J. S. Chase, H. M. Levy, M. B-Harvey, and E. D. Lazowska, "How to use a 64-bit virtual address space," Department of Computer Science and Engineering, University of Washington, Seattle, TR 92-03-02.

[13] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," *ACM Transactions on Computer Systems,* February 1990.

[14] B. Özden and A. Silberschatz, "A Framework for Large Shared Address Space Systems," Department of Computer Sciences, The University of Texas, Technical Report, August 1992.

[15] R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and its protection system," *Proceedings of the Fifth Symposium on Operating System Principles,* November 1977, pp. 1-10.

[16] R. S. Fabry, "Capability-based addressing," *Communications of the ACM,* July 1974, pp. 403-412.

[17] E. Cohen, and D. Jefferson, " Protection in the Hydra Operating System," *Proceedings of the Fifth Symposium on Operating System Principles,* November 1975, pp. 141-160.

[18] R. P. Goldberg, and R. Hassinger, "The double page anomaly," *Proceedings of AFIPS National Computer Conference,* 1974, pp. 195-199.

[19] R. Oehler, and R. D. Groves, "RISC System/6000 processor architecture," *IBM Journal of Research and Development,* January 1990, pp. 23-36.

[20] H. M. Levy, *Capability-Based Computer Systems,* Digital Press, Bedford, Massachusetts, 1984.

[21] V. Abrossimov, M. Rozier, and M. Gien, "Virtual memory management in Chorus," *Proceedings of Progress in Distributed Operating Systems and Distributed Systems Management Workshop,* April 1989.

[22] M. Stonebraker, "Virtual memory transaction management," *Operating Systems Review,* April 1984, pp. 8-16.

[23] M-C. Tam, J. M. Smith, and D. J. Farber, "A taxonomy-based comparison of several distributed shared memory systems," *ACM Operating Systems Review,* July 1990, pp. 40-67.

References

[1] R. B. Lee, "Precision Architecture," *IEEE Computer,* January 1989, pp. 78-91,

[2] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R400 Microprocessor User's Manual,* 1991.

[3] Digital Equipment Corporation, Maynard, MA, *Alpha Architecture Handbook,* 1992.

[4] R. C. Daley, and J. B. Dennis "Virtual memory, processes, and sharing in MULTICS," *Communication of the ACM,* May 1968, pp. 306-312.

[5] D. M. Ritchie, and K. Thompson, "The UNIX time-sharing system," *Communication of the ACM,* July 1974, pp. 365-375.

[6] A. Tevanian, Jr., "Architecture-independent virtual memory management for parallel and distributed environments: the Mach approach," Department of Computer Science Technical Report, Carnegie Mellon University, Pittsburgh, PA, CMU-CS-88-106, December 1987.

[7] F. G. Soltis, and R. L. Hoffman, "Design considerations for the IBM System/38," *Digest of Papers, COMPCON S'79,* 1979, pp. 132-137.

[8] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell, "Pilot: an operating system for a personal computer," *Communications of the ACM,* February 1980, pp. 81-92.

[9] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, "A structural view of the Cedar programming environment," *ACM Transactions on Programming Languages and Systems,* October 1986, pp. 419- 444.

[10] A. Chang, and M. F. Mergen, "801 storage: architecture and programming," *ACM Transactions on Computer Systems,* February 1988, pp. 28-50.

[11] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter, "Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors," *IBM Journal of Research and Development,* January 1990, pp. 105-109.

10 Conclusions

We have classified possible SAS models and examined the tradeoffs among different models. We have concentrated on methods that can use memory management hardware to provide protection among processes rather than special hardware support to provide fine-grained protection within a process. Our conclusion is that the SAS models used in existing systems suffer from the management of the address space or the protection domain. Existing systems based on these models either limit the computation domains, or yield poor performance. This discussion also indicates that paged segmentation for memory management, dynamic address binding, and load time access authorization are the most adequate methods in their dimensions to be used with interprocess protection in the SAS paradigm.

We believe that the shared address space paradigm will replace the private address space paradigm for designing operating systems for 64-bit and larger address space architectures. The private address space paradigm will become obsolete for the following reason. The sum of the virtual address spaces cannot be larger than the available physical storage. The switch from 32-bit to 64-bit processors is the first turning point in the history at which the virtual address space exceeds the available physical storage with several orders of magnitude. This gap will remain since processor densities are increasing at a rate which is at least as fast as storage densities. Hence, there will never be a need again for the use of the private address space paradigm.

problems since objects maintain their virtual addresses only temporarily. Note that although reclamation of the virtual addresses of the shared address space is trivial, the problem still exist in the second level storage, where the objects are maintained with unique names. For example, if that naming context is a hierarchical directory structure, the deletion of a shared object may leave dangling symbolic references. There are several solutions to the problem, which can be implemented effectively. For example, consider the solution where a symbolic name is not reused until all references to the symbolic name is deleted. The number of possible symbolic names that system allows is typically more than the number of possible virtual addresses (at least for current systems), so that users can call their files with a meaningful name. To illustrate, consider UNIX Version 7 that allows only fourteen characters per name (in contrast to 4.2BSD that allows 255 characters). Suppose there are $2^5 = 32$ different characters. Even in a flat directory structure in which file names are fourteen characters long, there will be $32^{14} = 2^{70}$ possible symbolic names compared to 2^{52} possible virtual addresses in a 64-bit address space with 4 K pages. Thus, the consequences of reclamation is less severe than the ones in a single-level store. Note that in a tree structured directory, the same symbolic name can still be reused in other subdirectories.

Although the dynamic address binding scheme has many advantages over the static binding scheme, there is still a question as to whether the indirection to access an object, the cost of dynamic linking, and that the restriction that objects cannot contain pointers, are not to be severe. We believe that this is not the case for the following reasons:

- Any shared address space introduces indirections to access the private data. Hardware support is necessary to access the private data efficiently. The same support can be used to dereference the objects which are assigned to virtual addresses dynamically as explained in [14].

- If segmentation is used, the overhead of dynamic linking becomes almost equal to the overhead of dynamic loading.

- One can argue that it is not always desirable that objects contain virtual address pointers. Such a feature, for instance, complicates copying of an object to another object. Therefore, either language or system support needs to be available for pointer translations [24, 25]. Besides, since each process uses the same active copy of the object, the translation needs to be done only once in the case of dynamic binding.

Reclamation or revocation cannot prevent a user from simply filling up the entire address space intentionally or accidentally. This problem is much more severe than the problem of filling up the entire disk spaces. If the system administration decides that a user has occupied too much disk space, his files can simply be deleted. The disk space is immediately available for reuse. (Note that possible problems due to dangling symbolic references are much easier to solve; this will be discussed in detail in Section 9.2.) However, once the disk space and the corresponding virtual addresses are taken away from the user, the addresses have to be reclaimed from all other users that can potentially access the deleted files. A malicious user can easily degrade the system performance if reclamation is done by garbage collection and decrease lifetime of the system if reclamation is done by a lock-key mechanism.

9.1.4 Static Data

A compiled program is typically composed of a number of objects, (e.g., a code region, a private static data region, and a stack region). Each object needs to be assigned to an address, when it is first created. However, private data region cannot be assigned to a virtual address, since each process which shares the program concurrently has to have a copy of the private data at a different address. This will generate ambiguity in managing objects: some objects with only symbolic names and some objects with both symbolic names and virtual addresses.

9.2 Dynamic Address Binding

An object is loaded into a container when a process accesses it and the container is released when all processes accessing the container terminate. An object can be loaded into different containers in its lifetime, and therefore it can have different virtual addresses. This model is based on the assumption that there is another level of storage, in which persistent objects are maintained permanently and a naming scheme, in which persistent objects are identified uniquely. Since objects have no absolute virtual addresses, the processes need to access the object through a register or another memory location indirectly. A shared address space system with dynamic binding does not rule out static address binding of virtual addresses to some system resources. The choice is left as a policy. For example, system administration may choose to statically bound some system wide services to absolute virtual addresses.

Dynamic binding eliminates the problems we have discussed in the Section 9.1. Reclamation, fragmentation, potential misuse, or presence of multiple versions of objects will not cause severe

9.1.2 Fragmentation

External fragmentation of the global virtual address space will cause severe problems. The problem is much more severe if the shared address space is based on a paging scheme as in [30]. For all logical segments that are larger than a page, contiguous address spaces have to be found. If the shared address space is implemented with segmentation, the occurrence of the problem is less probable. The problem only occurs if a linear data structure is larger than the maximum segment size, in which case segments with contiguous virtual addresses have to be found. *Compaction* methods cannot be used to alleviate external fragmentation, since virtual addresses are statically bound to objects.

Static address binding will also suffer from internal fragmentation. The problem is much more severe if the shared address space is managed with segmentation. In order to make segmentation effective, a large segment size has to be selected. For example, in the Hewlett-Packard PA-RISC, the segment size is 2^{32}. The number of possible segments in this architecture is 2^{32} (four billion). If the addresses are bound statically, the system can have only 4 billion segments during its entire lifetime. The granularity of objects changes widely between applications. For example, some object-oriented software systems typically deal with very small objects. In the Smalltalk-80 system, the average object size is 16.38 words [31]. Although it is expected that either the compiler or the programmer will gather the related objects into one segment, internal fragmentation of the address space may significantly reduce the capacity and lifetime of the system even in 64-bit architectures. The internal fragmentation is less severe if the shared address space is managed with paging. However, such an implementation suffers from external fragmentation, high overhead for managing the protection domains and for memory allocation, and from the inflexibility that a logical segment cannot grow beyond the allocated range when the segment is first created as explained in Section 8.

9.1.3 Revocation and Maintenance

A subtle selective revocation problem arises in the case of maintaining different versions of an object. If a new version of a object is installed, and if only a selected set of ex-users are allowed to access the new version, whereas the other users need to continue to use the old version, the virtual addresses of the two versions will conflict. Hence, it is not possible to support multiple versions of the object in an absolute virtual address space, unless all ex-users are only allowed to access the version installed before their own compilation time.

users are not allowed to create and delete objects freely, this argument is indeed valid. However, in a general purpose computing environment, if the addresses cannot be reclaimed, scalability and misuse will result in severe problems. There are several ways to implement reclamation, but these incur high space or time overhead or require additional hardware:

- **Garbage collection.** When an object is deleted, or at regular intervals, the operating system performs garbage collection to reclaim the virtual addresses. The entire file system has to be traversed to detect the dangling references. Since the addresses are statically bound, it is not sufficient to search the directories. The contents of files have to be searched for the addresses. This is extremely time-consuming. In order to reduce the garbage collection time, the operating system may keep track of the objects that a program references. This incurs $O(n^2)$ overhead in space where n is the number of objects in the system.

- **Reference count.** The operating system does not reuse the virtual address until all references to the object are deleted. The system keeps a count of the number of programs refencing an object. This count is incremented each time a program links the object and decremented when a program deletes the reference. It is possible to securely increment this count, if the operating system is involved in linking. However, the count is decremented when the client itself is deleted, or it is assumed that the client will release the object at some point of time and decrement the count. The trouble with this approach occurs if the client never releases the object, or the client itself is never deleted. In this case, the address can never be reused.

- **Lock-key scheme.** Each object maintains a unique bit pattern, called a lock, along with its absolute virtual address. A process needs to have that unique pattern (key) to access the object. When the object is deleted and the virtual address is given to another object, the lock is changed. An object that uses an out-of-date key is destroyed. Keys should not be directly accessed by the application. One way to preserve the integrity of keys is to use tagged architectures. Another way is that the operating system maintains the keys for each program. This incurs equivalent overhead as the first method for reclamation. It is also possible to build the lock-key scheme in hardware. This is equivalent to expanding the virtual address space. The IBM System/38 uses this method. The System/38 supports a 64-bit virtual address space. The segment size is $2^{16} = 64$ K bytes. The system hardware only supports a 48-bit physical address. Hence at any one time, there can be 2^{32} segments in existence. The remaining 16 bits of the segment number is used to reuse the 48-bit addresses.

to maintain the entire protection domain table in the main memory, if it is small. Otherwise, several disk accesses may be necessary to locate an entry in the protection domain table.

Segmentation also simplifies dynamic linking and loading, and sharing. Dynamic linking only requires binding an object to a virtual segment number. The overhead of dynamic linking becomes almost equal to the overhead of dynamic loading. However, paging requires relocating all pages in a logical segment. Similarly for dynamic loading, only access rights of a segment need to be considered as opposed to access rights of all pages constituting a logical segment. With the same reasoning, segmentation simplifies passing data structures between processes.

9 Address Binding

We have categorized methods for managing a shared address space into two groups – static and dynamic. In this section, we compare static address binding with dynamic address binding. We use the following criteria for the evaluation: ease of reclamation of virtual addresses, fragmentation, ease of maintenance of shared objects and managing private data of programs.

9.1 Static Address Binding

An object is loaded into a container when it is first created and is stored in the same container during its lifetime. This scheme is called as a *single-level store,* since there is no need for another level of storage and naming context for objects, other than the shared address space. There is a one-to-one mapping from objects to containers, (virtual addresses). Since objects have absolute addresses, this scheme eliminates the need for indirect addressing to access an object, and thus there is no need to dereference the object through a register or memory location. Although there is no need for symbolic names, the symbolic name is a convenient way for users to reference an object. However, compilers can bind segments statically to a program.

Static address binding, however, suffers from a number of drawbacks which will limit the computation domains that it can be used for. Below, we discuss these drawbacks.

9.1.1 Reclamation

The virtual address should be reclaimed once an object is deleted. One can argue that a 64-bit address space is sufficiently large, so that there is no need to reuse a virtual address even after an object is deleted. In certain computation domains, for instance in a dedicated database system, if

allocation scheme is open to misuse and will rapidly increase the size of the tables for memory management.

- By the same token, in order to expand the size of a logical segment, it may be necessary to move logical segments to different addresses. For this reason, the Opal operating system restricts a logical segment not to grow beyond the address range allocated to it initially [12].

- Since there is no support for address translation for logical segments, paging will slow down the memory references. For example, if the logical segment boundaries needs to be checked, it has to be done in software.

8.5 Fragmentation

Both paging and segmentation can cause internal fragmentation of physical memory, as well as internal fragmentation of the virtual address space. Furthermore, paging can also result in external fragmentation of the virtual address space due to variable sizes of the logical segments. The magnitude of the problems due to fragmentation of the virtual address space depends on the scheme used for address binding. We will discuss this issue further in Section 9.

8.6 Support for the Protection Domains

In the SAS paradigm, although an address space is shared among all processes, each process must have its own protection domain. Segmentation inherently provides this abstraction:

- It is easy to observe that any implementation of protection domain with paging will result in more space overhead than an implementation with segmentation. In paging, the system has to maintain different access rights for processes on each page, whereas in segmentation, access rights need to be maintained only for segments. For example, consider a process consisting of four segments: code, data, stack and a single file. Suppose that these segments add up to 2^{24} bytes, and that the page size in the system is 4K. With paging, the space overhead for the protection domain of the process is 4K, whereas with segmentation, it is only four.

- In the case that access rights are authorized at load time and a table is maintained for each protection domain, paging will yield poor performance. The protection domain for a process consists of a nonlinear table. The search time in this table will be longer with paging compared to segmentation, since the table will be much larger with paging. Furthermore, it is possible

used to reduce the search time in a nonlinear page table. We will compare segmentation in which a linear segment table is maintained for the allocated segments of the virtual address space, with a paging scheme where dynamic hashing is being used. One can view segmentation as hashing, in which the hash function is the virtual segment number and argue both will result in same number of disk accesses to locate a data. However, the problem with paging is that the size of the hash table can grow beyond the size of the segment table if a simple virtual memory allocation is used in which always new pages are allocated to load an object rather than searching for a variable size of container among the released pages. Furthermore, with segmentation, the bucket that contains the page table entries can be indexed since page table for a segment is linear. On the other hand, with paging, the bucket that contains the page table entries needs to be searched for the page table entry. If binary search is used, additional $O(logB)$ main memory accesses are required. In the best case, a dynamic hash function increases the space overhead by a factor of three compared to the number of allocated pages of the address space. Furthermore, deletion and insertion of pages involves more with hashing schemes than with segmentation.

8.3 Hardware Cost

Only a small amount of additional hardware support is required for segmentation compared to paging. The virtual address generated by the processor has to be checked at some point in time to verify that the address is within the segment. This point in time where such a check is made depends on the system. In any case, the processor needs to have a comparison logic to check whether the offset of an address is smaller than the segment size. In architectures such as the Intel 80386 and the IBM RISC System/6000, there are also segment registers. These registers add to the hardware cost, but they are provided for the sole purpose of expanding the address space in a processor with 32-bit internal data paths, rather than for segmentation.

8.4 Address Space Allocation

In a large address space system, objects of different sizes will be mapped into and removed from the address space. Forming logical segments on top of paging incurs very high overhead to impose such a dynamic structure on the address space.

- Address space allocation is difficult with paging, since contiguous pages need to be found a logical segment that spans multiple pages. The counter argument can be that the address space is large enough not to reuse the released virtual pages. However, such a memory

Another possible method to reduce the space overhead is to maintain a linear page table that grows and shrinks according to the system load. In order to maintain a table which is not much larger than the number of allocated pages, memory allocation needs to be controlled, such that the logical segments are built from the released pages, and the address space is reorganized from time to time by compaction methods. However, this scheme will result in a time-consuming virtual memory allocation due to the dynamic structure of the large address space. To avoid a time-consuming memory allocation scheme, pages can be allocated from unused virtual addresses. In this case, the page table size will grow very fast beyond the number of allocated pages.

With segmentation, each page table can be kept as large as necessary, but the page table can be still linear. Even in this case, the size of the segment and page tables can grow to a size that can no longer be kept in main memory. In a 64-bit virtual address computer, such as the HP Precision Architecture, with a 2^{32} segment size, the segment table requires 2^{32} (4 billion) entries. One solution is to page the segment table as in MULTICS [4]. A number of bits of the virtual address indexes a main memory resident table which in turn points to a segment table. The other solution is to use an inverted page table. However, the copy on disk can still be implemented with two levels and be linear. It is also possible to maintain a linear segment table only for allocated segments to further save space. To be able to keep the table as small as possible, the allocation of virtual addresses needs to be controlled. This is trivial, since any object can be loaded into any segment and since segments have fixed virtual addresses. The size of the table can be changed depending on the load of the system.

8.2 Performance

If the page table is linear, paging results in two memory accesses to reference a memory location in the case of a TLB miss. One access is to index the page table with the virtual page number and the other access is to reference the data. If the segment table is linear, then the TLB miss penalty is three memory accesses: access to the segment table, access to the page table, and access to the desired data. However, in a large address space, it is not possible to maintain a linear table in main memory. If an inverted page table is used, then the TLB miss penalty is same for both paging and segmentation.

The difference in performance shows up when the table in main memory does not contain the needed page. Since it is impossible to maintain a linear full size page table on secondary storage, it may take several disk accesses to locate a page table entry. Hash functions or B-trees can be

[2, 3]. Segmentation has not been used widely for 32-bit architectures for the following reasons. First, when the address space is small, paging results in better performance. Second, in a 32-bit architecture there is not enough room to structure the address space, as small segments are not effective. However, paging yields poor performance and high space overhead for large address spaces, and does not provide adequate support for the SAS paradigm. In the following, we will evaluate paging and segmentation with respect to space overhead for maintaining the address space, time overhead to reference a memory location (number of main memory and secondary storage accesses to reference data), hardware cost, ease of address space allocation, and support for protection domains.

8.1 Space Overhead

In order to reduce the number of memory accesses required to reference a memory location, it is desirable to maintain linear tables for memory management. In this case, the address translation takes only the additional step of indexing into the linear table. A simple method to maintain a linear table is to have a full size table that contains an entry for both allocated and unallocated virtual addresses. For example, in a 32-bit virtual address computer with paging, and a 4K page size, a full size page table requires 2^{20} entries. Although such a table requires 1 million words of physical memory, the availability of large main memory implies that full size page tables can be used. However, in order to conserve main memory, we can use an inverted page table scheme. The inverted page table has one entry for each physical memory frame. The entry contains the virtual address in addition to the physical address. An inverted page table trades off the search time to access a page with main memory overhead. Although an inverted page table conserves main memory, a conventional page table still needs to be stored on secondary storage. In a 64-bit virtual address computer with a 4K page size, the page table needs 2^{52} (4 quadrillion) entries, which is too large to be stored even on secondary storage.

To alleviate this problem, nonlinear page tables can be maintained on secondary storage. Such a page table will have only entries for the allocated pages of the virtual address space. A nonlinear page table will increase the necessary memory for the page table at least by a factor of two, since each entry has to include also the virtual page number. Hash functions or B-trees can be used to reduce the search time in a nonlinear page table. These structures increase the space overhead by a factor of three compared to the number of allocated pages of the address space.

access an object. In this scheme, although general revocation is trivial, selective revocation becomes complicated. Reacquisition or back-pointers [29] needs to be used for this purpose.

7.2 Load Time Authorization

The access rights are not kept with the containers of the shared address space; rather, they are given to the process. The operating system maintains separate tables for the protection domains and for the address space. The access rights are given to a process, either when the process is created, or when it accesses a container first time. The operating system maintains these access rights in the protection domain table per process. The shared page or segment tables for the address space only store the information shared between processes such as address translation, segment size, etc. This scheme results in tables with fixed-size entries and enables effective hardware support for address translation. It allows multiple processes with different access rights to reference an object without traps to the operating system (note that we assume that concurrency control is implemented with other mechanisms). This scheme is used in the Opal operating system with paging. However, combining this scheme with paging for memory management, as it is applied in [12] results in poor performance due to the cost of the paging maintenance. We will elaborate on the cost of paging in the next section. We note, however, that the load time authorization can be combined with paged segmentation for memory management to obtain better performance and less space overhead.

General revocation is trivial. In order to revoke the access right to an object, the corresponding container can be invalidated in the shared address space. Reacquisition or back-pointers [29] needs to be used for selective revocation.

8 Memory Management

We have identified three methods for memory management: pure segmentation, paging and segmentation. Pure segmentation complicates both physical memory and virtual address space allocation, since contiguous pieces of physical and virtual memory need to be found to load variable size segments. It also suffers from external fragmentation of physical and virtual memory. For these reasons, we will rule out this method, and only compare paged segmentation with paging. For the remainder of this paper, we shall refer to paged segmentation simply as segmentation.

Most of the 32-bit computers have chosen paging over segmentation. This widespread agreement on 32-bit architectures has influenced most of the new 64-bit processors to support only paging

this method any further here. Thus, we compare load time access authorization to access time authorization of access rights.

7.1 Access Time Authorization

The access rights for the protection domains are maintained with the containers of the shared address space. The operating system maintains a table to manage the address space. If there is an entry for each container, then each entry also contains a list of process identifiers and the access rights. When a process issues an operation on the container (e.g., read, write, execute), the list is checked for that process identifier. If the process identifier is in the list, the corresponding access rights are checked against the type of the operation that process has issued. This scheme results in a table with variable size entries, which is difficult to maintain. The more severe problem with this scheme is that it complicates the hardware support for memory management. In order to decrease the number of memory accesses to reference an object, a processor must contain a TLB with variable size entries. It is impractical to implement a hardware table with variable size entries. However, with the access time authorization scheme, revocation can be effectively and efficiently done. The access right to an object can be revoked from a process by simply deleting its process identifier from the list of process identifiers for the corresponding container.

Some recent SAS systems use variants of this scheme by trading the run time efficiency with the ease of maintenance and hardware support [10, 11]. The AIX operating system maintains only one process identifier and one set of access rights in the inverted page table. If the process identifier does not match with the identifier in the table, a trap occurs to the operating system. If the process has the right to access the object, then the operating system updates the address translation tables. In a uniprocessor system or in a single protection domain system, such as Pilot [8] and Cedar [9], this scheme can be effectively used. However, in a multiprocessor system, where more than one process may access an object at the same time, this scheme will result in excessive operating system traps and serialize the accesses to objects.

Another variant of this scheme is to associate a protection identifier with each set of access rights to the object, and distribute them to the processes. Only one set of access rights and protection identifier are maintained in the page or segment table. The operating system updates the table if a mismatch occurs. A similar scheme is used in the HP Precision architecture [1], in which a processor has four registers to maintain four different protection identifiers. This scheme is not adequate for multiprocessor systems, since processes with different access rights may concurrently

the protection policies they can enforce, and some of them [4] even cannot guarantee the basic need-to-know principle.

Recently, the intraprocess protection is emphasized with the popularity of the object-oriented paradigm. The main goal is to prevent the programming errors rather than misuse. This goal can efficiently achieved with language-based protection in systems with interprocess protection. Furthermore, in the case that objects need to be protected due to potential misuse, objects can be mapped into separate processes.

Although intraprocess protection can protect objects mapped into a process from each other, while in the interprocess protection the system can only protect the objects mapped into separate processes from each other, we can use language-based protection mechanisms to enforce intraprocess protection in a system which only provides interprocess protection. Language-based protection is effective for programming errors, however it fails to prevent the misuse by untrustworthy users and the programming errors if the assembly language is allowed to use along with the high-level language which imposes the protection. However, there are still approaches to circumvent both. In order to prevent potential misuse and errors due to assembly language in a system with only interprocess protection, only those objects that are allowed to access each other can be mapped into a process. The other processes can send messages to the process that contain objects to access the objects. Another method to prevent the programming errors due to the use of assembly language is that the system requires the use of only high-level languages [7].

In the remainder of the paper, we concentrate on methods that can used with interprocess protection, since we believe that intraprocess protection should be implemented outside the operating system by language-based or application level protection for efficiency reasons. Therefore, we consider only the methods that use basic access rights control of the memory management to provide interprocess protection.

7 Access Rights Authorization

We have categorized methods for managing a protection domain into three groups according to the time when an access right is authorized to a process. Since compile time authorization requires a tagged architecture in addition to the hardware for supporting memory management, we do not consider this method as a viable option for managing a protection domain, and will not consider

			IBM System/38	Pilot, Cedar	IBM AIX and CPR	Opal
Protection Domain	Access Rights Autho-zation	Compile Time	x			
		Load Time				x
		Access Time		x	x	
	Granu-larity	Intra-Process	x			
		Inter-process		x	x	x
Shared Address Space	Address Binding	Static	x	x		x
		Dynamic			x	
	Memory Manage-ment	Segment				
		Page		x		x
		Paged Segment	x		x	

Table 1: Taxonomy of shared address space systems.

store (e.g., unique symbolic name), but can obtain more than one virtual address during its lifetime.

5.3 Existing SAS Systems

Although any combination of the above four methods methods can be used to implement an SAS system, some combinations are more adequate than others. Some of the combinations are in fact already implemented in existing SAS systems. The SAS model with intraprocess protection has been explored in capability-based architectures such as the IBM System/38 [7]. The IBM System/38 uses static address binding and both compile and load time access right authorization. The SAS model with static address binding is used in Pilot [8] and Cedar [9]. However, they have only one protection domain. The IBM CPR [10] and AIX Version 3 [11] operating systems favor dynamic address binding. The Opal operating system [12] uses static address binding and memory management with paging. The classification of the SAS operating systems is shown in Table 1.

6 Protection Granularity

The main goal of protection is to support the *need-to-know* principle – a process should be able to access only those objects it requires to complete its task and that it has been authorized to access. This principle is useful in limiting the amount of damage that a faulty or untrustworthy process can cause.

There are two complimentary approaches for the protection of objects (to control access to objects) – intraprocess protection and interprocess protection. In the first case, the system (hardware or the operating system) can protect objects mapped into a process from each other, whereas in the latter case, the system can only protect the objects mapped into separate processes from each other.

System-provided intraprocess protection has been used in systems with sophisticated protection mechanisms such as the capability-based systems [20] and hierarchical ring structured protection systems [4]. These systems aim to prevent intentional violation of access control by untrustworthy users as well as the programming errors. On the other hand, they require extra hardware and yield poor performance to enforce sophisticated protection policies. Furthermore, they are restricted in

contiguous pages. The size of a container can be larger or equal to the size of the object. These containers are called *logical segments.* The virtual address of the first page determines the globally unique name of the logical segment in the shared address space. Since there is no hardware support for logical segments, the boundary or other checks for logical segments have to be done in software by the operating system. Paging simplifies physical memory allocation, since a page can be mapped into any of the available frames. However, paging complicates address space allocation, since logical segments have to be built from contiguous pages.

- **Paged Segmentation.** Paged segmentation is a combination of paging and pure segmentation. The virtual address space is divided into fixed-size segments, and each segment is in turn broken into fixed-size pages. The size of a segment can be selected depending on the size of the object. The size of the segment can be a multiple of the page size; hence, the size of the segment can be equal or larger than the size of the object. Paged segmentation simplifies the address space allocation, since any of the segments can be picked for an object. It also simplifies the physical memory allocation, since a page of a segment can be mapped into any of the available frames.

5.2.2 Address Binding

Address binding has severe implications in the SAS paradigm, since once an object is loaded into a container, that space cannot be used until the object releases the container. There are two complimentary ways of dealing with address binding.

- **Static.** A virtual address is assigned to an object when it is first created. The object maintains the same virtual address during its lifetime. In other words, an object is loaded into a container when it is first created and stays in the same container during its lifetime. Note that an object can be created at compile or run time

- **Dynamic.** A virtual address is assigned to an object, when the object becomes active, namely when at least one process accesses the object. In other words, an object is loaded into a container when the object becomes active. The container is released when all the processes that access the object terminate. A persistent object has a unique name in the permanent

- **Load Time.** The access rights are authorized either when the process is created, or during execution time when the process first accesses and loads the container into its protection domain. Once the process has the access rights, the subsequent references to the object are not validated. In MULTICS, access rights to a segment is authorized when the segment is loaded into the protection domain of a process. Similarly in Unix, the access rights are assigned to a process at load time.

- **Access Time.** The access rights are never given to a process and they are verified at each access. This scheme is also called as the *access-control list* approach. The identifiers of those processes that are allowed to access a container are maintained in a list along the container. The list is checked at each access. The inverted page tables in the IBM AIX [11] and Mach [6] operating systems are examples of this scheme. The access rights and identifiers are maintained in the inverted page table.

5.2 Methods for Managing the Address Space

There are two issues to consider for the address space management. The first one is memory management, which determines the implementation of containers and the mapping from the virtual addresses to physical addresses. The second one is the binding time of a virtual address to a persistent object, which determines when a persistent object is loaded into a container from the name space in which objects are stored permanently.

5.2.1 Memory Management

There are a number of different methods for memory management. The underlying hardware support determines the memory management algorithm.

- **Pure Segmentation.** The virtual address space is divided into containers of arbitrary sizes, called *segments*. The segment is the same size as the object. Pure segmentation complicates both address space and physical memory allocation, since contiguous pieces of address space and physical memory need to be found to load variable size segments.

- **Paging.** The virtual address space is divided into fixed-size blocks called pages, and the physical memory into blocks of the same size called frames. Containers have to be built from

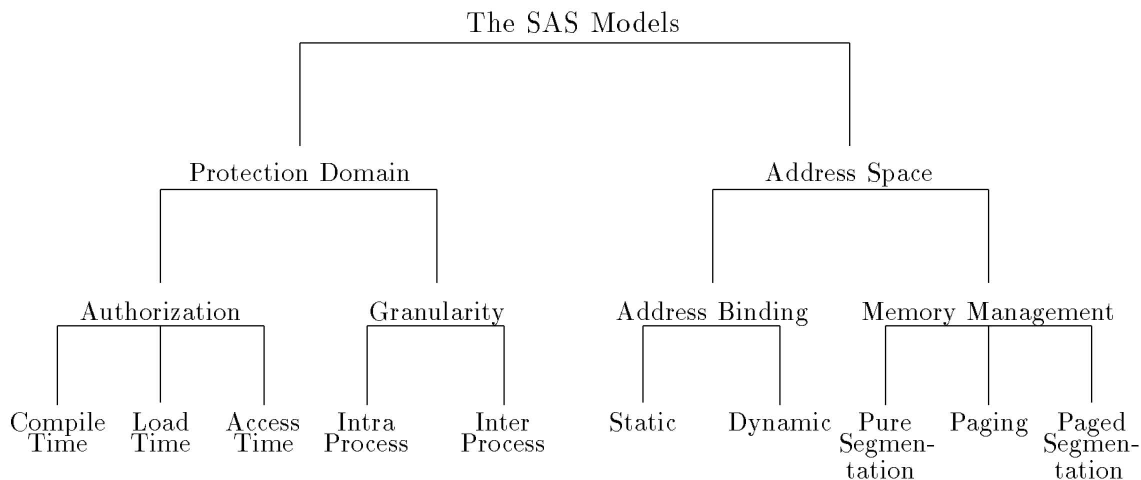


Figure 1: Classification of shared address space implementations.

5.1.1 Granularity of Protection

The granularity of protection can be classified into two categories.

- **Intraprocess.** The system can protect objects that are mapped into a process from each other. In order to support fine grain protection within a process, there has to be either additional hardware available, such as an elaborate tagged architecture [7, 15], or the operating system has to intervene to enforce the protection [17].
- **Interprocess.** The system only guarantees to protect the processes from each other. The system cannot protect objects mapped into the protection domain of a process from each other.

5.1.2 Authorization of Access Rights

The authorization of access rights o the objects that a program needs to access, can be granted at three different points in time.

- **Compile Time.** The access rights are authorized when the program is compiled. Obviously, an application program should not be allowed to modify the access rights. Therefore, compile-time authorization requires an elaborate tagged architecture [7, 15] to preserve the integrity of the access rights. The "authorized pointers" in the IBM System/38 is an example of the access rights assigned at compile time.

- If there are no homonyms and synonyms, virtual addresses can be used to implement coherency protocols in distributed systems in which a physical address is local to a processor and has no global meaning.
- Synonyms and homonyms also influence memory management of the operating system. In a PAS system, the operating system typically maintains several tables to manage the memory. There is a page table for address translation and access rights specification per process. However, if there are no synonyms and homonyms, a single table is sufficient to maintain the shared information between processes such as address translation, reference history or invalidation bit. One copy of such information eliminates the consistency problem between multiple copies and localizes updates for page replacement and allocation to a single table. Elimination of synonyms and homonyms also simplifies maintaining main memory resident tables, such as inverted page tables or frame tables. Since there are no synonyms, the entries of these tables will be fixed-size. Since there will be no homonyms, the search in these tables will be simplified.

5 Classification of SAS Models

There are different methods for managing both the protection domain and the address space. The combinations of these methods result in different SAS models that yield different performance and overhead. In this section, we briefly introduce the dimensions that classify the SAS models. Figure 1 summarizes the classification.

5.1 Methods for Managing a Protection Domain

There are two issues to consider for protection domain management. The first one is the granularity of protection, which is the unit in which the system can protect objects from each other. The second one is the authorization time of the access rights. The access right authorization determines whether the access rights are maintained with the containers, or whether they are distributed to the processes. Since both affect the system performance, we classify methods for managing protection domains based on these two dimensions.

adds extra overhead. Furthermore, if copying is not necessary and only one copy needs to be maintained, the coherency problems can arise between multiple copies.

Similarly, sharing persistent objects that contain pointers to other objects becomes difficult in the PAS paradigm. One method is the use of surrogates instead of pointers [26]. Surrogates are not virtual addresses but they are defined in some external naming context and interpreted by the application level software. Examples are the unique object identifiers in object-based systems and symbolic file names. Surrogates are interpreted at reference time. If only one copy of the data structure has to be maintained, each reference to a surrogate needs to be interpreted, since the translation to a virtual address will be different in each process. Methods such as swizzling [27] reduce the cost of interpreting the pointers by translating them once and using the same translation for further references. However, such an optimization requires multiple copies of the data structure, one for each sharing process.

- In the SAS paradigm, there exist no *synonyms* and *homonyms.* If more than one virtual address is mapped into the same physical address, these virtual addresses are called as synonyms. If the same virtual address is mapped into different physical addresses in different address spaces, these addresses are called as homonyms. Below, we discuss the benefits of eliminating synonyms and homonyms.
  - Virtually addressed caches can be used effectively. In a PAS system, homonyms generate inconsistency in the cache. Inconsistencies can be resolved either by flushing out the cache at the context-switch time or tagging its entries with process ID's. Tagging requires extra hardware, which is expensive. Flushing during the context switch degrades the performance due to cold-starts. In a PAS system, synonyms generate multiple copies of the same data in a virtual cache. This results in coherency problems when one copy is modified.
  - In a PAS system, the translation-lookaside buffer (TLB) becomes inconsistent at the context switch due to homonyms and different protection information for each process. Similarly, TLB inconsistencies can be resolved either by flushing out the TLB during the context switch or tagging its entries with process ID's, which degrade performance or increase the cost. Note that TLB inconsistency due to the protection information is also possible in an SAS system. However, effective solutions are possible as presented in [14].

pendently. Since virtual addresses are selected independently and it is not necessarily known at programming or compilation time which programs will run concurrently, it is difficult to coordinate the interaction between independent processes.

The following are the advantages of the SAS paradigm over the PAS paradigm:

- The PAS paradigm does not directly support the shared memory programming paradigm between independent programs.

- In the PAS paradigm, the message passing cannot be implemented efficiently. The independent processes cannot pass pointers between each other. In order to send a message, the data has to be copied from the address space of the sender to the address space of the receiver. In certain cases, copying may be desirable. However, if it is not necessary, the cost of copying increases the overhead of message passing. The optimizations to reduce the cost of passing long messages requires the involvement of the operating system, since a pointer does not have a meaning in the receiving process. For example, the Mach operating system [6] inspects the messages. If it detects a pointer, it first maps the message into its address space. When the message is received, the operating system finds an unallocated virtual address in the the address space of the receiving task and changes the page table of the task to contain the message. Finding an unallocated space in the address space of the receiving task may be complicated.

  In the SAS paradigm, the operating system needs to change only the access rights of the receiving process to the segment that contains the message. Furthermore, if the sender and receiver trust each other, the operating system needs not to be involved in the communication process. The sender can simply write the address of the message into the mailbox of the receiver. This will further reduce the cost of message passing.

- In the PAS paradigm, passing data structures that contain pointers becomes more difficult. In this case, it is not sufficient that the operating system changes the memory management tables for the receiver. The data structures have to be copied since the pointers have different meanings in sending and receiving processes. Linearizing methods can be used for this purpose [24, 25], in which the sender packs the data into the message with a different representation and the receiver unpacks the message. Besides the cost of packing and unpacking, copying

become context-independent. In other words, two processes have a common virtual address in their protection domains if and only if they share a container. This is in contrast to the PAS paradigm where a virtual address different meanings in each domain. Thus, the objects can be named with different virtual addresses in different protection domains. Two processes can use the same virtual address to refer to different objects. When two processes share an object, the physical address of the object is the same in both processes, but the virtual address of the object can be different in both processes.

We want to point out the difference between the SAS paradigm and a model based on *lightweight processes* (or threads). Lightweight processes share a protection domain. In the PAS paradigm, if processes share the protection domain they also automatically share the address space. It is also possible that multiple processes in the SAS paradigm execute in the same protection domain. We will also refer to these processes as lightweight processes (or threads). A process is represented in the operating system by a process control block, (PCB). The protection domain of the process is represented with a field in the PCB. If two processes share a protection domain, it means that they share the operating system data structure which implements the protection domain, namely the corresponding field of the PCB. In contrast, heavyweight processes in the SAS paradigm do not share a protection domain and the address space is not an attribute of a process (not a field in the PCB).

4 Advantages of the SAS Paradigm

In this section, we discuss some of the major advantages of the SAS paradigm in comparison to the PAS paradigm. The main deficiency of the PAS paradigm is the lack of appropriate mechanisms for supporting efficient communication and sharing. In addition, it also complicates the system design.

The PAS paradigm can be effectively used to support communication and sharing in a parallel programming environment, where multiple processes (lightweight or heavyweight) execute the same program. The interaction between these parallel processes is coordinated by the programmer or the compiler. In this model, sharing and communication can be implemented efficiently, since the addresses are the same in parallel processes. However, the PAS paradigm lacks effective support for sharing and communication in a multiprogramming environment, where sharing is between independent processes that may each execute code which has been programmed or compiled inde-

or persistent. There is a storage hierarchy in which the persistent objects are stored permanently and a naming context, which identifies the persistent objects uniquely (e.g., a directory structure, in which each object has a symbolic name). A *container* is a range of virtual addresses. The *shared address space* is the collection of all containers. An object is loaded at some point in time into the shared address space, and it is stored in a container in the shared address space. An object is bound to a virtual address when it is loaded into a container. The memory management algorithm of the operating system determines the implementation of containers. The size of the container can be greater or equal to the size of the object depending on the memory management algorithm.

A *process* is an environment in which a program is executed. It is the basic unit of resource allocation. The type of the operation that a process is allowed to execute on a container is called as the *access right.* The access rights to a container are defined by the access rights to the corresponding object. Each process executes within a *protection domain* (or execution domain). A protection domain specifies the containers that the process is allowed to access and the access rights on these containers. Our definition of protection domain includes the containers that are accessed by the instructions executed on behalf of the process. We want to emphasize that our definition is slightly different from the more general definition of a protection domain, which is defined as the objects that a process can access. The following example clarifies the distinction. Suppose that files are not mapped into the virtual address space and a read system call copies parts of a file into a buffer in the address space of a process. According to our definition, the copy of the file in the buffer is within the protection domain of the process but not the file itself, whereas according to the general definition, the file is in the protection domain.

The SAS paradigm consists of two components: the address space and the set of all protection domains. The protection domain is an attribute of a process, but the virtual address space is not. The virtual address space is a system-wide resource shared among processes. Containers can be loaded into a protection domain at the time when the process is generated or when the process first accesses the container.

The main differences between the PAS paradigm and the SAS paradigm are the concepts of a virtual address space and protection domain. In the PAS paradigm, the address space and protection domain are the same concept, so the PAS paradigm consists of the set of all virtual address spaces. This is in contrast with the SAS paradigm where the address space and protection domain represent different concepts.

In the SAS paradigm, an address has the same meaning in all processes. Thus, addresses

- Memory utilization is increased. Higher memory utilization increases the multiprogramming level and hence, the system throughput. Main memory utilization is increased since there will be no redundant copies in main memory and since the operating system does not need to maintain another level of buffering to access files. Secondary storage utilization is increased since there will be only one copy of an object either in swapping disk or in the file system.

- With adequate virtual memory management, application domains such as object-oriented programming and transaction management can directly use the operating system mechanisms instead of using custom-designed buffering. This increases performance.

- Maintenance of shared objects is simplified. The coherency problem due to the multiple images of an object in different processes is eliminated. There is no need to recompile or relink the programs, when shared code is modified, if the same copy of executables are shared, instead of statically binding the relocatable binary code of subroutines to a program at compile time.

- The applications do not need to manage their own buffers to access files.

  - This potentially decreases the number of I/O operations, and hence, increases the performance. If the files are not mapped into the virtual address space, the application program has to write a modified buffer back into the file system, before it can replace the buffer. If the virtual page, on which the buffer is located is not in main memory, a page fault will occur to bring the page from swapping disk. This adds an extra I/O operation to access a file.
  - The same phenomenon also causes the *double paging anomaly* [18]. The anomaly refers to the increase in the number of page faults when the number of buffers is increased without an increase in the size of the physical memory. Mapping the files into the virtual address space avoids this anomaly and the system performance can be increased by extending the physical memory.

3 The SAS Paradigm

In order to define the SAS paradigm, we will first introduce our terminology. An *object* is a collection of logically related code or data, such as a file or a program. An object can be either temporary

models, which differ in the way the address space and protection domain are managed. The focus of this paper is to classify the possible SAS models and examine the tradeoffs among the different models. We concentrate on methods that can use memory management hardware to provide protection among processes rather than special hardware support to provide fine-grained protection within a process.

The remainder of the paper is organized as follows. In the next section, we examine the benefits of a large address space system. Section 3 defines the shared address space paradigm. Section 5 provides a taxonomy of the shared address space models. Sections 6, 7, 8 and 9 discuss the tradeoffs between different models and identify the problems in existing approaches to the shared address space model. Section 10 presents our conclusions.

2 Benefits of a Large Address Space

A large address space enables the mapping of files along with temporary data and code into the virtual address space. The processes share the same copy of the code and data directly from the virtual address space. The idea was first suggested and used in MULTICS [4]; more recently, the Pilot system [8], the IBM AS/400 [7], Mach [6], Chorus [21] and AIX [11] operating systems have used similar approaches. However, this concept has not gained wide popularity due to the fact that most current day systems support only a small address space.

In a 32-bit architecture, mapping large files into the address space does not leave much room for the programs. On the other hand, this problem disappears in a 64-bit address space. The following are the benefits of a large address space and sharing directly from the virtual address space:

- The performance is increased due to the following reasons:
  - Sharing one copy of code or temporary data yields fewer page faults to backing store. This decreases the I/O traffic
  - The unnecessary copying of data between processes are eliminated. For example in most operating systems, files are mapped into the operating system's memory. Even to read a file, a process needs to issue a system call to copy the file from the address space of the operating system to its own address space.
  - The context-switches between processes are reduced.

1 Introduction

Most operating systems on computers with 32-bit or smaller address spaces are based on the *private address space* (PAS) paradigm where each process has a separate address space [4, 5, 6]. This paradigm has emerged as a result of the size of the address space in 32-bit or smaller architectures, in which the number of possible virtual addresses is relatively small. In a small address space system, there can be more objects that need to be accessed by processes than the number of available virtual addresses. Thus, in the PAS paradigm, each process views the entire space as dedicated to itself, so that each process is provided with sufficient number of addresses to name the objects that it accesses. However, with the recent emergence of the 64-bit processors [1, 2, 3], the restricted private address space paradigm can be replaced with a more general one. Since the number of addresses is sufficient to name the objects that all processes access, the 64-bit processors enables the *shared address space* (SAS) paradigm in which all processes execute concurrently in a shared global address space [7, 8, 9, 10, 11, 12].

The private address space paradigm yields high overhead for interprocess communication and sharing, whereas the shared address space paradigm can reduce the cost of communication and provide simple abstractions to the application level to build variety of communication schemes. Furthermore, the shared address space paradigm results in reduced number of I/O operations, increased memory and secondary storage utilization, and simplifies the operating system and hardware design in comparison to the PAS paradigm. It also facilitates direct operating system support for application domains such as databases and object-oriented systems.

We believe that the shared address space paradigm will replace the private address space paradigm for designing operating systems for 64-bit and larger address space architectures, and that the private address space paradigm will become obsolete. The reason for this is the following. The sum of the virtual address spaces cannot be larger than the available physical storage. The switch from 32-bit to 64-bit processors is the first turning point in the history at which the virtual address space exceeds the available physical storage with several orders of magnitude. This gap will remain with the technological trend in which processor densities are increasing at a rate at least as fast as the storage densities. Hence, there will be never a need again for the private address space paradigm.

There are two key issues in the design of a shared address space system— address space management and protection domain management. The SAS paradigm can be classified into various

A Taxonomy of Shared Address Space Systems[1]

Banu Özden
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712

Avi Silberschatz
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

The availability of 64-bit processors enables operating systems the use of the *shared address space* (SAS) paradigm where all processes execute in the same address space. This is in contrast to most existing operating systems that use the *private address space* (PAS) paradigm where each process views the entire space as dedicated to itself. The PAS paradigm yields high overhead for interprocess communication and sharing, whereas the SAS paradigm can reduce the cost of communication and provide simple abstractions to the application level to build variety of communication schemes. Furthermore, the use of the SAS paradigm results in increased system performance. The SAS paradigm can be classified into various models, which differ in the way the address space and protection domain are managed. In this paper, we provide a taxonomy of the SAS models and discuss the tradeoffs between these models.

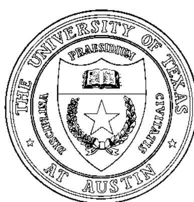
[1]This material is based in part upon work supported by the Texas Advanced Technology Program under Grant No. ATP-024, the National Science Foundation under Grant Nos. IRI-9003341 and IRI-9106450, and grants from the IBM and Hewlett-Packard corporations.

A TAXONOMY OF SHARED ADDRESS SPACE SYSTEMS

Banu Özden
Avi Silberschatz

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-92-33 July 1992



DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712