Single Address Space Operating Systems

Tim Wilkinson[1] Kevin Murray[1] Stephen Russell[2]
Gernot Heiser[2] Jochen Liedtke[3]

UNSW-CSE-TR-9504 — 13 November 1995

[1] Systems Architecture Research Centre, City University, Northampton Square, London EC1V 0HB, UK, e-mail: {tim,kam}@sarc.city.ac.uk, fax: +44 171 477 8587

[2] School of Computer Science and Engineering, University of New South Wales, Sydney 2052, Australia, e-mail: {smr,gernot}@cse.unsw.edu.au, fax: +61 2 385 5995

[3] German National Research Center for Information Technology, GMD SET-RS, Schloß Birlinghoven, 53757 Sankt Augustin, Germany, e-mail: jochen.liedtke@gmd.de, fax: +49 2241 14 2105

Abstract

Single address-space operating systems offer many advantages for modern systems design. We outline in this paper how such a system deals with the issues of memory protection, user-level naming, resource management, and translation management in a large, sparse address space, as well as fault tolerance and reliability. We also explain how a POSIX compliant interface can be supported on such a system.

1 Motivation

For many years, the number of bits available for addressing memory constituted one of the most serious restrictions imposed on programmers. While the introduction of virtual memory made it easier for programmers to deal with the limited amount of physical memory available, even virtual address spaces were too small to address all of the data needed by a program.

As a result, current operating systems provide a large number of different address spaces. In a time-shared system, each process has its own address space containing the memory objects on which the program can operate directly. However, a large amount of data on which programs need to operate are outside this address space. This especially includes all *persistent* data, i.e. data whose lifetime is independent of any particular process. Such data is generally kept in *files*.

A file represents an address space of its own. A data item within a file is addressed by its position relative to the beginning of the file. However, such an address is different from a normal memory address as it cannot be used by an instruction to access the data. Hence, the system has at least two different *naming mechanisms*. Data in virtual memory can be named and accessed simply by issuing its virtual memory address, while data in persistent memory is identified by a file name and an offset, and complex operations are required to make the data accessible.

These non-uniform access mechanisms also significantly complicate the long-term storage and sharing of some types of data. Imagine a program which constructs a dynamic data structure, such as a binary tree. The data structure is composed of a large number of memory objects which have been dynamically allocated, and the different sub-objects are connected with pointers. These pointers are virtual memory addresses whose values are not, in general, under the control of the application program, and depend on memory allocation calls performed previously by the process.

Now suppose the program tries to save the whole data structure in persistent memory (i.e. a file) so that it can be retrieved later. This presents a serious problem: the pointer values (addresses) have meaning only within their original address space. When moved into a different address space, they become meaningless bit patterns. If the file is read back by a later execution of the same (or a related) program, dynamically allocated memory objects end up at different addresses, and the pointers are invalid. Similar problems occur if programs attempt to pass data structures via an inter-process communication channel.

Two basic strategies exist for dealing with these problem. One is to convert pointers into position independent references for storage or communication. This process is called *flattening*, and must generally be done by the programmer. The alternative is to store pointers in a portable form, then translate them automatically when they are used, a process called *pointer swizzling* [Wil91]. Pointer swizzling is only possible if the system is able to detect all pointers. This imposes significant restrictions on pointer use, which are generally incompatible with languages like C.

Shared memory offers a partial solution to these problems. Complex data structures can be shared, but only if the shared memory region resides at the same virtual address for all participating processes, both currently *and in the future*. Reaching agreement on the addresses to share is not always possible if more than a few processes are involved. Further problems are that all objects in the shared address range will have the same protection state, and it is difficult or impossible to allow sharing of just parts of the memory.

The need to move data between multiple address spaces results in programs that are slower, more complex, and less able to cooperate effectively. These problems could be avoided if all data were put into the *same* address space. Obviously, such an address space must be large, which is the primary reason such an approach has not been used in the past.[1] The advent of 64-bit computer architectures, such as the HP-PA, the MIPS R4000, or the DEC Alpha, has now made this approach feasible. A 64-bit address space is big enough to allow the unification of all data, transient as well as persistent, on all nodes of a distributed system of thousands of machines. In such a single address space operating system (SASOS), there is a single, system wide name for each object — its virtual memory address. Sharing in such a system is trivial, as knowledge of the address is all that is required for accessing shared data.

The single address space also incorporates all persistent data. In a traditional OS, a memory object only exists in the creator process' address space, and vanishes with that address space when the process exits. In a SASOS, however, the address space persists throughout the life of the system, and hence objects allocated in the address space persist as well. As a consequence, a SASOS needs no file system, and non-volatile (disk) store is nothing more than backing store for virtual memory paging.

In this paper we present some of the issues relating to SASOS design, and present possible approaches to their implementation. We use two SASOS as case studies: the Angel system [MSS+93] developed at City University, London, UK, and the Mungi system [HERV93], from the University of New South Wales, Sydney, Australia. A native 32-bit prototype of Angel has been in operation for about a year, while a native 64-bit prototype of Mungi is presently under development. A similar system, Opal [CLFL94] from the University of Washington, Seattle, USA, has recently been implemented on top of Mach.

2 Memory Protection

While SASOS make sharing of data easy, this must not happen at the expense of security. In a traditional OS, memory protection is based on the fact that address space boundaries can only be crossed with the cooperation of the OS, and so access to objects external to a process' address space is under full control of the system.

[1] An exception was the Monads system [RA85] which used custom hardware to implement a large shared address space.

As there are no such address space boundaries in a SASOS, this seems, at a first glance, to weaken protection.

In fact, memory protection in SASOS is by no means weaker than in traditional systems [CLFL94]. As far as protection is concerned, the concept of an address space is replaced by a *protection domain*, which is the set of objects a process is allowed to access. As in every virtual memory system, a process can only access areas of virtual memory for which a mapping to physical memory has been established, and every attempt to access unmapped memory will result in a page fault. When the system handles that fault, it can verify whether the process has permission to access that memory region; i.e., whether it is part of the process' protection domain. If it is not, the system will generate a *protection fault*. In essence, protection in a SASOS is provided not by controlling what is *in* the address space, but by controlling what parts of this can be *accessed*.

The handling of protection faults is system specific. In **Angel** protection faults are handled by a user-level *protection server* which, in theory, can implement any protection model desired. The current implementation maintains a very general and flexible system of access control lists. Rather than using lists of `(owner, permissions)` pairs, Angel allows the construction of permissions trees, one tree per object. At the base of this tree are `(object, permissions)` pairs which are combined in the tree using AND, OR and NOT operations (see Figure 1).

When a process asks to add an object to its domain by presenting the object's address, the object's permission tree is interrogated with respect to the current state of the domain. For each leaf of the tree which matches an object in the domain, the branch is considered true (i.e. all permissions valid). These are combined and filtered using the logical operators within the tree to produce a set of permissions for the requesting process. The result is then compared with the permissions requested by the process and, assuming there are no conflicts, the object is placed into the domain. Modification of permissions trees is controlled by additional permissions bits, defining which processes can modify the trees and thus preventing security being compromised.

Mungi's protection model [VRH93] is based on password capabilities [APW86]. Capabilities may confer read, write, execute and delete rights. The password associated with each capability is a large random number chosen by the object's owner. Multiple capabilities may be created with the same rights but different passwords, allowing selective revocation at a later time. The set of capabilities for each object is maintained in a global distributed data structure, the *object table*.

The system maintains a process' protection domain in the form of a set of capability lists referred to by the process control block. When a protection fault occurs, these lists are compared to the set of valid capabilities for the object, and if a match is found, the appropriate address mapping is established.

The capability lists are user maintained data structures. Some of them are public lists, allowing access to shared objects. Others belong to individual users and contain their private passwords. When a user logs in, an initial protection domain

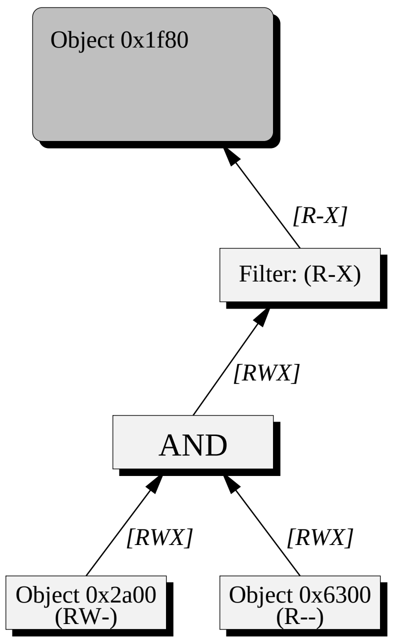

Figure 1: Example of a permissions tree for an object in Angel. In this case, a process is granted RX access to Object 0x1f80 if it has at least RW access to Object 0x2a00 and R access to Object 0x6300.

is established, and is typically inherited by all subsequent processes. The protection domains can then be manipulated by either changing the contents of the capability lists, which affects all of the processes, or by adding or deleting lists on a per-process basis.

Presentation of capabilities in Mungi is done implicitly when a process attempts to access an address for the first time. Due to implicit validation, services such as name managers do not have to store passwords, and hence can be implemented as library procedures. As well, it is possible to construct a confined protection domain which does *not* contain capabilities for some or all of its capability lists. Such protection domains allow processes to access data, but not the associated capabilities.

In order to allow user processes controlled access to privileged operations, Mungi uses a protected procedure call mechanism called *protection domain extension* (PDX). PDX procedures have capabilities which are added to the caller's protection domain on entry, and are automatically removed upon exit from the procedure.

3 User-Level Naming

A SASOS names objects by their addresses. However, 64-bit numbers are not very convenient as user-level names. A SASOS therefore needs a symbolic naming scheme which maps names to addresses. Neither Angel nor Mungi support a naming scheme as part of the kernel, allowing the user or application to choose a scheme that is most appropriate.

In practice, of course, users expect the system to provide a naming service. Both systems implement a default namespace scheme based on ideas demonstrated by Plan 9 [PPTT91]. Essentially, each process has its own tree-structured, private namespace which it can manipulate as it sees fit. Into this can be incorporated other namespaces, such as those of other processes, persistent "file system" stores, interfaces to servers, and so forth. At the leaves of these namespaces are the actual objects. It is therefore easy for a process to use its namespace to map a textual pathname to an object address or capability. This can then be presented to the kernel in order to gain access to the required object.

4 Resource Management

A system which provides persistent objects needs to be able to deal with the inevitable garbage left behind by careless users and buggy software, otherwise all secondary storage would eventually fill up with unused objects.

Angel's approach to garbage management is to build an object hierarchy. At the base of this is the *persistent root object*. All other objects are created with reference either to this object, or to another object which can itself trace a reference back to the root. However, to avoid the rigidity and limitations of a strict tree structure, additional references between objects can be added as required.

The result is a flexible object hierarchy graph in which an object persists as long as it can trace a line of references between itself and the root. Unfortunately this additional flexibility makes it impossible to use simple reference counting to determine whether an object should persist. Instead, garbage collection must be used to identify unreferenced objects and free the address space and storage resources.

In practice, each object in Angel is first created with reference to some transient parental object, most often the process itself. Consequently, when the process terminates and the process object is removed, this object will be removed as well. However, if additional references have been added, the object can persist beyond the lifetime of the process which created it; if referenced by a group of processes acting in parallel the object would persist for the lifetime of the parallel program; if referenced by another persistent object, the object can exist indefinitely. By maintaining a reference graph it is always possible to locate any persistent object which exists in the system, even if a user assigned symbolic name has been lost. How such "anonymous" objects should be treated is up the user.

The approach taken in **Mungi** is to not do automatic garbage detection, but to hold users responsible for their persistent objects. Objects created by a process are removed automatically by the system when the process exits, unless the process explicitly requests that they persist. In this case, the process must have supplied a *bank account* which is periodically charged by a *rent collector* for the storage used by the object. A *paymaster* periodically deposits funds in each account. An empty or overdrawn account cannot be used to create new objects, forcing the user to clean up. The rent collector issues an *account statement* to show users where their money goes.

The advantage of this system is that the accounting is done asynchronously. Object creation/deallocation is not slowed down by accounting, and the rent collector and paymaster can do their job at times of low system usage. The system is also freed from the need to keep track of pointers between objects. Furthermore, the rent can easily be adjusted in response to high demand, forcing users to clean up. Note that in such a situation users who have let their account balance drop low will be the first to run out of funds, while those using storage economically will be less affected.

5 Managing a Large, Sparse Address Space

One of the most serious problems facing a SASOS is managing the huge address space; in particular, implementing virtual address translation efficiently. Traditional multi-level page tables (MPT), which in a 32-bit address space typically use two levels, would require at least 5 levels in a 64-bit address space.

In UNIX, a process' address space typically consists of only two or three contiguous segments, for text, data and stack. This model is well supported by MPTs. In a SASOS, however, each object accessed by a process corresponds to a separate contiguous segment of virtual address space, hence the total number of active segments is much larger. Such sparse use of address space results in inefficient space use by MPTs. In the most extreme case of single page objects distributed uniformly throughout the address space, more memory is used for page tables than for the actual data.

A common way to deal with a large address space is to use an inverted (hashed) page table (IPT). The size of such a page table is only dependent on the amount of *physical* memory available, and is unaffected by sparsity. The problem with IPTs is that they do not support sharing very well. Suppose two processes share a large data segment with different permissions; one has read-write access, while the other has read-only access. On each context switch, all the IPT entries of that object need to be invalidated or have their write-permission bit flipped. This basically requires a scan of the whole IPT, an expensive operation.

Sharing is best supported by a hierarchical page table structure that can cope with sparse address spaces, such as *guarded page tables* (GPTs) [Lie94]. The basic idea of GPTs is illustrated in Fig. 2: The second and third level page tables in that example are are completely sparse, each one contains only one single non-nil entry.

Consequently, there is only one valid path through these two tables: if the leftmost bits of the virtual address v are 11, the following bits *must* be 1011, otherwise a page fault will occur. Guarded page tables omit these two tables and the associated

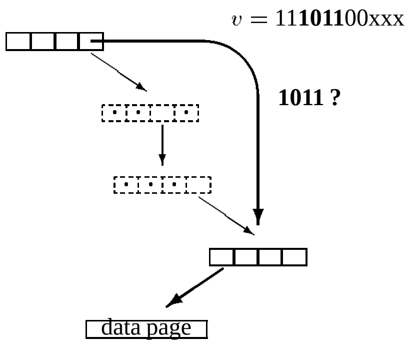

Figure 2: Guarded Page Tables

translation steps. The bit string 1011 is instead recorded as a *guard* in the first-level page. At each translation step, the leading bits of the remainder of the virtual address are matched against this variable-length guard; if they match, the guard bits are stripped off the virtual address and translation of the remaining bits continues at the next level. By recording the size of the next level page table together with the guard, the size of individual tables can be any power of two, which leads to significant space savings. Note that GPTs with empty guards work like normal MPTs.

A prototype implementation for a MIPS R4600-based system shows that the overhead of GPT lookup is reasonable as long as TLB miss rates are low. If a software cache for TLB entries is used to decrease the number of page table lookups, the GPT scheme performs well even with TLB miss rates of several percent [LE95].

6 Fault tolerance and reliability

Most operating systems neglect issues of fault tolerance. In UNIX systems this can be annoying but is not fatal to the operation of the machine. Most failures can be recovered by rebooting the relevant machine and checking the integrity of the data on disk.

In a SASOS reliability and data integrity is more of a problem. The use of a single persistent data store, shared across a network of machines, makes the failure of any single machine difficult to tolerate — the failure of one will generally result in the failure of all.

Fault tolerance in single address space systems has been studied in [Wil93]. This proposed and modeled a highly distributed fault tolerance scheme based on the observation that, unlike multi-address space UNIX-like systems, in a SASOS only one resource must be reliable—the address space. Other software structures are built on top of this, and thereby “inherit” reliability. This way, processes and their data can be preserved in the case of faults. (Obviously, this cannot prevent data loss due to broken peripherals such as printers.)

The fault tolerance scheme has three components. Firstly, the distributed memory protocol must be capable of tolerating machine failures. While data may be lost on the failed machine, the remaining protocol state must be recoverable. One way to achieve this is to maintain the protocol information in a distributed, doubly linked ring. Failure of a single node will not prevent access to other ring elements or subsequent repair.

Secondly, data must be duplicated onto different machines to prevent failure of one making some data inaccessible. The multiple writer protocol [BCZ90] used for Angel's distributed shared memory (DSM) already maintains local copies of pages. Data are also written back to multiple disks to prevent disk failures causing problems.

Finally, since it is unreasonable to propagate every piece of changed data immediately, a checkpoint and rollback policy is implemented whereby any uncheckpointed changes made by a process to the address space are discarded when a machine fails. By monitoring data dependencies in normal operation (using the DSM system), it is easy to determine which process' checkpoints depend on which, limiting the scope of the rollback to the few checkpoints directly effected. Short of using a consistent checkpointing scheme, Angel limits the amount of interdependencies by forcing checkpoints of dependent data if necessary. Failure to do this could result in "domino rollbacks", where the system rolls back indefinitely in the presence of failures.

This scheme guarantees the single address space always remains causally correct, even in the presence of failing machines. After recovery, the processes that had been executing on the failed machine can be restarted on different nodes.

7 UNIX Compatibility

A drastically new model of computer systems, such as that of a SASOS, has little chance to gain acceptance if it does not offer a migration path for existing software. It is therefore important to show that a traditional OS can be emulated under a SASOS. As a typical example we therefore discuss how UNIX can be emulated, or, more accurately, how a POSIX [POS90] compliant library can be built.

A large part of the POSIX interface deals with I/O. While no explicit file I/O exists in a SASOS, there is no reason why a file system cannot be emulated. For example, the `open` call uses the naming service to convert the user-supplied name into a memory address, and sets up a data structure containing, among other things, a current position pointer. Obviously, no explicit file buffering is required, avoiding some of the overhead inherent in traditional I/O.

In a SASOS there is no guarantee that an object can be enlarged, although a randomised memory allocation strategy makes this highly likely. However, UNIX file system semantics do not require that each "file object" is mapped onto a single, contiguous SASOS object. Furthermore, allocated memory is cheap as long as it is not being accessed (and therefore does not require backing store). It is possible to allocate a large junk, possibly many gigabytes, of virtual memory for objects which are likely to grow.

The largest problem is presented by the *fork()* system call, which explicitly duplicates the address space. Obviously, this is not possible in a SASOS where there is only one address space — identical copies of objects can be generated but they must reside at different addresses.

In practice, very few programs actually use `fork`. Those which execute other processes mostly use some higher level function, e.g. `popen()`, which can implemented in a SASOS without problems. However, no POSIX compliance can be claimed without supporting `fork`.

Fortunately, a solution is possible [WMSS93]. The program's code can be shared between child and parent without problems. However, **all** addresses in a data object are represented as offsets from the object's base address. Consequently, the data object can be duplicated for the child process, and its notion of the object's base changed. The child will then proceed to use the new copy of the data, the parent will use the old, and neither will notice the difference. The cost of this is bearable, as shown in Table 1.

Benchmark	Slowdown
Dhrystone	3.8%
Espresso	9.3%
Spice2g6	7.3%

Table 1: Performance loss experienced by a number of applications resulting from running them with `fork()` support enabled.

Some care must be taken to prevent any absolute address being used in programs which support `fork`. Most of these can be handled by the compiler and library interfaces, but stack and frame pointers must be adjusted after a fork. Similarly, direct addressing of other objects must be restricted, since this would involve more absolute addresses.

8 Open issues

SASOS are attractive for supporting distributed computing systems, like workstation environments, as they make the network transparent to users. Even process migration is quite easy in theory: if a process' register set is migrated, the resumed process will fault its working set across the network. However, the question when a process should be migrated in order to balance the load or reduce network traffic has not yet been resolved. Other research issues resulting from distribution are network security, particularly the authentication of data requests originating on remote nodes, and how a SASOS could work in a network of heterogenous computing nodes.

Computer users, including application programmers, have grown used to a view of a computer systems which has resulted from technical limitations of previous hardware generations. Each program lived in a world of its own, and any interaction with other worlds (programs or persistent data) were cumbersome. The design of most contemporary programming languages reflects this view.

In many respects, a SASOS represents a much simpler and more flexible model of computing. The question arises how this can be employed to present users with more attractive computing environments, and what an appropriate programming language might look like. A partial answer is given by object-oriented database systems, like ObjectStore [LLOW91] or O_2 [D^+91]. These systems present a very convenient data model. While their implementation is obviously possible on a traditional OS, their use of 64-bit unique object identifiers essentially simulates basic features of a SASOS. Consequently, the implementation of such a system is expected to be much simpler, and more efficient, in a SASOS.

9 Summary

The concept of a single address space presents an attractive and exciting new approach to operating systems. SASOS use recent advances in hardware architecture to overcome restrictions which previous technology had imposed on systems design. The result is greatly simplified data sharing due to a convenient shared memory programming model. Orthogonal persistence relieves programmers from the need to flatten data structures in order to make them compatible with the byte stream model of I/O. We have shown that difficulties facing the implementation of this system model can be overcome, and that it is possible to emulate a traditional computing environment such as UNIX. However, the main strength of SASOS lies in their ability to support novel environments, such as object-oriented database systems.

References

[APW86] M. Anderson, R. Pose, and C.S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, 1986.

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Conference on Principles and Practice of Parallel Programming*, pages 168–176. ACM, 1990.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12:271–307, November 1994.

[D$^+$91] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[HERV93] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. Report 9314, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1993.

[LE95] Jochen Liedtke and Kevin Elphinstone. Guarded page tables on MIPS R4600 or an exercise in architecture-dependent micro optimization. School of Computer Science and Engineering Report 9503, University of NSW, University of NSW, Sydney 2052, Australia, November 1995.

[Lie94] Jochen Liedtke. Address space sparsity and fine granularity. In *6th SIGOPS European Workshop*, Schloß Dagstuhl, Germany, September 1994.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.

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

[POS90] *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*. IEEE, 1990. IEEE Std 1003.1-1990, ISO/IEC 9945-1.

[PPTT91] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. Plan 9, a distributed system. In *European Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 43–50, Tromsø, Norway, May 1991.

[RA85] John Rosenberg and David Abramson. MONADS-PC—a capability-based workstation to support software engineering. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, pages 222–31. IEEE, 1985.

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

[Wil91] Paul R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *Computer Architecture News*, 19(4):6–13, June 1991.

[Wil93] Tim Wilkinson. *Implementing Fault Tolerance in a 64-Bit Distributed Operating System*. PhD thesis, Systems Architecture Research Centre, City University, London, UK, July 1993.

[WMSS93] Tim Wilkinson, Kevin Murray, Ashley Saulsbury, and Tom Stiemerling. Compiling for a 64-bit single address space architecture. Technical report TCU/SARC/1993/1, Systems Architecture Research Centre, City University, London, UK, March 1993.