SCS&E Report 9314
November, 1993

Mungi: A Distributed Single Address-Space Operating System

Gernot Heiser, Kevin Elphinstone, Stephen Russell, Jerry Vochteloo

(The text of this report has been accepted for ACSC-17)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

With the development of 64-bit microprocessors, it is now possible to combine local, secondary and remote storage into a large single address-space. This results in a uniform method for naming and accessing objects regardless of their location, removes the distinction between persistent and transient data, and simplifies the migration of data and processes.

This paper describes the *Mungi* single address-space operating system. Mungi provides a distributed single level address-space, which is protected using password capabilities. The protection system performs efficiently on conventional architectures, and is simple enough that most programs do not need to be aware of its operation.

1 Introduction

The advent of 64-bit microprocessors allows the investigation of a new class of operating systems, in which all the data managed by a traditional system are combined into a *single* address-space. A 64-bit address-space is not only large enough to hold all the data a process may ever want to access during its lifetime, but is even large enough for all the data available in a medium-scale distributed system; i.e. a network of hundreds of computing nodes, and it will be large enough for years to come [10].

A *single address-space operating system* (SAOS) which supports this model offers many advantages. Naming, for example, is greatly simplified by eliminating the distinctions between local and remote storage, and between primary and secondary memory. The use of 64-bit addresses as object names within a single address space makes sharing of all forms of data much easier. Accessing any data requires nothing more than issuing the appropriate address. This should be compared to existing distributed operating systems which typically offer different interfaces for accessing local and remote variables [9, 33, 35, 18]. The migration of data and processes is also greatly simplified, as addresses are location independent, and the environment a process sees does not depend on the node on which it is executing.

Another advantage of a SAOS is its support for persistent programming systems. It has been observed [4] that typically 30% of the code of application programs only serves to convert between the structured, high-level representation for data in memory, and the flat, low-level representation required for long-term storage of data in file systems. Persistent programming systems like Napier88 [24] try to eliminate this overhead by allowing the long-term storage of structured data, but have been hampered by a lack of support from traditional operating systems. A SAOS, on the other hand, directly supports persistence. Other improvements in efficiency result from eliminating unnecessary copying of data between address spaces, and by reducing the costs of context switches, a significant performance limitation for current operating systems [3].

Other groups have also recognised the advantages of providing a form of shared memory in a distributed system. These include *Distributed Shared Memory* (DSM) systems for supporting parallel programming [8, 20, 22], and *Persistent Object* systems [1, 9, 12]. While our objectives are similar to these earlier systems, 64-bit architectures allow new approaches to shared memory and persistence to be investigated.

There are three major issues to be resolved as a consequence of switching to a single address-space. The first is how best to handle distribution: how objects in the address space are located and how they migrate as needed. The second issue is how to manage the allocation of memory within the large address space, and how memory is deallocated or reclaimed. The last, and possibly most important issue, is how to provide privacy and protection of data in a system where all data are potentially visible to everyone.

The Distributed Systems group at UNSW is currently implementing *Mungi*, a SAOS designed for a medium-scale network of homogeneous machines. The system provides a single shared virtual memory where pages are transparently distributed amongst the individual nodes. Mungi features a protection mechanism based on password capabilities [2] which does not require specialised hardware for efficient operation, unlike other capability systems [11, 26, 31]. The protection scheme allows users to easily define protection domains which may be extended or restricted as required, and includes a mechanism similar to the UNIX set-uid facility.

As far as possible, we have concentrated on providing a small kernel that offers simple but flexible mechanisms, and have avoided dictating policies in areas such as object management. For example, the protection system is designed to be unintrusive and allows existing programs with no knowledge of capabilities to run without modification. A further consequence of our approach is that the concepts of addressing and protection are completely separated, which reduces the number of privileged services required by the system. All system data structures exist in the shared address-space, which simplifies the system because no additional mechanisms are required to distribute and access the data structures.

The next three sections present our proposals for the issues described earlier. This is followed by a comparison with related work in Section 5. Finally, we present our conclusions.

2 Object Location and Migration

The user's view of virtual memory in Mungi is that of a large address space sparsely populated by memory segments, called *objects*, of (almost) arbitrary size. Memory access is controlled at the object level; i.e. all bytes of an object have the same access mode for a given user. Memory allocation creates new objects which can be truncated or destroyed, but not expanded. As we want to impose as few policies as possible, the system is unconcerned with the internal structure of objects, which are left to be implemented by higher software layers.

From the system's point of view, the basic unit of memory is the page. An object is a contiguous sequence of pages and always starts at a page boundary. Pages are the unit of migration and replication, and different pages of the same object can reside on different nodes. As users only see the single global address space, this distribution is completely transparent. Whenever a process with sufficient permission attempts to access a non-resident page, the kernel handles the resulting page fault by locating the page and allowing the process to access it.

2.1 Distributed Object Directory

Information about all the objects in Mungi is recorded in a single directory called the *object table* (OT). Each object's entry in the OT contains its address and size, some accounting information, and a list of passwords with corresponding access rights. The passwords are used to validate object access, as described in Section 4.4.

The OT is implemented as a B$^+$-tree (Figure 1). Its internal nodes form an index, where the keys are the start and end addresses of the objects. Each non-leaf node contains several thousand such keys, so the height of the tree is expected to be small (3-4 levels). This data structure allows efficient translation of an arbitrary address to its corresponding object (if it exists).

The OT is itself an object in the single address-space, and is distributed and (partially) replicated. The problem of contention in accessing the shared OT is eliminated by a policy which partitions the address space for management purposes, as described in Section 3. This policy results in updates mostly occurring in the section of the OT corresponding to the node's partitions. Other accesses to the OT will be mostly read-only.

2.2 Page Tables

When a page fault occurs, the page table is consulted for the physical address of the page. This is similar to a traditional paging system, except that in Mungi the page may reside on a remote node. The page tables indicate one of six possible states of a page: resident, on-disk (locally), remote, zero-on-use, unallocated, and unknown. Accesses to resident, local, zero-on-use or unallocated pages are handled in the obvious way. Faults on remote pages or pages of unknown status are discussed in Section 2.3.

The huge size of the address space makes it impossible to store a full page table. We therefore use a sparse multi-level data structure, where the full page number is broken into a set of indices for the different levels of subtables. (The current design uses a five level page table and a software-loaded TLB.) Subtables whose entries are all identical are not allocated but are represented by the corresponding entry in the next higher level table.

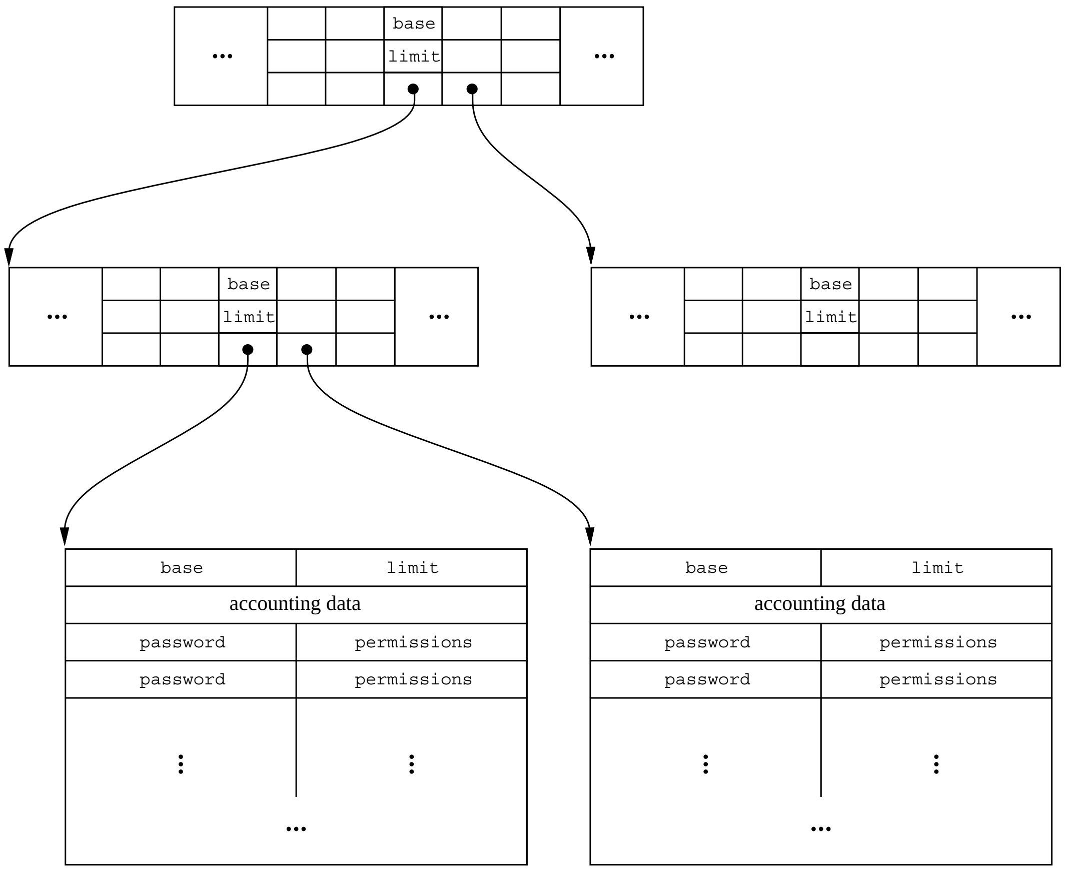

Figure 1: Structure of the object table

To test the suitability of the sparse multi-level page table we built a simulator which allows us to study the effect of memory allocation policies on page table size [13]. The simulations were driven by synthetic data based on statistics gathered from local UNIX file systems, whose distribution agreed with results of other studies [5, 29]. Our results show that the sparse multi-level page table quickly reaches a steady state where it consumes around 0.2% of the allocated memory, provided addresses are being reused (Figure 2).

If however, addresses are not being reused, page tables are at least an order of magnitude larger and the address space is much more fragmented. The simulations also indicate that under a no-reuse policy the page tables will continue to grow even when the total amount of allocated memory is not increasing.

Our design allows addresses to be reused safely, unlike other approachs which rely on unique addresses [32]. Each time a portion of the address-space is reallocated, new passwords are assigned to protect the region (as described in Section 4), invalidating any dangling references.

2.3 Page Location

Each allocated page has a unique *owner*, which is the node holding the master copy of the page. There may be additional read-only copies as well. Page ownership changes over time, and there is no way to infer the owner of a page from just its address.

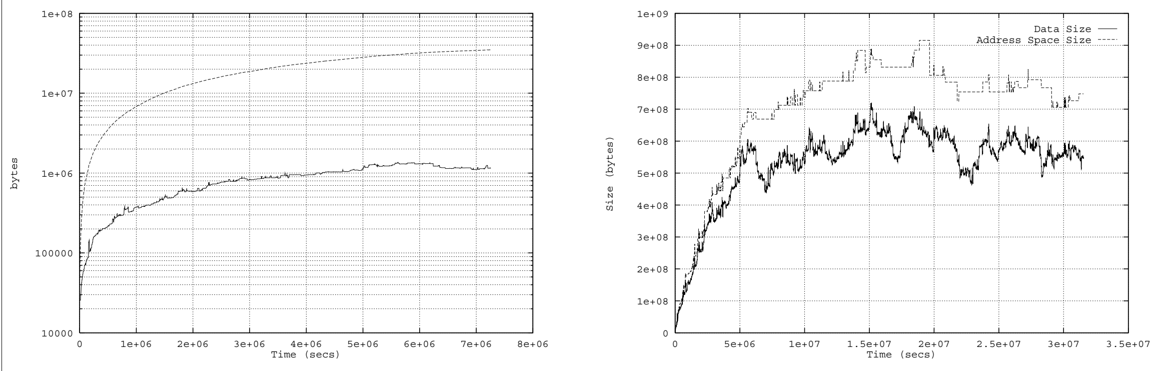

Figure 2: Page table size (left hand graph) with (bottom curve) and without (top curve) address reuse. The right hand graph shows the total amount of memory allocated (bottom curve) and the amount of address-space used (top curve) if address-space is being reused.

The page table entries for remote pages contain a *location hint* to increase the efficiency of page location. When handling a page fault for a remote page, the kernel obtains the page (or a copy) by sending a message to the node indicated by the hint in the page table. If the page is indeed owned by that node, it will send it to the requester. It not, the node will either use any location hint it has to forward the request, or otherwise broadcast the request on behalf of the requester. Locality of reference makes it likely that neighbouring pages are owned by the same node, so the location hints of neighbouring pages are used if none exist for the required page. An alternative approach employed in Monads [16] uses pointer chaining to avoid broadcasts.

If the page status is unknown, the kernel immediately broadcasts a request for the page. If there is an owner, it will respond to the broadcast. Otherwise, the page is unallocated and the node which manages the address-space partition containing that page informs the requester of the page's status.

2.4 Page Transfer

When a node requests a page from its owner, the response it receives depends on whether or not the requester is attempting to write to the page. It the request is a read, the owner simply sends a read-only copy of the page, and the owner copy is also marked read-only. A write request causes *ownership* to be transferred as well as the page's contents. To avoid losing or duplicating page ownership in the case of node crashes or network errors, a variant of a two phase distributed commit protocol [7] is used for the ownership transfer. This transfer also invalidates all read-only copies of the page. The dynamic distributed ownership management is similar to the one used in Ivy [21].

However, unlike DSM systems such as Ivy, we anticipate that sharing of pages in our system will fall into two categories: objects which are shared between very few processes, e.g. message buffers, and objects such as executable programs which are shared between almost all nodes. A page's owner therefore maintains a small list of the holders of copies for invalidation purposes. If the list overflows, the page is assumed to be widely shared and invalidation requires a broadcast.

3 Address-Space Management

3.1 Memory Allocation and Deallocation

As indicated earlier, the global address-space is partitioned for the purpose of memory management. A partition is identified by a 12 bit *address-space partition id* (API), which is the most significant bits of an address. A partition is mounted on a single node, and only that node can allocate or deallocate memory from that partition. This node is termed the partition's *creator node* and knows authoritatively which of the partition's pages are allocated and which are free. Note that the creator does not have any special knowledge of the present location of any allocated page of the partition.

When a process requests the creation of a new object, the local node will allocate a sufficient number of pages from one of its mounted partitions. All of these pages are initially marked zero-on-use in the page table, and an entry describing the object is added to the OT. Only the creator node can destroy an object; if a process on another node requests destruction, the request is forwarded to the creator node. This strategy ensures that only the creator node needs to keep track of unallocated pages. It also increases the locality of write accesses to the OT, thus allowing for its efficient distribution.

3.2 Memory Management Classes

In traditional operating systems, there exist two classes of objects: *transient* memory objects which cease to exist once the process to which they belong exits, and *persistent* objects which outlive individual processes. Transient objects are usually private, while persistent objects are potentially shared by many processes. These objects are clearly distinct—they use different naming schemes and access mechanisms.

In Mungi, the difference in naming and access mechanisms is removed. However, the fact remains that there are classes of objects which are inherently transient, like a process stack, and objects that are meant to persist for a long time, like databases or executable program code. Transient objects are likely to remain on their creator node and are seldom shared, while persistent objects are frequently shared and so are likely to migrate. To support both (and possibly further) classes of objects, different memory management policies are employed.

Mungi allows the specification of a *usage indication* when allocating an object. There are currently three such classes: transient and unshared objects (e.g. program stacks and heaps), transient and shared objects (e.g. message buffers), and persistent objects. The kernel's memory manager allocates these different classes of objects from different partitions. This makes creation and destruction of transient object quite fast, as the corresponding part of the OT is always local and never referenced by other nodes.

Each new process is initially allocated three private transient objects: a process control block (PCB), a stack and a heap. The PCB is a system object inaccessible to the user process, and contains a table of all transient objects associated with the process. These objects are automatically destroyed by the system when the process exits. All other objects are persistent in the sense that they exist until they are explicitly destroyed.

3.3 Garbage Management

One of the advantages of the global address-space is that no restrictions are imposed on the use and storage of pointers in user data structures. The system, however, cannot easily distinguish pointers from other user data, so it becomes impossible to maintain accurate reference counts, and thus automatic garbage collection based on reference counts is not possible.[1] Alternative garbage collection strategies are also

[1] Note that this applies to *allocated* memory, there is no reason why memory cannot be reused after explicit deallocation.

impractical due to the sheer size of the address-space. We are therefore investigating other methods of garbage management such as charging users for backing store via leases [15] or other economic models [34].

4 Protection

The potentially biggest problem associated with a single address-space is protection. In traditional operating systems, protection relies on the separation of address-spaces: since it is impossible for a process to address any object outside its own address-space, explicit system intervention is required to make such objects accessible. The system has full control over such accesses and can reliably impose a protection model.

In a single address-space, however, every object is visible to each process, and no explicit system interaction is required to access an arbitrary object. Different protection mechanisms which do not depend on address-space separation must therefore be employed.

4.1 Password Capabilities

Capabilities provide a location-independent mechanism for naming and protecting objects, and are therefore suited to a SAOS. As we do not want to impose any restrictions on the storage of capabilities, the obvious choice is a sparse capability scheme. Most of these, however, require encryption to prevent forgery, which makes creation and validation of capabilities expensive. We therefore chose *password capabilities* [2], which, in Mungi, consist of two parts: a 64-bit address and (currently) a 64-bit password. Each object may have several different capabilities associated with it. Possession of a capability grants the holder a set of access modes for the object.

To make the system easy to use, the protection system aims to be as unintrusive as possible. In particular, we want to be able to run existing programs without having to modify them to include knowledge of the capability system. In the normal case of a plain address being used to access memory, the associated capability must still somehow be presented to the kernel so that it can validate the access. The system must have access to the process' *protection domain*, which is defined as the set of capabilities that the process holds.

An explicit `open` operation would restrict the need for presenting capabilities to the first access only. Unlike traditional approaches, however, an `open` operation is not required in a SAOS, as there is no need to instruct the system to map files into the user's address-space. An *implicit* open operation can therefore be used, and is favoured because it is simpler and supports existing programs. This requires that the system can find the associated password when the address is first presented. The data structures to facilitate finding passwords are described in Sections 4.3 and 4.4 below.

4.2 Capability Types and Access Modes

In Mungi, objects are the basic units of protection. A capability always defines access rights for a whole object, and the kernel supports four modes of access to objects: read, write, execute, and destroy. Every capability permitting all four basic access modes is called an *owner capability*. When an object is created (i.e., virtual memory is allocated) the system returns an owner capability to the creator process. Its password is a random number generated by the system. Note that there does not exist a *unique* owner process or user for each object; *any* process which presents an owner capability to the system has permission to perform any operation on the object.

Mungi also provides a special form of execute mode, called *protection domain extension*, which temporarily modifies a protection domain. This implements a mechanism similar to the UNIX set-uid service, and is discussed further in Section 4.6.

A scheme to derive less powerful capabilities is also provided. This is similar to a method proposed for Amoeba [25] and is based on the use of one-way functions. When an object is created, the system derives a full set of capabilities from the owner capability and enters them into the OT. Unlike Amoeba, an owner can also create new capabilities and remove existing ones, which allows selective revocation of object access. Further details on the derivation scheme are presented in [36].

4.3 The Capability Tree

Data structures are needed to allow users to store and manipulate their capabilities, as well as allowing the system to quickly find capabilities while validating a memory access. Ideally, these data structures should also reflect the user's intuitive view of the protection model.

Most existing operating systems support a hierarchical model of protection. In UNIX, for example, files have three sets of permissions which govern access by the owner, the owner's group, and everybody else. This hierarchy seems to map well to actual use, as a system typically has files which are needed by everyone (e.g. programs in /bin), those which are relevant to a particular group of users (e.g. project-related programs and documents), and those which are private.

To support a similar hierarchy of protection capability are stored in a structure called the *capability tree* (Ctree), which is a single object shared by all kernels in the system. A node of this tree is called a *protection node* (Pnode) and is linked to a group of capabilities.

Capabilities for the most public objects are stored at the root node of the Ctree, while private capabilities are stored at the leaves. Group capabilities are stored in Pnodes at intermediate levels. When the system searches the tree, it will search from some Pnode all the way to the root, thus encountering the more private capabilities first. We call the set of capabilities encountered when traversing the Ctree from a particular Pnode to the root a *regular protection domain* (RPD). Note that RPDs are not strictly hierarchical, as the same capability can be stored at different Pnodes. Each user of the system has an associated RPD, which is defined by a pointer stored in the system's user directory and which points to a Pnode.

Each Pnode contains a pointer to a *capability list* (Clist). While the Ctree is a system data structure, the Clists are owned by users. Every user holding a valid capability can modify a Clist by adding or removing capabilities, thus modifying the corresponding RPDs. Of course, normal users will only hold capabilities to their private Clists (belonging to Pnodes distant from the root), while only the system administrator will hold capabilities for the root Pnode's Clist.

Each Pnode may also contain a pointer to a user-provided *protection fault handler* which provides an alternative search strategy. When the kernel encounters a protection fault handler while searching for a capability, it will perform an upcall to the handler, which then returns either a capability or a failure indication, in which case the kernel continues its search.

Users are able to add Pnodes to the Ctree. To do this, the user provides a capability for a new Clist, and a pointer to an existing Pnode which becomes the parent of the new Pnode. To perform this operation, the user must have read permission on the parent's Clist. Similarly, users may delete a Pnode if they hold a write capability to the node's Clist.

4.4 Active Protection Domains

While the Ctree reflects the hierarchy of protection that users might expect, it is not necessarily the most efficient or flexible way of maintaining protection domains. One problem is that processes cannot modify their protection domain, e.g. to restrict it before calling untrusted programs, without affecting the protection domains of other processes. Another problem is that it is difficult to determine how many pointers exist for a given Pnode, which makes it impossible to reclaim unreferenced Pnodes, and so the Ctree may grow indefinitely. This is particularly a problem with processes that terminate abnormally without removing any Pnodes they have added to the Ctree.

To overcome these problems, we introduce an *active protection domain* (APD), which is a data structure defining the protection domain in which a process is executing. APDs are similar to *local name spaces* in HYDRA [11], and *process resource lists* in CAP [27], and consist of an array of Clist and protection fault handler pointers held in the PCB. When a user logs into the system, the login process' APD is initialised from the user's RPD by traversing the Ctree, starting at the Pnode pointer found in the user directory, and copying all Clist and protection fault handler pointers into the APD. Subsequently, the user process is free to modify its APD by adding or removing Clists. Figure 3 shows the relation between the Ctree and an APD. Note that not all Clists need to appear in the Ctree.

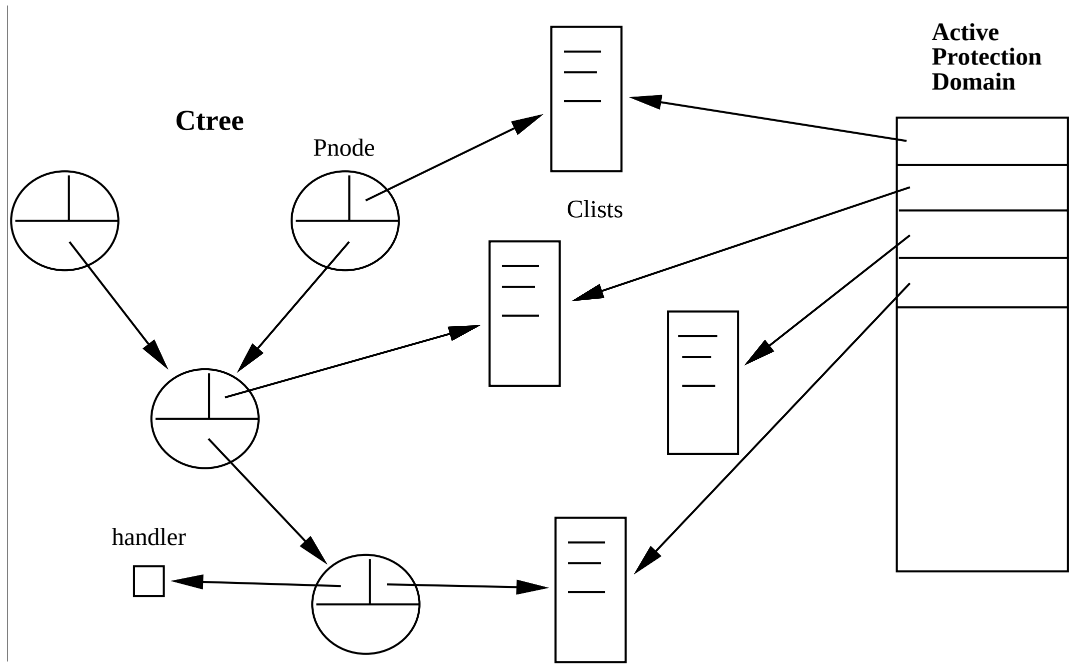

Figure 3: Capability tree and active protection domain

When an object is accessed for the first time, a protection fault is generated, as the system has no information on the validity of the access. The system then consults the OT to obtain the base address of the object containing the faulting address, as well as the passwords and access modes belonging to that object. Next, the faulting process' APD is searched for a matching capability granting at least the required access mode. If the validation is successful, the kernel's protection and translation tables are updated and the access is allowed to proceed. We expect that the validation cost will be comparable to the cost of an `open` operation on a file in a traditional operating system like UNIX.

It should be pointed out that revocation of access rights as mentioned in Section 4.2 will only prevent new mappings. Pages that are already mapped will remain accessible even *after* the corresponding capability has been revoked. This is not different from systems like UNIX, where a change in a file's protection mask has no effect on processes which have already opened the file. However, it is possible to flush cached protection information periodically to require re-validation.

4.5 Tailoring Protection Domains

Enlarging an APD by adding new capabilities is necessary to allow processes to create and share objects. Reducing an APD is essential to safely deal with untrusted programs: a user should be able to execute an untrusted program in a protection domain which contains only those capabilities the program needs to perform its duties. Both of these situations require services to initialise and modify APDs.

When a process is created, the parent provides a pointer to an initial APD data structure. This structure, for example, can be obtained from an RPD using a standard service routine, or may be specially constructed by the parent. A common case is for the child to inherit the parent's APD.

APD modifications can be performed in two ways. A process may, provided it holds the appropriate capabilities, modify the actual Clists pointed to by its APD. Such a change will, of course, affect all processes whose APDs contains those Clists. Alternatively, system calls are provided to allow the process to modify the array of Clist and handler pointers in its APD. As well, Mungi allows a process' APD to be locked, but not unlocked. A process cannot modify a locked APD, nor can it use explicit presentation of capabilities if its APD is locked.

As an example of these mechanisms in operation, consider the case of executing an untrusted program. The parent process creates a buffer which will be used to pass data to and from the program. The parent then creates a new process whose APD is locked and only contains capabilities to access the buffer and to execute the untrusted program. The child process executes the untrusted program in this restricted environment, while the parent waits for the job to terminate.

4.6 Temporary Extension of Protection Domains

In UNIX systems, it is possible to substitute temporarily one protection domain with another using the set-uid mechanism. This is extremely useful to let normal users perform special operations in a controlled manner via a privileged program. Mungi supports a a special kind of procedure, the *PDX* (protection domain extension) procedure, to perform a similar task.

Each time a user process attempts to call a PDX procedure, the system consults the object table in the usual manner to validate the execute capability for the procedure. The system then discovers that the object is a PDX procedure, which has an associated list of valid entry points and a single Clist pointer. If the call attempted by the user is consistent with the entry-point list, the system pushes the Clist pointer on the caller's APD and, prior to transferring control to the procedure, swaps the stacked return address with a dummy value. When the protected procedure returns, the dummy return address will cause a protection fault, which is caught by the kernel. The kernel restores the original APD and lets the user program resume execution from the original point of call.

The PDX mechanism is transparent to the caller, who does not need to know of the special status of the procedure. Restricting PDX access to controlled entry-points avoids security problems such as entering a procedure after its validation code. The ability to associate different sets of entry points with each PDX capability allows selective access control. Note that, unlike the UNIX set-uid facility, the PDX mechanism does allow access to the caller's environment. However, the caller can tailor a confine protection domain before calling the procedure if necessary.

5 Related Work

Some of the issues discussed in the previous section have been investigated previously. For example, Mungi shares some attributes of distributed shared memory (DSM) systems. However, it is different in emphasis. Existing DSM systems are either distributed multiprocessor systems [8, 20, 22], or provide a shared memory service maintained by the kernel [14] or by user-level servers [6, 23]. Both classes of DSM systems are specialised to support multiprocessing applications, while our goal is to provide a general-purpose operating system built on top of the single address-space. As well, the DSM systems are of limited size and scope, and do not address the problem of node failure, or of how to provide effective protection and access control for shared memory.

Mungi also shares some similarities with previous operating systems, such as Multics [28], Locus [30], and Domain [19], which allowed portions of files to be mapped into process address spaces. In Mungi, all objects reside in the single shared address space, so no explicit mapping is required. The availability of sufficiently large address spaces today overcomes the limitations that constrained these earlier systems.

More recently, other groups have also considered the design of large address-space operating systems. The Monads project [31, 17] was one of the first to investigate supporting a global virtual address space. The goal of the system was to provide direct support for software engineering principles such as modularisation and encapsulation using a specialised architecture. The shared address space is segmented, and consists of volumes, modules and objects. Access to memory is protected using a segregated capability system which does not allow reuse of addresses. Mungi differs by providing an unsegmented address space which simplifies object migration, and password capabilities can be freely manipulated by users. Our design is also based on a conventional architecture. However we do not provide direct support for encapsulation other than the basic protection system.

The Opal system [10] also supports a single large address space. While their goals are similar to ours, it is not yet clear what policies will be used for such important issues as address space management, object location and migration, or the software model of protection. Opal uses a form of password capabilities called *protected pointers* to control access to objects. Protected pointers contain *portal numbers* which are used in cross-domain procedure calls, which serve a similar purpose to our PDX procedures. Cross-domain calls use different parameter passing mechanisms than normal procedure calls in Opal, however, so the different kinds of procedures are not transparent to the user. Opal also allows explicit and implicit validation of access to a memory location. However, two different mechanisms are used: explicit validation is done using protected pointers, while implicit validation uses access control lists. It is not clear why two different validation systems are necessary.

6 Conclusions

A single address space offers many advantages over traditional operating systems. All objects can be accessed in a uniform way, sharing is trivially achieved by passing around addresses, and migration of objects is transparent. As all memory objects are persistent, there is no need for a separate file system with its associated overhead.

A potential problem is security, as any object is always visible to each process, and object access does not require explicit system interaction. We present a novel approach to protection in a single address space based on password capabilities. Our system separates the long term storage of users' capabilities from implicit capability presentation for validation of object accesses. The protection scheme provides a high degree of transparency which allows existing programs to run without modifications. Convenient mechanisms for temporary or permanent modification of a process' protection domain are provided.

The object model allows efficient distribution of system data structures, like the single object directory or the capability store, without requiring special mechanisms for their location and replication. This results in a very simple kernel.

References

[1] D. A. Abramson and J. L. Keedy. Implementing a large virtual memory in a distributed computing system. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, 1985.

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

[3] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 4, pages 108–21, 1991.

[4] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26:360–5, 1983.

[5] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurement of a distributed file system. In *Proceedings of the 13th ACM Symposium on OS Principles*, pages 198–212, 1991.

[6] J. K. Bennett, J. B. Carter, and W. 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.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] R. Bisiani and M. Ravishankar. PLUS: A distributed shared-memory system. In *17th International Symposium on Computer Architectures*, pages 115–124. IEEE, 1990.

[9] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13:65–76, 1987.

[10] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 80–85, Key Biscayne, Florida, 1992. IEEE.

[11] E. Cohen and D. Jefferson. Protection in the HYDRA operating system. In *Proceedings of the 5th ACM Symposium on OS Principles*, volume 5, pages 141–59, 1975.

[12] P. Dasgupta, R. J. LeBlanc, and W. Appelbe. The clouds distributed operating system. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[13] K. Elphinstone. Address space management issues in the Mungi operating system. Report 9312, School of Computer Science and Engineering, University of NSW, Kensington, NSW, Australia, 2033, November 1993.

[14] B. D. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on OS Principles*, pages 211–223, 1989.

[15] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on OS Principles*, pages 202–10, 1989.

[16] F. Henskens. *A Capability-Based Persistent Distributed Shared Virtual Memory*. PhD thesis, University of Newcastle, NSW, Australia, 1991.

[17] F. A. Henskens, J. Rosenberg, and J. L. Keedy. A capability-based distributed shared memory. In *Proceedings of the 14th Australian Computer Science Conference*, pages 29.1–12, 1991.

[18] R. Lea, P. Amaral, and C. Jacquemot. COOL-2: an object oriented support platform built above the Chorus micro-kernel. In L.-F. Cabrera, V. Russo, and M. Shapiro, editors, *Proceedings of the 1st International Workshop on Object Orientation in Operating Systems*, pages 68–72, Palo Alto, USA, 1991. IEEE.

[19] P. Leach, P. Levine, J. Hamilton, and B. Stumpf. The file system of an integrated local network. In *ACM Computer Science Conference*, New Orleans, 1985.

[20] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *International Symposium on Computer Architectures*, pages 148–59. IEEE, 1990.

[21] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7:321–59, 1989.

[22] K. Li and R. Schaefer. A hypercube shared virtual memory system. In *International Conference on Parallel Processing*, pages 125–32. IEEE, 1989.

[23] R. G. Minnich and D. J. Farber. The Mether system: Distributed shared memory for SunOS 4.0. In *Proceedings of the 1989 Summer USENIX Conference*, pages 51–60, 1989.

[24] R. Morrison, A. L. Brown, R. Conner, and A. Dearle. Napier88 reference manual. Persistent Programming Research Report PPRR-77-89, Universities of Glasgow and St. Andrews, 1989.

[25] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–99, 1986.

[26] R. Needham and R. Walker. The Cambridge CAP computer and its protection system. *Proceedings of the 6th ACM Symposium on OS Principles*, pages 1–10, 1977.

[27] R. Needham and R. Walker. The Cambridge CAP computer and its protection system. In *ACM Symposium on OS Principles*, pages 1–10, 1977.

[28] E. I. Organick. *The Multics System: an Examination of its Structure*. MIT Press, 1972.

[29] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on OS Principles*, pages 15–24, 1985.

[30] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: a network transparent, high reliability distributed system. In *Proceedings of the 8th ACM Symposium on OS Principles*, pages 169–77, 1981.

[31] J. Rosenberg and D. 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.

[32] J. Rosenberg and J. L. Keedy. Object management and addressing in the MONADS architecture. In *Proceedings of the 2nd International Workshop on Persistent Object Systems*, Appin, Scotland, 1987. IEEE.

[33] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. SOS: an object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[34] A. S. Tanenbaum and S. Mullender. The design of a capability-based distributed operating system. Technical Report IR-88, Vrije Universiteit, November 1984.

[35] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33:46–63, 1990.

[36] J. Vochteloo, S. Russell, and G. Heiser. Capability-based protection in the Mungi operating system. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*. IEEE, 1993.