User-level Management of Kernel Memory

Andreas Haeberlen[1] and Kevin Elphinstone[2]

[1] University of Karlsruhe, System Architecture Group
76128 Karlsruhe, Germany
haeberlen@ira.uka.de
[2] University of New South Wales
Sydney, 2052, Australia
kevine@cse.unsw.edu.au

Abstract. Kernel memory is a resource that must be managed carefully in order to ensure the efficiency and safety of the system. The use of an inappropriate management policy can weaken the isolation between subsystems, lead to suboptimal performance, and even make the kernel vulnerable to denial-of-service attacks. Yet, many existing kernels use only a single built-in policy, which is always a compromise between performance and generality.
In this paper, we address this problem by exporting control over kernel memory to user-level pagers. Thus, subsystems can implement their own application-specific management policies while independent subsystems can still be isolated from each other.
The pagers have full control over the memory resources they manage; they can even preempt and later restore individual pages of kernel memory. Still, protection is not compromised because the kernel converts its metadata into a safe representation before exporting it. Therefore, pagers need only be trusted by their respective clients.
We describe the model we use to page kernel memory and various techniques for obtaining a safe external representation for kernel metadata. We also report experiences with an experimental kernel that implements our scheme and outline our plans to further develop the approach.

1 Introduction

Operating systems obviously need resource management. Any multitasking or multiuser system needs to ensure resources are efficiently managed to fulfil some desired system-level policy, such as maximising overall throughput or guaranteeing availability to high priority tasks. Poor or simplistic resource management can result in underutilisation, low performance, or even denial of service.

Kernel memory is an often overlooked resource. It is required to implement higher-level resources or services for applications; examples include page tables for implementing virtual memory, buffer caches for file providing, and thread control blocks (TCB) to implement threads. Physical memory is the ultimate resource consumed by kernel memory, and thus simplistic kernel memory management is ultimately simplistic and problematic physical memory management.

As demonstrated by Scout [27], a management approach encompassing all kernel memory is required to avoid denial-of-service attacks.

Several operating systems manage their kernel memory carefully. Scout provides limits on kernel memory per protection domain and per *path* [27]; a path is a logical execution flow through one or more domains. Eros [26] and the Cache kernel [4] both view kernel physical memory as a cache of kernel metadata and as such can evict cache entries when cache capacity is exceeded. However, these systems share one thing in common: they all carefully manage kernel memory in such a way as to fulfil a single overall system policy. This is understandable as each system has a particular focus and is designed to meet its specific needs.

We believe these kernels are overly restrictive in their management of kernel memory and thus limit their application in areas outside their original focus. Related work has shown that applications are often ill-served by the default operating system policy [1, 28] and can benefit significantly from managing their own memory resources [7,9,11,13,15,20]. Ideally, a kernel should be adaptable to different application areas, and even support concurrent applications with differing requirements on kernel memory management whilst preventing interference between the applications.

An example might be a real-time system running together with an insecure best-effort system. The two have very different requirements for kernel memory management: the real-time system may require preallocated and pinned memory to ensure deadlines are met, whereas the best-effort system only needs cache-like memory behaviour to meet its current needs.

Paged virtual memory has become ubiquitous in modern systems as it provides a well understood, flexible, and efficient mechanism to manage the physical memory of applications. Virtual memory has proved sufficient to manage physical memory usage between competing clients, provide recoverable and transactional memory [5,24], provide more predictable or improved cache behaviour via page colouring [14], enable predictable access timing via pinning, and even enable secure application-controlled virtual memory by safely exporting control of basic virtual memory mechanisms [7,17,22]. Given the power, maturity, and understanding of applying virtual memory techniques to user-level applications, we believe virtual memory techniques can also be applied to manage kernel memory.

By paging kernel memory and safely exporting that control to user-level, we believe we can harness the power and flexibility of virtual memory to support classes of applications requiring careful management of kernel memory without targeting and thus restricting our approach to a particular application area; also, we can concurrently support different applications while ensuring isolation from each other.

Moreover, the generality of our approach allows us to unify and replace various existing mechanisms. One example is user-level persistence, which can be easily implemented when kernel metadata is fully accessible. Another example is cache colouring [14], which requires control over the in-kernel placement policy.

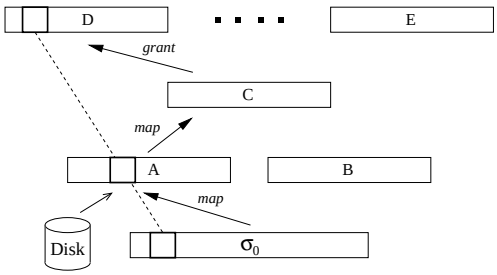

Fig. 1. Virtual memory primitives

2 The Approach

We chose the L4 microkernel as the platform to evaluate our ideas. L4 is a small microkernel which reduces the complexity of the problem. It also has a powerful model for constructing user-level address spaces [16] which we believe can be applied to kernel memory. Our approach to kernel memory management aims to place all kernel memory logically within a kernel virtual address space, which is realised by user-level *kpagers* using the same model that is used to construct user-level virtual address spaces. We believe our approach is unique in that it allows untrusted user-level pagers to safely supply and preempt kernel memory. Before we proceed to describe our approach in more detail, a brief description of the L4 virtual address space model is warranted.

L4 implements a recursive virtual address space model which permits virtual memory management to be performed entirely at user level. Initially, all physical memory is mapped within the root address space σ_0; new address spaces can then be constructed by mapping regions of accessible virtual memory from one address space to the next.

Memory regions can either be `map`-ped or `grant`-ed. Mapping and granting is performed by sending typed objects in IPC messages. In the case of `map`, the sender retains control of the newly derived mapping and can later use another primitive (`unmap`) to revoke the mapping, including any further mappings derived from the new mapping. In the case of `grant`, the region is transferred to the receiver and disappears from the granter's address space (see Figure 1).

Page faults are handled by the kernel transforming them into messages delivered via IPC. Every thread has a pager thread associated with it. The pager is responsible for managing a thread's address space. Whenever a thread takes a page fault, the kernel catches the fault, blocks the thread and synthesizes a page-fault IPC message to the pager on the thread's behalf. The pager can then respond with a mapping and thus unblock the thread.

This model has been successfully used to construct several very different systems as user-level applications, including real-time systems and single-address-space systems [8, 10, 12, 21]. We believe it can also be used to manage kernel memory.

2.1 The Basic Model

We propose the following extension to the L4 memory model to facilitate kernel memory management. While these extensions are L4 specific, they should also be applicable to other systems. We associate each thread with a *kpager* which receives kernel page faults when the kernel requires more memory for a thread. The kpager can choose to `map` any page it possesses to resolve the fault. Like a normal pager, the kpager can revoke the memory at any point by invoking `unmap` on the supplied page.

This basic model is more complicated in reality due to important differences between paging an application and paging the kernel. By paging the kernel to a user-level pager, we are storing critical information in an object backed by an untrusted, insecure pager. To succeed, we need to ensure that no kpager can obtain sensitive kernel information, nor compromise the kernel. However, it is acceptable for a kpager to obtain information associated with its clients, or to compromise its clients.

We consider kernel memory currently in use (the equivalent to memory paged in), and kernel state not in the kernel (the equivalent to memory paged out) separately. In-use memory is protected from kpager interference and examination by revoking user-level read-write access rights to the page. The kpager still logically possesses the page and can `unmap` it from the kernel to gain normal access once again.

Paged-out kernel memory can be freely examined and modified by the kpager. To prevent the disclosure of sensitive information, the kernel transforms the contents of a page into a safe external representation prior to exporting it back to the kpager. To avoid interference by potential kpager modifications to the exported state, the kernel validates the contents when paging it back into the kernel, and converts the contents back into its in-kernel representation. The exact transformation to and from the external representation is dependent on the particular kernel memory being exported and imported. The following section describes the classes of data we deal with.

2.2 Kernel Data Structures

In terms of ease of exporting kernel data to kpagers, we have identified three broad classes of kernel data: *safe*, *redundant*, and *sensitive*. The classes are not necessarily mutually exclusive.

Data is *safe* to export as-is if it is readily available (readable and writable) to the client. If any restrictions are placed on data availability to the client, the data is still *safe* if the restrictions can remain in place after the data is exported to and imported from the kpager.

Some kernel data can be readily reconstructed from data held by user-level applications. This *redundant* data can be exported by simply discarding the contents of the page, and returning a vacant page to the pager. One example is a page table which is discarded when exported, and rebuilt when imported via page faults to user-level pagers.

Data is *sensitive* to export if it refers to kernel internals, to clients other than those being paged by the kpager, or to client attributes not freely accessible to the client itself. Unrestricted access to *sensitive* data could compromise the kernel, detrimentally affect the clients of other kpagers, or raise the privileges of clients beyond what is directly achievable by the client or the kpager. *Sensitive* data can be exported, for example, by sealing it cryptographically before exporting it to user-level, and validating it when it is returned to the kernel.

3 Implementation

This section describes the more interesting details of our implementation. We focus on L4 on Intel's IA-32 architecture, but we believe the techniques described are readily applicable to other architectures. The IA-32 L4 kernel has the following in-kernel data structures: page tables, thread control blocks (TCBs), mappings nodes, and node tables.

3.1 Page Tables

Pages tables are *redundant* data as they are constructed by a user-level application's invocation of the `map`, `grant`, and `unmap` primitives. The user-level pager typically has a superset of the kernel's page table which it uses to manage its clients' virtual address spaces. Page tables are also *sensitive* data as they contain physical addresses. To avoid the potential security issues in exporting *sensitive* data, we actually export the page tables vacant.

3.2 Mapping Nodes

Mapping nodes are used to track the derivation tree of mappings that represent the current state of all address spaces in the system. This mapping database is required to implement the `unmap` primitive. `unmap` removes any mapping derived from a specified mapping and, optionally, the specified mapping itself. Like page tables, the mapping database is *redundant* data constructed by invoking `map`, `grant`, and `unmap`. In principle, the data structure could be exported vacant.

However, the data structure is a hierarchical tree. Thus, to export part of it vacant, any branches derived from the newly vacated part must also be invalidated to ensure `unmap` is correct when applied closer to the root of the tree. Hence, a simplistic approach to vacating pages from the mapping database could result in significant, cascading invalidation. We avoid this by localising mapping nodes prior to exporting them.

Localisation is a general technique we use to transform particular *sensitive* data into *safe* data. Data is exported by translating all data in the page from in-kernel references to references valid in the local user-level context of the client. When returned to the kernel, the page is validated by translating the client-local references back into kernel data. By translating the kernel data into local references, we safely export the data by restricting the contents of the page to

references to objects the client can directly manipulate. Any permutation of the page returned to the kernel could have been constructed directly by the client by invoking operations on local objects.

A mapping node contains a reference to the virtual page and address space it is associated with, a reference to the page and address space from which it is derived, and reference to any further derived mappings. The mapper of the page determines the page from which the mapping is derived, and the receiver of the mapping determines the location where the received mapping is placed.

We split the mapping node into a sender-derived part and a receiver-derived part. The sender's and receiver's kpager pages the respective parts. Each part is localised with respect to the sender and receiver, thus making it examinable by the respective parties. Kpager modifications of the data can only result in situations that could have been created through cooperative application of the mapping primitives by both parties.

3.3 Node Tables

Node tables exist to provide a mapping between virtual memory regions and the corresponding nodes in the mapping database. They are closely related to page tables and have a similar structure; for each page table entry, there is a corresponding node table entry which points to the associated mapping node. Unlike page table entries, however, these pointers are not *redundant*; they are required by the kernel e.g. to locate preempted mapping nodes. Therefore we export them by localising them to the context of the client.

3.4 Thread Control Blocks

Thread control blocks (TCBs) implement kernel threads. A thread's TCB contains a thread's register set and activation stack (if in kernel mode), the thread's state (e.g. waiting or running), its scheduling parameters (time slice, priority, run queue link), and other queue links related to IPC. In order to support lazy thread switching [19], the TCBs are divided into two parts, one of which (the UTCB) is user accessible, and the other (the KTCB) resides in protected kernel memory. The content of the UTCB is modifiable at user level by the thread it implements, and thus is not protected in any way. It is *safe* and we simply export the complete contents to the kpager. Similarly, the thread's user-level register set is also *safe*.

The kernel activation stack and state is *sensitive*. In order to safely export it we use a *continuation*, a special kernel object which contains a digest of the state that is encoded in the stack. Only particular safe points within the kernel need to be represented by the continuations, and those can be safely revalidated when faulted back into the kernel. Examination of the continuation only gives a kpager coarse knowledge of the particular thread's kernel state, and modification of the exported data results in a mutation to some other valid kernel state which can only affect the client thread involved. The integrity of other threads and the kernel itself is preserved.

Scheduling parameters and implementation are *sensitive*. We are currently exploring how to safely export them. If we adopted a hierarchical proportional-share scheduling scheme with kpagers determining scheduling parameters, we could localise the scheduling parameters in terms of shares of the kpager's allocation. However, we are wary of unifying both scheduling and memory management into a single hierarchy. Currently, a copy of the scheduling parameters is kept in the kernel.

3.5 Deadlocks

In a system with pageable kernel metadata, the kernel must be prepared to handle situations where it lacks the metadata necessary to complete an operation. These situations can occur when additional metadata needs to be allocated, or when existing metadata has been paged out. In either case, care must be taken to ensure progress, i.e. to prevent the system from being caught in a deadlock.

To this end, two different problems need to be solved. First, the kernel must not deadlock *internally*, e.g. because the page fault handler itself causes a page fault. Second, the page fault messages must not cause deadlocks in the *user-level system*.

The first problem is common to all pageable kernels; it is essentially a matter of system design. In our system, we solved it by eliminating all circular dependencies between kernel data structures, and by imposing a strict hierarchy. The second problem, however, cannot be solved entirely at kernel level because the user can always create a deadlock, e.g. by establishing a circular dependency between a pager and one of its clients. The kernel can therefore only guarantee that it is *possible* to construct a deadlock-free system with reasonable effort, and that unrelated subsystems are not affected when a deadlock does occur.

In an L4 system, the only critical operation is sending a `map` message via IPC. When a kernel page fault occurs while a kpager is using this operation to resolve another page fault in one of its clients, the kpager is blocked indefinitely because it can never handle the second fault. In this case, however, the kernel can easily detect the deadlock and resolve it by aborting the operation. Both threads are notified and can use this information to avoid further deadlocks, e.g. by handling the page faults in a different order.

We use *fault ordering* to reduce the overhead induced by deadlock resolution. When the kernel detects that it needs multiple resources $r_1, \ldots, r_n$ to complete an operation, it chooses an order $(i_1, \ldots, i_n)$ such that r_{i_j} does not depend on any r_{i_k} with $k > j$. Such an order always exists because the metadata is structured hierarchically. The kernel can then effectively avoid deadlocks by requesting the resources in that order.

3.6 Other Details

We enable accounting and control of kernel memory usage by associating the memory mapped to the kernel with a resource principal. Tasks (i.e. address spaces containing one or more threads) were chosen as resource principals since

most kernel data (page tables, etc.) is used to implement tasks and is shared between all threads in the task. The kernel only uses kernel memory associated with the requestor of a service. Once exhausted, the kernel can fault in more pages on behalf of a task from the task's kpager. Therefore, kpagers can accurately account and control the amount of kernel memory used by individual tasks.

Typically, a pager has a contract to implement virtual memory regions for its clients. For this purpose, it uses the mapping primitives and its physical memory resources; it also keeps a mapping between virtual page addresses and their contents, which reside either in memory or on external storage. However, while the client of a normal pager does not know the current assignment between physical pages and virtual memory regions and therefore must treat the region as an opaque object, the client of a kpager (the kernel) has full knowledge and can therefore operate on the memory as it sees fit, even access the physical frames directly. Thus, virtual page addresses become content identifiers and need not bear any resemblance to the actual virtual addresses used by the kernel. Kernel page faults can be signalled when content is not present or when more memory is required, not necessarily as a result of hardware-based page faults. This gives the kernel implementor full freedom, but still preserves the simple pager model for all user-level code.

Different kernel pages have different costs associated with revoking them from the kernel; for example, a root page directory is more costly to revoke than a leaf directory. To allow fine tuning of kpager policy, we are exploring the possibility of giving specialised kpagers information about the internal data types of a particular kernel. This can be done cleanly by assigning kernel data types to specific virtual page ranges. A specialised kpager can make use of this information, e.g. to adjust its replacement policy or to discard vacated pages instead of writing them to backing store. At the same time, a generic kpager can function correctly, albeit sub-optimally.

4 Evaluation

We have constructed an experimental L4 kernel to serve as a platform to develop and experiment with our ideas. It implements a modified L4 API and allows kpagers to page most dynamically allocated kernel memory. All memory-management related data is paged, and most TCB data is paged (all but approximately 100 bytes of an original 1 Kbyte TCB).

The kernel is stable and complete enough to run L^{4}Linux [10], a derivative of Linux 2.4.20 that was modified to run on top of the L4 microkernel. We used this system to get a first impression of performance.

4.1 Kernel Memory Usage

In order to determine the amount of kernel memory used by typical applications, we booted a standard Debian distribution on top of L^{4}Linux. After opening an

Space	Application	Threads	Resident	#P	#N	#M	#U	Metadata
30.1	σ_0	1	131.080k	3	1	8	1	52k
32.1	L^{4}Linux	19	129.804k	5	5	8	3	84k
214.2	pingpong	2	20k	4	4	1	1	40k
216.2	init	2	76k	5	5	1	1	48k
218.2	bash	2	52k	5	5	2	1	52k
21a.2	bash	2	392k	5	5	2	1	52k
21c.2	getty	2	80k	5	5	1	1	48k
21e.2	syslogd	2	152k	5	5	1	1	48k
220.2	portmap	2	96k	5	5	1	1	48k
222.2	klogd	2	108k	5	5	1	1	48k
224.2	rpc.statd	2	108k	5	5	1	1	48k
226.2	gpm	2	96k	5	5	1	1	48k
228.2	inetd	2	100k	5	5	1	1	48k
22a.2	lpd	2	112k	5	5	1	1	48k
22c.2	smbd	2	260k	5	5	1	1	48k
22e.2	rpc.nfsd	2	272k	5	5	1	1	48k
230.2	rpc.mountd	2	284k	5	5	1	1	48k
232.2	cron	2	140k	5	5	1	1	48k
234.2	getty	2	80k	5	5	1	1	48k
236.2	getty	2	80k	5	5	1	1	48k
238.2	getty	2	80k	5	5	1	1	48k
23a.2	cc	2	164k	5	5	1	1	48k
23e.2	emacs	2	2.700k	5	5	4	1	60k

Fig. 2. Memory usage under L^{4}Linux. Table shows resident set size, number of pages used for page tables (P), node tables (N), mapping database (M), user TCBs (U), and total kernel memory usage.

`emacs` session and starting a compile job, we obtained a snapshot of the system and analysed the usage of kernel memory (Figure 2).

We found that a typical[3] application consumes approximately 100-300kB of user memory and 40-60kB of kernel memory. We conclude that a nonnegligible portion of main memory is used as kernel memory; hence, some extra effort for managing it seems justified.

We also found that the numbers are surprisingly high and do not vary much between small and large applications. This is due to high internal fragmentation, which is largely caused by sparsely populated page tables and cannot be avoided by the kernel alone since the page table format is dictated by the IA-32 hardware. However, by replacing the standard Linux address space layout with a more compact one, the overhead could be reduced significantly, in some cases by up to 50%.

[3] The root pager σ_0 and the L^{4}Linux server have atypical resident set sizes because they have all physical memory (128MB in our experiment) mapped to their address spaces. Most of that memory is mapped on to other applications.

A comparison to other L4 kernels for the IA-32 shows that the effective overhead of our scheme amounts to only 1.5 frames or 6kB, which we consider sufficiently low.

4.2 Policy Overhead

In order to determine the temporal overhead for a simple user-level allocation policy, we measured the time required to handle a kernel page fault. To this end, we modified our kernel to support an optional in-kernel memory pool. When this pool is in use, no kernel page faults are generated.

We then ran a simple test application that causes a page fault in a previously untouched memory region. This memory region was carefully chosen so that multiple instances of kernel metadata (a page table and a node table) would be required to handle the fault. Without the in-kernel memory pool, the kernel would thus have to send two additional page faults.

In-kernel allocator	1 fault	18,091 cycles (± 100)
User-level allocator	3 faults	21,454 cycles (± 100)

Fig. 3. Cycles required to handle a complex page fault, for which the kernel must allocate two additional pages of kernel memory.

The experiment was performed on a dual Pentium II/400 system with 192 MB of main memory; we used the performance counters of the CPU to measure the cycles required in both cases (Figure 3). The difference of approximately 3,400 cycles is explained by the additional overhead for generating two fault IPCs, executing the user-level fault handler twice, and crossing the user-kernel boundary four times. This indicates an effective overhead of 1,700 cycles per kernel page fault on this machine.

In the previous section, we demonstrated that a typical L^{4}Linux task uses less than 60kB of kernel metadata. This is equivalent to 15 frames. We estimate that requesting these frames from a simple user-level manager, e.g. one that implements a Quota policy, causes an additional one-time overhead of $15 \cdot 1,700 = 25,500$ cycles or $64\mu s$, which we consider acceptable, especially given that our microkernel is still completely unoptimized.

5 Related Work

There has been previous work on managing kernel memory from user level. The path abstraction in Scout [27], Resource Containers [2] and Virtual Services [23] can be used to account for and limit consumption of kernel memory; all of them can be controlled from user level. The same is possible in extensible systems like SPIN [3] and VINO [6], where code can be uploaded into the kernel at runtime. However, all of these approaches use a global policy for the entire system, and

neither of them supports preemption or revocation of kernel memory, except by killing the principal.

EROS [25] and the Cache Kernel [4] use a different approach in which the kernel acts as a cache for metadata and can evict objects from this cache when its capacity is exceeded. However, neither the capacity nor the allocation of these caches can be controlled by applications, and it is difficult to isolate subsystems from each other.

Liedtke has described another approach where applications can resolve a shortage of kernel memory by donating some of their own memory to the kernel [18]. However, the model is incomplete as no mechanism is provided to revoke or reclaim memory from the kernel.

The Fluke kernel can export kernel state to user level, which has been used to implement user-level checkpointing [29]. However, kernel memory itself cannot be managed.

6 Conclusions and Future Work

In this paper, we have presented a mechanism that can be used to safely export control over kernel memory to user level. Unlike previous solutions, it supports graceful preemption and revocation of kernel memory, which makes it possible to implement not only basic policies like FCFS or quotas, but also more advanced strategies such as Working Set. Also, every subsystem can implement its own custom policy, allowing it to benefit from specific knowledge about its current and future needs.

To demonstrate the feasibility of our approach, we have implemented it in an experimental kernel that supports the L4 API. The experimental kernel allows all memory-related metadata and most TCB metadata to be paged from user level. A small portion of the TCB (approximately 10%) is not paged because this would require changes to the L4 API. We plan to continue refining our design to eliminate the remaining unexported data; also, we will conduct further experiments to apply different management policies and to evaluate their performance.

We believe that our mechanism is powerful enough to be used beyond the simple control of physical memory consumption. We envisage kpagers enabling subsystem checkpointing by capturing both the kernel and user-level state of a subsystem. Kpagers should also enable paging of kernel data to backing store, thus allowing kernel memory to exceed physical memory limitations. Page coloring [14] might also be advantageous when applied to kernel memory.

In summary, we believe we can safely export management of kernel memory to user-level pagers. Our system should be flexible enough to do any or all schemes concurrently on isolated subsystems, without requiring kernel modification.

References

1. Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proc. 4th ASPLOS*, pages 96–107. ACM Press, Apr 1991.
2. Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proc. 3rd OSDI*, pages 45–58, Feb 1999.
3. Brian N. Bershad *et al.* SPIN: an extensible microkernel for application-specific operating system services. In *Proc. 6th ACM SIGOPS European Workshop*, pages 68–71, 1994.
4. David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proc. 1st OSDI*, pages 179–193, Nov 1994.
5. David R. Cheriton and Kenneth J. Duda. Logged virtual memory. In *Proc. 15th ACM SOSP*, pages 26–38, Dec 1995.
6. Yasuhiro Endo, James Gwertzman, Margo Seltzer, Christopher Small, Keith A. Smith, and Diane Tang. VINO: The 1994 fall harvest. Technical Report TR-34-94, Harvard Computer Center for Research in Computing Technology, 1994.
7. Dawson R. Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM: Application-level virtual memory. In *Proc. 5th HotOS*, pages 72–77, May 1995.
8. A. Gefflaut *et al.* The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.
9. Steven M. Hand. Self-paging in the Nemesis operating system. In *Proc. 3rd OSDI*, pages 73–86. USENIX Association, Feb 1999.
10. Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proc. 16th ACM SOSP*. ACM, 1997.
11. Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proc. 5th ASPLOS*, pages 187–197, Oct 1992.
12. Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9), Jul 1998.
13. John P. Kearns and Samuel DeFazio. Diversity in database reference behaviour. *Performance Evaluation Review*, 17(1):11–19, May 1989.
14. R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM TOCS*, 10(4):338–359, Nov 1992.
15. Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 48–64. ACM Press, 1993.
16. Jochen Liedtke. On μ-kernel construction. In *Proc. 15th ACM SOSP*, pages 237–250. ACM Press, Dec 1995.
17. Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9), Sep 1996.
18. Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proc. 6th HotOS*, May 1997.
19. Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proc. 8th HotOS*, pages 15–18, May 2001.
20. Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. Technical Report

TR-90-09-05, Department of Computer Science and Engineering, University of Washington, 1990.
21. Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proc. 23rd Real-Time Systems Symposium*, Dec 2002.
22. Richard Rashid *et al.* Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8), Aug 1988.
23. John Reumann, Ashish Mehra, Kang G. Shin, and Dilip Kandlur. Virtual services: A new abstraction for server consolidation. In *Proc. of the 2000 USENIX ATC*, Jun 2000.
24. M. Satyanarayanan, Harry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. *ACM TOCS*, 12(1), Feb 1994.
25. Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. State caching in the EROS kernel. In *Proc. 7th Intl. Workshop on Persistent Object Systems*, pages 88–100, 1996.
26. Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proc. 17th ACM SOSP*, pages 170–185, Dec 1999.
27. Oliver Spatscheck and Larry L. Peterson. Defending against denial of service attacks in Scout. In *Proc. 3rd OSDI*, pages 59–72, Feb 1999.
28. Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, Jul 1981.
29. Patrick Tullmann, Jay Lepreau, Bryan Ford, and Mike Hibler. User-level checkpointing through exportable kernel state. In *Proc. 5th Intl. Workshop on Object Orientation in Operating Systems*, Seattle, WA, Oct 1996.