Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Managing Kernel Memory Resources from User Level

Andreas Haeberlen

Diplomarbeit

Verantwortlicher Betreuer:	Prof. Dr. Alfred Schmitt
Betreuende Mitarbeiter:	Dr. Kevin Elphinstone
	Dipl.-Inf. Volkmar Uhlig

25. April 2003

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 25. April 2003

Andreas Haeberlen

Abstract

In order to implement abstractions like threads or address spaces, operating system kernels need to maintain the corresponding metadata. This metadata is usually stored in kernel memory, i.e. in a region of physical memory that is reserved for kernel use. As the amount of available kernel memory is limited, its allocation must be controlled carefully; otherwise, applications can run a denial-of-service attack against the kernel by consuming all of its resources.

Some operating system kernels have addressed this problem by providing powerful management policies. However, with a single global policy, it is difficult to accommodate multiple domains with different requirements at the same time. Also, existing solutions allow only limited control over kernel memory; for example, it is often not possible to reduce an allocation, except by killing the task that currently holds it. This makes it difficult to respond to changing load situations or suspected denial-of-service attacks.

In this thesis, we present a new scheme which uses paged virtual memory to control kernel memory resources from user level. Memory can be preempted from the kernel and restored later; any kernel metadata affected by this is converted into an external representation, which can be safely exported. To demonstrate our approach, we apply it to an existing kernel, the L4 microkernel. We also present an experimental implementation.

Acknowledgements

This work would not have been possible without the continuous support of my advisors, Dr. Kevin Elphinstone and Volkmar Uhlig. I am grateful for their insights, expert guidance, and encouragement.

I would like to thank the members of the System Architecture groups at Karlsruhe, Dresden and Sydney, for their valuable feedback and many interesting discussions. There are too many names to mention them all, but I would like to thank Marcus Völp and Adam Wiggins for patiently discussing many crazy ideas, and Gerd Liefländer for his encouragement and constant support throughout this thesis. Special thanks go to Uwe Dannowski for detailed proofreading and moral support – and, of course, for sharing his endless supply of chocolate.

Contents

Chapter 1

Introduction

Resource management is one of the core responsibilities of any operating system. Directly after system startup, the operating system has full control over all the resources offered by the hardware, such as processing power, storage space, energy, or bandwidth. It must then distribute these resources to applications in an appropriate fashion, i.e. in a way that maximizes user utility.

Today, most operating systems support protection, i.e. they can prevent applications from accessing the resources of another application unless they have been explicitly authorized to do so. This is obviously beneficial for a multi-user system where users may want to hide sensitive information from other users, but it can also be used to enhance the robustness of the system by isolating subsystems that are untrusted or potentially faulty.

The restrictions required to implement protection are usually enforced by hardware that can distinguish multiple privilege levels. Only one part of the operating system – the *kernel* – is allowed to run at the highest privilege level; it uses the resources to implement a number of *core abstractions*, such as threads or address spaces, which it provides to applications.

Most of these core abstractions are associated with metadata. For example, each thread requires a thread control block to store its context and its state, and each address space needs a page table, which contains the translation from addresses to memory resources. If applications were allowed to access this metadata directly, they could easily change the resource allocation and thus circumvent protection. Therefore, the kernel must protect the metadata as well.

However, in order to store the metadata, the kernel must use some memory resources for itself. This *kernel memory* is a resource in its own right and must be managed carefully, like any other resource. In particular, the amount of available kernel memory is limited, e.g. by the amount of physical memory in the system. If it becomes exhausted, the kernel must deny further requests, which renders the system effectively unusable.

Therefore, the use of a simplistic allocation policy like FCFS leads to many potential problems, such as:

- **Denial of service:** An attacker can run a denial-of-service attack against the kernel by executing an application that consumes all available kernel memory. This can be done in many ways, e.g. by creating a large number of threads, which usually does not require any special privileges.
- **No isolation:** The kernel memory consumption of one subsystem affects all other subsystems. The resulting cross talk makes it impossible to give quality-of-service guarantees for a particular subsystem.
- **No predictability:** Under high load, kernel memory may become exhausted. From the application's point of view, this might happen at any time, and therefore any system call can potentially fail. This makes it impossible to offer guaranteed response time, e.g. for a real-time client.

Several operating systems have avoided these problems by enhancing the kernel with a powerful policy for metadata management. In the Scout system, it is possible to limit the amount of kernel resources used by a particular I/O path [40], which has been demonstrated to be an effective defense against denial-of-service attacks [49]. The Resource Container abstraction provides systems with a method to account kernel resources towards individual activities, and to impose limits on their resource usage [3].

Other operating systems have provided means to dynamically customize or replace the management policy. SPIN uses a two-level allocation scheme in which a system-wide allocator distributes resources to multiple user allocators; code for the user allocators can be uploaded to the kernel at runtime [4]. The Cache Kernel does not manage its metadata itself; instead, it acts as a cache for system objects that are provided by user-mode application kernels [9]. The kernel can evict objects from this cache when it needs more memory.

1.1 Problem statement

All of these approaches have in common that the ultimate policy used to manage kernel memory is global to the entire system. We believe this is overly restrictive. Such a policy is always designed with a particular focus and therefore works well for some applications, but not so well for others; therefore, it is unsuitable for a general kernel which must be able to accommodate subsystems with different requirements *at the same time*. For example, users may want to run real-time tasks concurrently with best-effort applications, and service providers may need to provide different QoS levels to their customers. This is difficult to achieve with a single global policy.

Furthermore, most of the existing solutions provide only very limited control over kernel memory resources. For example, it is usually not possible to reduce an allocation once it has been granted, except by killing the current owner. This makes it difficult to respond to changing load situations or suspected denial-of-service attacks because killing tasks is usually not an option.

Finally, related work has shown that applications can benefit significantly from managing their memory resources according to their own policy [15, 19, 20]. This is possible because the application has specific knowledge and can therefore give far more accurate predictions on future resource usage than the kernel. Although this argument has mostly been used for ordinary memory resources, it can be applied to kernel memory as well.

1.2 Approach

In this paper, we present a new scheme for kernel memory management which is based on paged virtual memory. Virtual memory has become ubiquitous in modern systems as it provides a well understood, flexible, and efficient mechanism to manage physical memory resources. It has proven sufficient to control the memory usage of competing clients, provide recoverable and transactional memory [10, 45], provide more predictable or improved cache behavior via page coloring [28], enable predictable access timing via pinning, and even enable secure application-controlled virtual memory by safely exporting control of basic virtual memory mechanisms [15, 35, 42]. We show that this powerful mechanism can also be used for kernel memory, and we demonstrate that similar benefits can be obtained.

Clearly, if the kernel is to be paged by user-level applications, care must be taken not to compromise protection, e.g. by allowing applications to see – or even modify – internal metadata of the kernel. In our system, this is achieved by converting the metadata into an *external representation* before exporting it to user level. It has already been shown that this can be done safely, i.e. without enabling applications to increase their privileges [53]. When the external representation is returned to the kernel, the original metadata can be fully reconstructed.

We demonstrate the feasibility of our scheme by applying it to an existing kernel, the L4 microkernel. Previous L4 implementations have mostly used fixed-size kernel memory pools with FCFS allocation. By making some minor changes to the API and the internal structure of the kernel, we arrive at a kernel that is fully pageable, i.e. all of its internal memory resources can be managed from user level. We also present an experimental implementation, L4/Strawberry, which we use to analyze the performance impact of this scheme.

The rest of this thesis is structured as follows: In Section 2, we summarize related work, and Section 3 describes the approach in more detail. In the following three sections, we discuss how we applied our scheme to L4. Section 4 contains a detailed description of the changes we had to make to the original L4 model, Section 5 analyzes the consequences, e.g. for performance and protection, and Section 6 presents and evaluates our experimental kernel, L4/Strawberry. Finally, in Section 7, we summarize and present our conclusions.

Chapter 2

Related Work

In this chapter, we discuss previous work on kernel metadata management. We begin by describing various approaches to controlling the allocation of kernel memory, then we present an overview of work that has been done to safely export kernel metadata to user level.

2.1 Kernel memory management

Similar to a user-level application, the kernel manages its memory resources according to a certain set of policies. For example, an allocation policy is used to decide whether or not a specific request for kernel memory should be granted, and a placement policy is applied to choose one out of multiple free memory regions. Most of the differences between existing approaches are related to certain aspects of these policies, i.e. where they are implemented, how powerful they are, and how much influence applications can exert on them. The following four basic approaches exist:

- **Static in-kernel policy:** A fixed management policy is built into the kernel at compile time. The policy can be customized at runtime, e.g. by changing certain parameters, but it cannot be replaced with a fundamentally different policy.

- **Extensible kernel:** The kernel allows new functionality to be added at runtime, e.g. by uploading small pieces of code. Thus, the management policy can be changed or replaced entirely.

- **State caching:** Kernel objects are maintained by user-level application kernels, who also control the corresponding management policies. The system kernel reads these objects into an internal cache before it uses them; when the objects are evicted from this cache, they are written back to user level.

- **User-level managers:** The kernel only provides a mechanism for memory management; the actual policy is implemented by user-level applications.

In the following sections, we discuss each of these approaches in more detail, and we describe and cite related work.

2.1.1 Static in-kernel policy

The L4 microkernel [34] allocates metadata from an in-kernel memory pool, which is created during startup and has a fixed size. As long as there is enough free space available, new requests are always granted. When the pool is exhausted, no new metadata can be allocated, and the respective system call fails with an error code. In this case, the requestor can try to free up some memory, e.g. by deleting threads or address spaces.

The Linux 2.4 memory manager [44] dynamically assigns physical memory frames to one of multiple pools or *caches*. These caches include the buffer cache, the inode cache, a cache for process mapped virtual memory and a slab cache for kernel metadata. Most of the caches can grow and shrink on demand; thus, the kernel can free up additional memory for kernel metadata by reducing e.g. the size of the buffer cache. However, the kernel can still experience memory pressure when all physical memory is in use. In this case, an emergency recovery function (the Out-of-Memory killer) is invoked, which frees up memory by randomly terminating tasks.

The K42 kernel [26] segregates kernel memory into pinned memory and paged memory. Kernel pinned memory contains all code and data necessary to do paging I/O, while the rest is subjected to the normal paging scheme and can be paged out to backing store. Thus, the size of the metadata used by the kernel can exceed the amount of physical memory.

Several other approaches introduce a new abstraction for resource principals. The Scout operating system [40, 49] supports a special *path* abstraction that represents a stream of data flowing through several subsystems, e.g. packets of a certain TCP connection. Resource Containers [3] can be used to account resource consumption towards individual activities; for example, a single thread can serve requests with different priorities by dynamically binding to the respective containers. The Solaris Resource Manager [51] introduces limit nodes, or *lnodes*, which support user- or group-based resource control. Virtual Services [43] support fine-grain accounting in the presence of shared services; the mechanism intercepts system calls and uses a classification mechanism to infer the current principal. All of these approaches support resource limits per principal. If the limit is exceeded, the system can take action; for example, the principal may be notified, blocked, or transferred to a best-effort service class.

Analysis

All of the above approaches rely on a single, system-global allocation policy. We consider this overly restrictive because this policy is always a compromise between performance and generality; related work has shown that applications are often ill-served by the default operating system policy [1, 50] and can benefit significantly from managing their own memory resources [15, 19, 20, 27, 29, 39]. Also, while the allocation policy can be configured in some of the approaches, other policies, e.g. for placement or replacement, cannot be influenced at all.

Furthermore, the effectivity of this approach depends strongly on the policy that is being used. FCFS, which is implemented in L4 and Linux, cannot prevent applications from monopolizing kernel memory or running a Denial-of-Service attack against the kernel. Most of the other schemes use some variant of the Quota policy, which can either provide predictability (with reservations) or good utilization (with overcommitment), but not both.

Finally, previous work has consistently failed to address the issue of preemption and revocation. In most cases, once an allocation of kernel memory is granted, it can only be freed indirectly by the corresponding principal if it chooses to release the metadata object that is stored in it; sometimes killing the principal is offered as an alternative. However, this is unacceptable in most cases.

2.1.2 Extensible kernels

The SPIN operating system [4, 5] allows applications to customize the services offered by the kernel. This is done by installing a so-called *spindle*, which contains application-specific code that is written in Modula-3, a type-safe language. Although spindles are executed with kernel privileges, they cannot interfere with the rest of the system; the kernel relies on a combination of compile-time and run-time checks to prevent unauthorized access. New functionality can be based on *core services*, which are part of a framework for managing memory and processor resources that is implemented by the kernel itself.

The VINO kernel [14] uses a similar approach. Applications may customize the implementation of kernel resources by *grafting* an extension into the kernel. However, modules need not be written in a type-safe language; instead, they must be created with a trusted compiler that inserts base-and-bounds checks or sandboxing instructions and then marks the resulting code with a digital signature.

Another approach, which is widely used in current operating systems, is to insert kernel extensions directly and *without* any additional checks. In this case, installing new extensions is a privileged operation that can only be performed by the supervisor or by users with similar authority.

Analysis

For our discussion, we distinguish two different types of kernel extensions: *Untrusted* extensions are executed in kernel mode, but not with full privileges; they are subjected to an in-kernel protection mechanism which prevents them from invoking certain operations. *Trusted* extensions are not restricted in any way.

Untrusted extensions solve only part of the problem. Since they do not have full privileges, they can only modify or extend, but not entirely replace kernel functionality. For example, untrusted extensions are usually barred from accessing the MMU, so they cannot replace the virtual memory subsystem; instead, they must rely on some kind of basic service provided by the original kernel. For security reasons, memory used by this basic service cannot be managed by an untrusted extension.

Trusted extensions can only be installed by trusted applications; thus, untrusted applications cannot benefit from a custom policy.

2.1.3 State caching

The V++ Cache Kernel [9] does not fully implement all the functionality associated with its core abstractions, namely threads and address spaces; it merely acts as a cache for these objects. The management functions are provided by user-level *application kernels*, who are also responsible for loading the objects into the cache when they are needed. When the cache is full, the kernel evicts objects from its cache and writes them back to user level. In this system, user data and kernel metadata need not be strictly separated; they are both stored in ordinary memory and can be maintained entirely at user level.

The EROS kernel [46, 47] implements an abstract virtual machine that is based on type-safe capabilities. Classical kernel metadata, such as process and address space descriptors, is constructed from capabilities and maintained at user level. In order to avoid parsing and validating these data structures on every access, the kernel keeps several caches and auxiliary data structures, e.g. a context cache for recently used process contexts. When necessary, information is evicted from the caches and written back to the original data structures.

Analysis

In this approach, kernel metadata is maintained at user level and stored in ordinary memory, where it can be subjected to standard memory management policies. Hence, no separate policy is required to manage kernel metadata.

However, applications must compete for space in the kernel cache instead, which is also a limited resource. A malicious task can easily flood this cache and thus degrade the performance of other tasks. Therefore, the caching approach by itself is not sufficient to isolate subsystems or to prevent Denial-of-Service attacks; it must be combined with a management policy for the kernel cache. It is likely that applications can benefit from customizing this policy as well, e.g. by evicting less important objects first. The resulting problem is similar to the one discussed in this thesis.

2.1.4 User-level managers

Liedtke et al. [36] proposed an extension to the memory pool model that was used in the original L4 microkernel [34]. In their approach, the kernel still allocates metadata from an in-kernel memory pool and fails system calls when this pool is exhausted. However, applications can resolve this situation by *donating* some of their own memory to the kernel. User-level memory managers must ensure that the memory being donated is removed from all user address spaces; afterwards, the kernel can use the additional memory to allocate metadata for the donator.

The Calypso translation layer [52], a virtual memory subsystem for the L4 microkernel, implements a similar mechanism that differs from the donation model in two important points: It uses per-task memory pools instead of a single central pool, and it does not fail system calls when additional memory is needed. Instead, the requestor is suspended and a page fault is sent to its pager. The pager can then resolve the fault by allocating additional memory.

In both approaches, the kernel ensures that the memory it uses for metadata is not mapped to any other address space. This effectively prevents user access to kernel metadata, but complicates the process of reclaiming kernel memory that is no longer used. In an earlier approach proposed by the author [18], this problem is solved by allowing the pagers to retain control over the memory they allocate to the kernel. However, the kernel must still protect and lock pages that contain live metadata. Since this would allow malicious applications to hijack memory from their pagers, a special permission bit is introduced for memory that may be given to the kernel.

Analysis

By exporting an interface for controlling kernel memory allocation, the kernel enables applications to use their own management policies; also, different policies can co-exist in the same system.

However, the above mechanisms are unnecessarily restrictive because they do not allow managers to decrease or revoke an allocation of kernel memory, except by killing the principal. Thus, only very simple policies like FCFS and Quota can be implemented, and it is difficult for the system to respond to sudden load changes or suspected Denial-of-Service attacks.

2.2 Exporting kernel state to user level

Several existing kernels are able to export their internal state to user level. However, this feature has been used mainly to implement process migration and orthogonally persistent operating systems, and not to make kernel memory preemptible.

The Fluke microkernel [16, 53] makes every exported operation fully interruptible and restartable. Thus, the complete state of a thread can always be exposed to user level, even when it is involved in a system call. All kernel state is kept in user-visible kernel objects, so-called *flobs*, and can be cleanly exported and imported at any time. This is exploited by a user-level checkpointer, which periodically *pickles* the kernel state, i.e. converts it into an externalized form, and writes it to stable storage.

L3 [33] and its successor, L4 [7, 48], export kernel metadata to a single trusted entity, the *checkpoint server*. Although this server runs with user privileges, it logically owns part of kernel memory and can access this part directly using mappings in its address space. The checkpoint server is also allowed to revoke certain privileges from the kernel, e.g. to implement copy-on-write for kernel metadata.

The Charm microkernel [12], which is derived from the Grasshopper system [11], exposes its metadata to user level in order to support applications in implementing their own persistence policy. However, applications are not allowed to create or modify metadata; security-related metadata is not exposed.

2.3 Summary

In short, previous approaches to kernel memory management suffer from at least one of the following deficiencies:

- They do not allow untrusted applications to benefit from a custom management policy,
- They cannot be used to enforce subsystem isolation, or to prevent Denial-of-Service attacks,
- They are only applicable to part of the variable-size kernel state, but not to *all* of it, or
- They support only a very limited range of management policies

Exporting kernel state to user level is not a new feature, but to our knowledge, it has never been used to page kernel memory.

Chapter 3

Paging the Kernel

In this section, we first discuss requirements for kernel memory management at a general level, and we define a set of properties we consider necessary for any such management scheme. Then we present our proposed solution, which is an application of paged virtual memory to kernel metadata. Finally, we show that this solution meets all of our requirements.

3.1 Requirements

We begin by defining a set of generic requirements that should apply to any resource management scheme:

1. **Limits:** Managers should be able to limit the amount of resources used by individual principals. The principals should not be allowed to exceed this limit without agreement of the manager.

2. **No reallocation:** Resources that have been allocated to a particular principal should always be available to it; they should not be transparently reallocated to other principals.

3. **Revocation:** A manager that has allocated some or all of its resources to a principal should be able to revoke this allocation at any time.

The above is sufficient to manage ordinary resources; however, kernel memory is special because it is used to store metadata for *all* kernel-level resources, *including itself*. If the metadata for kernel memory were excluded from the scheme, the question would immediately arise how this meta-metadata should be managed, i.e. the problem would reappear on another level. Therefore, we add the following requirement:

4. **Full scope:** It should be possible to apply the mechanism to *all* variable-size kernel metadata

This excludes kernel code and fixed-size metadata such as global variables, which can be preallocated during startup.

Kernel memory is also special in the sense that it is not explicitly requested by a principal; instead, it is implicitly allocated when an application uses a core abstraction. Therefore, an appropriate principal must be identified *a priori*. Although other choices are possible, we decide for the principal that uses the core abstraction, which leads to the following requirement:

5. **Accounting:** When a resource is used to provide a service for a particular principal, it should be possible to account the resource towards that principal.

Metadata that is stored in kernel memory is directly relevant for protection. Hence, care must be taken not to give too much authority to managers; otherwise, they would have to be part of the trusted computing base, and an untrusted subsystem could not easily benefit from a domain-specific policy. Therefore, we add another requirement:

6. **No trust:** Resource managers should not have to be trusted. They must not have power to circumvent protection, nor should they be able to affect principals other than their own clients.

Finally, we add some desirable features that are not strictly necessary, but are required to implement many powerful policies:

7. **Preemption:** It should be possible to revoke a resource temporarily and save its state in order to restore it later.

8. **Notification:** Managers should be notified when one of their clients needs additional resources.

9. **Virtualization:** It should be possible for managers to hand out virtual resources, and to back them with physical resources when necessary.

10. **Delegation:** Managers should be allowed to delegate control over some or all of their resources to other managers

3.2 Proposed scheme

Our approach to kernel metadata management is to construct the kernel virtual address space using the same model that is used to construct user-level virtual address spaces, i.e. with paged virtual memory. The proposed model consists of four basic parts: A scheme for assigning virtual addresses to kernel objects, a notification mechanism, means for allocating and revoking kernel memory, and an external representation for exporting kernel metadata safely to user level. We now describe each of those parts in more detail.

3.2.1 Resource space

Kernel objects are placed in *resource space*, a special memory region[1] which is part of every virtual address space. When the kernel interacts with user-level resource managers, it uses virtual addresses in resource space to request memory for a specific kernel object, or to reclaim an object that has been preempted. User-level managers can back part of resource space with their physical memory resources.

Because virtual addresses are used to identify kernel objects, they must be unique throughout the lifetime of the respective object, i.e. objects must not be relocated. Also, user-level managers must not be able to deduce any protected information merely from the location of a kernel object.

3.2.2 Notification

When the kernel detects that a principal lacks the necessary metadata to complete an operation, it determines the resource manager which is responsible for that principal and notifies this manager. The notification contains at least an identifier for the principal and the virtual address of the missing kernel object in resource space. The faulting operation is suspended until the resource is provided by the manager, i.e. the corresponding region in resource space is backed with physical memory.

This operation corresponds to a page fault in the classical paging scheme; in fact, if page faults are exported to user level, they can be used directly. However, a page fault usually blocks a single thread, whereas in this case, the faulting operation can involve multiple threads which must all be blocked.

[1]It is not strictly necessary to separate resource space from user space; they can overlap or even be identical. However, conflicts between user and kernel objects may then be hard to avoid.

3.2.3 Allocation and deallocation

Resource managers can allocate some or all of their memory resources to back the resource spaces of their clients, and they can revoke these allocations at any time. Every allocation is bound to a principal and a specific region in resource space; the kernel guarantees that the resource will only be used for metadata in that region, and only for that specific principal.

This corresponds to the *map* and *unmap* primitives typically found in user-level memory management schemes. Of course, the kernel must ensure that live metadata cannot be accessed from user level, e.g. by temporarily revoking access to the corresponding resources.

3.2.4 External representation

When a memory resource containing live metadata is preempted, the metadata m is converted into an external representation $e = f(m)$. The conversion function f must have the following properties:

1. **Reversible:** The inverse function f^{-1} must exist, and $f^{-1}(f(m)) = m$ must hold for all valid metadata instances m.

2. **Safe:** Given metadata $e = f(m)$ in external form, it must be impossible or at least computationally hard to find an e' such that $m' = f^{-1}(e')$ is a valid metadata instance that conveys more privileges than m.

3. **Monotonous:** The representation of $f(m)$ must not be larger than the representation of m itself.

This operation does not have an equivalent in the classical paging scheme. The reason is that by modifying memory contents, a pager can only affect its own clients, whereas a manager for kernel memory could potentially affect the entire system if it was given access to kernel metadata in its original form.

3.3 Analysis

By definition, our scheme fulfills requirements 2 (no reallocation), 3 (revocation), 7 (preemption) and 8 (notification). Also, because the resource manager is notified whenever additional resources are needed, it can easily enforce limits and account for resource consumption, which corresponds to requirements 1 (limits) and 5 (accounting). Furthermore, it is free to use different physical resources over time to back the same kernel object, as long as it copies the external representation of the contents every time; hence, requirement 9 (virtualization) is also met.

Assuming that the external representation is safe, the resource manager cannot use it to compromise protection; anything it could do, i.e. delete, modify or forge metadata, can only affect its own clients, who have to have implicit trust in it anyway. Therefore, it is safe to have subsystems managed by untrusted managers because the rest of the system cannot be compromised, and requirement 6 (isolation) is satisfied.

Requirement 10 (delegation) is not explicitly addressed; however, delegation of memory resources is a well-understood problem, and we do not see any specific issues that would prevent the application of an existing scheme. As the type of the metadata was not restricted, requirement 4 (full scope) is satisfied as well.

3.4 Techniques for exporting metadata

Obviously, the main challenge in implementing the above scheme is to find an appropriate external representation for all kernel data structures. To see that this is possible in principle, consider a simple cryptographic scheme where the metadata is encrypted upon preemption and exported as ciphertext. However, there are more efficient techniques; this section presents some of them.

3.4.1 Localization

Some kernel metadata contains references to non-local resources. For example, Unix file control blocks contain inode numbers, which are global to a file system, and memory descriptors contain physical frame numbers, which are global in the entire system. If these references were exported unmodified, principals could easily forge references to resources they could not access otherwise, e.g. by guessing, and thus compromise protection.

This problem can be solved by translating the references to a local namespace. In the example, the inode number can be represented by a file name under which the inode is known to the principal, and the physical frame number can be translated to a virtual address. Local references can be safely exported because the set of objects they can refer to is now limited by the namespace, and there is no way of forging a reference to anything outside that namespace.

Of course, the local namespace can still contain objects the principal must not access, e.g. names of protected files; in this case, the kernel can validate the references by applying permission checks. Also, the principal can still replace one local reference with another one; however, the same effect can usually be achieved with standard kernel primitives, e.g. by closing one file and then opening another.

3.4.2 Partial preemption

Most kernels contain composite data structures such as lists or trees. These data structures consist of multiple parts that are connected by references. Because such a structure can grow very large, preempting it in one piece can be rather expensive. On the other hand, the individual pieces are hard to export because they contain references to objects in kernel space. If these were exported unmodified, a malicious principal could change them and thus cause inconsistencies in the kernel state; if it is able to guess the address of a privileged object, it could even compromise protection.

This problem can be solved with *pointer swizzling*, a standard technique used in persistent object stores [41]. When part of a composite object is preempted, all references to and from this part are replaced with virtual addresses in resource space, which is local to the principal. These addresses can then be used to locate the referenced objects when the part is restored.

Obviously, a malicious principal can still modify the local references. However, it cannot increase its privileges by doing so because it can only forge references to objects in its local resource space, i.e. to objects it already owns. It could also try to establish a circular reference or create pointers to invalid objects; however, this can easily be detected by the kernel when the objects are restored.

3.4.3 Splitting

Some metadata cannot be localized to a single principal because it describes a relationship between multiple principals, e.g. between a parent and its children, a server and its clients, or between partners in a communication. Unless there is mutual trust among the principals, this metadata cannot be exported to only one of them.

This problem can be solved by splitting the metadata into multiple parts, and by exporting at least one part to each principal. The metadata can then only be fully restored by agreement of *all* principals. Forgery is impossible because each part can be cross-checked with all the other parts.

Obviously, one part must be sufficient to locate the other parts. Therefore, each part must contain a reference to at least one other part, e.g. an identifier for the principal and a virtual address in the resource space of that principal. Because unidirectional references would still permit forgery by guessing, it is advisable to use bidirectional references. If link and backlink do not match, the part can be considered forged and subsequently discarded.

3.4.4 Export unmodified

When a core abstraction is owned exclusively by one principal, most of its state is likely to be under control of that principal anyway. Examples include the general-purpose registers of a thread, the file pointers of an open file, or the data in a message buffer. This state can be modified freely by its owner using system calls and/or hardware instructions; therefore it does not make sense to protect it. Instead, it can be exported unmodified.

However, it may be necessary to validate the state when it is restored. For example, the flags register is usually considered part of the thread context and can be exported directly; however, on some systems, it also contains the current privilege level, which must not be modified by applications and should thus be checked before the register is restored.

3.4.5 Discard and retrieve

Sometimes the kernel must store the same information in different places. For example, some data structures may be supplemented by caches to improve efficiency. The state in such a cache is *derived* from the original data structure. Given that data structure, the state of the cache can be fully reconstructed.

Depending on the amount of work necessary for reconstruction, it may be more efficient *not* to export derived state; instead, it can be discarded upon preemption and reconstructed – perhaps lazily – when it is needed again. This has the additional benefit that, as only one instance of the state is exported, inconsistencies are prevented by design, and the corresponding checks can be omitted.

3.4.6 Cryptographic sealing

Sometimes kernel metadata may not be exportable by one of the above methods, e.g. because of protection issues. In these cases, it may be possible to simply encrypt it and export it as ciphertext. Cryptographic signatures may be used to protect the metadata against tampering.

Of course, without additional means to validate the metadata, this method is only as safe as the cryptographic scheme it relies on; if an application somehow manages to break the encryption, it has unrestricted access to kernel metadata. An attacker can use standard techniques such as replay attacks, dictionary attacks or known-plaintext attacks; dictionary attacks are particularly dangerous for small data structures because only a small dictionary is required.

Additionally, depending on the encryption scheme, this method may be prohibitively slow. Yet, in spite of all those disadvantages, it may be useful if preemptions occur rarely, or for metadata that cannot be exported by any other method.

Chapter 4

Application to L4

To validate our approach, we decided to apply it to an existing kernel. For this experiment, we chose the L4 microkernel because it provides only a small set of core abstractions and therefore does not have to maintain a lot of metadata. Compared to a monolithic kernel such as Linux, this considerably reduces the implementation effort. Also, the virtual memory system in L4 already fulfills many of our requirements and can thus be easily adapted to our scheme.

4.1 The L4 microkernel

The original L4 microkernel was developed by Jochen Liedtke at the German National Research Center for Information Technology [32]. Since then, many extended and revised versions have been designed; for our work, we decided to use the Version 4 API [30]. In this version, the kernel provides only two basic abstractions, threads and address spaces, which are complemented by a synchronous IPC primitive for inter-thread communication. All other functionality is provided by user-level components.

L4 implements the recursive virtual address space model [34], which permits virtual memory management to be performed entirely at user level. Initially, all physical memory is mapped to the root address space σ_0; new address spaces can then be constructed recursively by mapping regions of virtual memory. In L4, this is done synchronously by sending a descriptor as part of an IPC message, which ensures the agreement of both sender and receiver. The actual mapping is established by the kernel during message transfer.

Memory objects can either be *mapped* or *granted*. In the former case, the sender retains ownership of the object and can later use another primitive, *unmap*, to reclaim the object and revoke the mapping, including any derived mappings. In the latter case, ownership is transferred to the receiver, and the object disappears

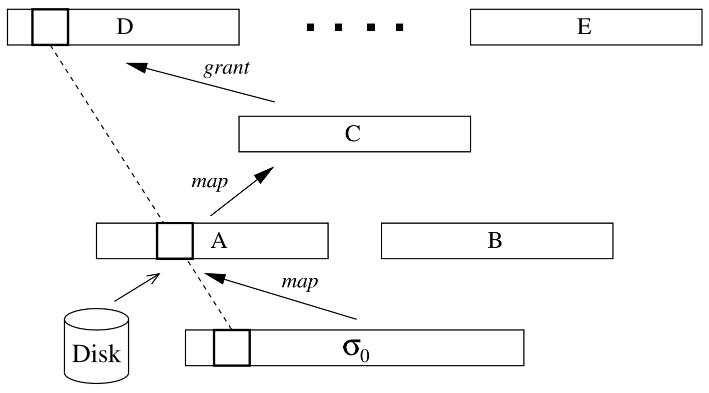


Figure 4.1: Virtual memory primitives in L4.

from the sender's address space. Unmapping is performed asynchronously and does not require consent of the receiver.

Page faults are virtualized by the kernel. Every thread is associated with another thread, its *pager*, which is responsible for managing its address space. Whenever the thread takes a page fault, the kernel catches the fault, blocks the thread and synthesizes a *page fault message* on its behalf, which is sent to the pager via IPC. The pager can then respond with a mapping and thus unblock the thread again.

Furthermore, the kernel has a built-in thread dispatcher which implements a fixed-priority scheme. The thread name space is managed by a special task, which has the authority to create and delete threads through a privileged system call.

4.2 Overview

The L4 virtual memory model already fulfills many of our requirements: It can be performed entirely at user level, supports delegation, has a notification scheme (page fault messages) and primitives for allocation and revocation (`map` and `unmap`). Therefore, we decided not to introduce a separate mechanism for managing kernel memory, but rather to extend the existing scheme.

This approach has the advantage that user and kernel memory can be handled uniformly, i.e. with the same primitives and by the same manager; thus, managers can be simpler and more reusable. Also, a single resource pool can be used for both user and kernel memory; hence, unused kernel memory can be reallocated as user memory and vice versa, which should result in better utilization.

The disadvantage is that some special properties of kernel memory cannot be exploited, the most important being its finer granularity. Many kernel objects, e.g. thread control blocks or address space descriptors, are considerably smaller than a page frame; yet, if VM primitives are to be used, they must be allocated at page frame granularity, which inevitably leads to fragmentation. Nevertheless, we believe that this does not justify the cost of adding another mechanism to the kernel.

Similarly, we did not introduce a separate address space for kernel memory resources. Instead, we decided to use a previously unused virtual memory region, the kernel area, which is part of each address space. This area contains kernel mappings and cannot be used by applications; no valid page faults can happen there, and no memory can be mapped to it, so addresses in this area can never conflict with valid memory addresses. Also, the region is virtual and can thus be managed independently in each address space.

The above decisions result in the following mapping of our scheme to L4:

General scheme	Application to L4
Resource space	Kernel area in virtual address space
Notifications	Page fault messages
Allocation primitive	`IPC` with `map` element
Revocation primitive	`Unmap`
Delegation	Mapping between resource managers

Additionally, we chose an external representation for all exportable kernel data structures. This representation will be discussed in Section 4.7.

4.3 Data structures

The L4 Version 4 API has been implemented for many architectures, including IA-32, IA64, PowerPC, MIPS and Alpha; for some of them, even multiple competing kernels exist. A comprehensive discussion would exceed the scope of this thesis; therefore, we concentrate on one specific architecture, Intel's IA-32. Of the many features of this architecture, only three are relevant for this discussion: Its 32-bit address space, its two-level, hardware-walked page tables and its smallest frame size, which is 4kB.

Most L4 kernels for the IA-32 maintain variations of the following data structures:

- One *page table* per address space, which consists of one root frame and up to 1,024 second-level frames of 4kB each.

- One *map node* and one *cap node* for every memory mapping. These nodes consist of four 32-bit words each and are always paired, reflecting the fact that mappings are always established between two address spaces. The map node describes the recipient's view, e.g. the size and location of the mapping in virtual address space, while the cap node describes the sender's view, such as the memory region from which the mapping is derived, or the privileges it conveys.

- One *node table* per address space, which links page table entries to the corresponding map nodes. It has exactly the same structure as the page table.

- One *kernel thread control block (KTCB)* for every active thread. The KTCB stores the protected part of the thread context, such as state, priority or register contents. Its size depends on the implementation and usually ranges between 170 bytes and 2kB[1].

- One *user thread control block (UTCB)* for every active thread, which holds the user-visible part of the thread context. Each UTCB is 512 bytes long and includes a message buffer of 64 words, which is used for IPC [38].

Additionally, the kernel uses several global variables, such as a timer and a pointer to the current thread. However, the amount of memory occupied by these variables is small and does never change; therefore, the corresponding resources can be preallocated during startup and do not have to be part of the management scheme.

4.3.1 The mapping database

Together, map nodes and cap nodes form the *mapping database*, a tree-like data structure that is used to keep track of all memory mappings in the system (see Figure 4.2). The main reason for having a mapping database is the recursive nature of the `unmap` primitive, which is defined to revoke not only the mapping to which it is applied, but all transitively derived mappings as well. The mapping database not only provides an efficient way to find all of those mappings, it also stores the hierarchy and the conveyed privileges.

An entry in the mapping database is essentially a tuple

$$m = (sbase, sspace, dbase, dspace, length, priv)$$

with the following components:

[1]Some implementations do not have per-thread kernel stacks; also, the FPU/MMX register state is sometimes kept in a different structure, which is allocated on demand.

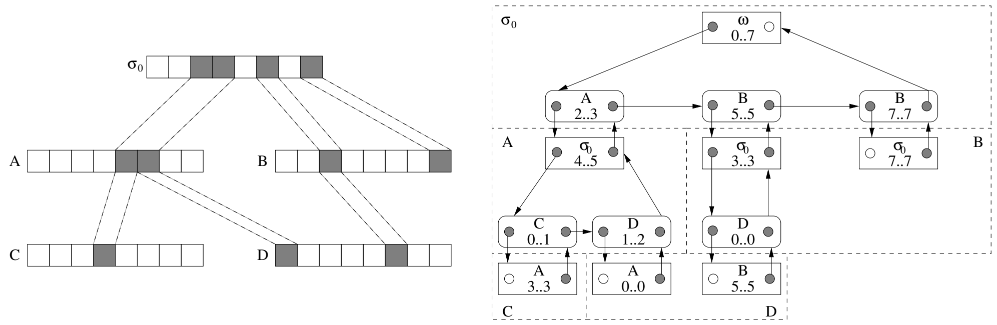


Figure 4.2: Mappings and the corresponding nodes in the mapping database.

Component	Description	Storage
$sspace$	Source address space	MapN
$sbase$	Offset in source address space	CapN
$dspace$	Destination address space	CapN
$dbase$	Offset in destination address space	MapN
$length$	Size of the mapping in bytes	CapN, MapN
$priv$	Maximum privileges conveyed	CapN

Thus, the region $[sbase, sbase{+}length)$ in the source address space is mapped to $[dbase, dbase{+}length)$ in the destination space with maximum privileges $priv$; the effective privileges are obviously limited by the privileges available in the source address space. Both map node and cap node contain a length; the effective length is determined by taking the minimum of the two.

As map nodes and cap nodes are always paired, some implementations have chosen to co-locate them in a single node type. This node can then be much smaller because it does not have to contain mutual references, a source space identifier and a second length; however, it is difficult to find a safe external representation for such a node. More details will be discussed in Section 4.7.

4.3.2 The node table

The node table is a 'shadow' data structure to the page table and has exactly the same structure. For every entry in the page table, it contains a reference to the map node from which it was established. During `unmap`, this reference is used to quickly locate the affected subtrees in the mapping database. In principle, the kernel could also find these subtrees without a node table, for example by scanning the entire database. However, this operation would be very expensive; also, it would be impossible to guarantee isolation since *all* nodes in the database would have to be examined, and not just the ones belonging to the respective subsystem.

Because of the similarity between page and node tables, some implementations have chosen to co-locate them, e.g. by keeping them in adjacent physical frames. In such a scheme, both tables can share a single root frame, which saves one 4kB frame per address space. However, this is not possible in our scheme because the node table serves a dual purpose (it also stores the virtual addresses of preempted map nodes); therefore, we maintain the node table as an independent data structure.

4.4 Accounting

When the kernel requests memory resources, it does so on behalf of a particular resource principal. Therefore, having identified all kernel memory resources, we must now associate each of them with an appropriate resource principal.

4.4.1 Principals

The choice of principal affects the resulting system in the following three ways:

1. **Granularity:** Since every resource request contains an identifier for the faulting principal, it is to be expected that user-level managers will base their accounting scheme on this information. Therefore, if a too large entity is used as principal, the resulting accounting scheme will be unnecessarily coarse.

2. **Utilization:** Since resources allocated to a principal must always be available to it, unused parts cannot be reallocated transparently to other principals and must remain empty. Hence, if a too small entity is chosen as principal, kernel memory may be underutilized as a result.

3. **Protection:** In order to support preemption, an external representation must be chosen for all kernel metadata. However, if the principals are not chosen carefully, e.g. if they can span multiple protection domains, it may be difficult or impossible to find a safe representation.

The only principals in the original L4 API are threads and tasks, i.e. groups of threads sharing the same address space. Therefore, resources must either be associated with threads or tasks, or a new abstraction, such as thread groups or resource containers, must be introduced.

Page and node tables are used to implement address spaces, which are already associated with tasks. Therefore, it seemed natural to use tasks as resource principals for these two data structures. Since address spaces are shared by all threads in a task, a more fine-grain scheme does not make sense; also, there is no protection between threads in the same task, and therefore protection is not an issue.

Both KTCBs and UTCBs are associated with individual threads; however, they are much smaller than the smallest possible allocation of kernel memory (one page frame), and therefore using individual threads as principals would lead to bad utilization. Using groups of threads is possible but requires an additional core abstraction. Therefore we decided to follow the protection argument from above and use tasks as principals for both types of TCBs.

Similar to page tables, map and cap nodes are used to implement address spaces and therefore should be accounted to tasks. However, mappings are usually established between *different* tasks, which leads to the question whether sender, receiver, or both of them should be charged.

As it is the receiver who gets the largest benefit from the mapping, it would be natural to account the entire node pair to the receiver. The opposite approach is also feasible - the sender pays and then charges its client for the resource consumption - but has the disadvantage that metadata from different clients share the same resources and is thus more difficult to separate; for example, it is unclear who should pay for fragmentation.

It is for security reasons that we decided to use the third approach, i.e. to account the cap node to the sender of the mapping and the map node to the receiver. This reflects the fact that mappings can only be established by mutual agreement of sender and receiver during synchronous IPC. If one of them were given control over both nodes, it would be much easier to taint with mappings, e.g. by forging nodes. See Section 5.1 for a detailed discussion.

4.4.2 Sharing

The example of the mapping database shows that it is not always easy to use an existing resource principal for accounting, e.g. when metadata is used by multiple entities. In this case, it may be necessary to introduce a new core abstraction, e.g. resource containers [3]. If there is a fixed relationship between the principals, such as sender and receiver of a mapping, it may also be possible to solve the problem by splitting the data structure and accounting for the individual pieces.

Some kernels maintain data structures such as packet buffers or a global page cache, which are shared across the entire system. This would require a more sophisticated accounting scheme, e.g. one that distinguishes between 'owners' and 'sharers' of a resource. In L4, however, the problem need not be solved at kernel level because the corresponding functionality is implemented by user-level applications.

4.5 Placement

In order to make kernel metadata pageable, it must be part of a virtual address space; therefore, we place all kernel data structures in a special memory region, the *resource area*. This memory region can be backed by resource managers with physical memory frames, just like ordinary virtual memory.

In this section, we discuss possible placement schemes, i.e. ways to assign virtual addresses to kernel objects.

4.5.1 Scope

When the kernel communicates with user-level managers, virtual addresses are used to identify the kernel objects that are being requested, allocated or revoked. Like all identifiers, these addresses are associated with a *scope*, i.e. an area in which they can be used without further qualification. This scope need not be identical with the address space of the corresponding principal; it could span multiple or even all address spaces in the system. If the scope consists of multiple address spaces, the kernel must ensure that non-overlapping address ranges are assigned to all kernel objects in these spaces.

The use of a system-global scope would simplify the assignment of identifiers considerably. For example, it would be possible to directly use the virtual address where the object is stored inside the kernel. In this case, identifiers obviously cannot overlap. However, they could easily be used by applications to circumvent protection. For example, a covert channel could be established by periodically creating a kernel object at a well-known location. Other address spaces can then continually check for its existence, and information can be transmitted by modulating the creation frequency.

The only other scope that is natively supported by L4 is the address space; therefore, choosing any other scope would require an extra kernel abstraction. However, a typical resource manager is expected to be responsible for multiple principals with different address spaces, so space-local identifiers by themselves are not sufficient to identify resources. Fortunately, in all situations where the identifiers will be actually used (during request, allocation and revocation), they are further qualified by the ID of the principal, so the smaller scope is not a problem.

For our experiment, we decided to use space-local placement because this seemed a good compromise between security and implementation effort.

4.5.2 Transparency

There are two fundamentally different ways to assign virtual addresses to objects. One is to use an *opaque* placement scheme in which any object can potentially be placed anywhere; hence, it is impossible to learn anything about an object from its address alone. Another possibility is to use a *transparent* placement scheme in which certain objects are restricted to certain regions; for example, some regions could be reserved for objects of a specific type.

The main disadvantage of using transparent placement for kernel objects is that it inevitably leaks some information about the internal structure of the kernel to user-level managers; for example, it may be possible to infer the size or purpose of certain kernel objects. Also, even the mere presence or absence of kernel objects at a particular address may convey information.

On the other hand, the use of transparent placement has considerable advantages because it enables resource managers to make better policy decisions. For example, if the address allows a distinction between important objects and less important ones, the managers can respond to memory pressure by revoking frames with less important objects first. If the address conveys a dependency between objects, managers may choose to revoke entire subgroups of objects, which results in better utilization because the dependent objects are useless without their parent objects anyway.

Of course, if a manager bases its policy on that kind of information, it is no longer portable across different platforms or even different kernel versions on the same platform. However, if portability is an issue, the manager can still treat transparent addresses as opaque.

For our experiment, we decided to use transparent placement. This gives us the opportunity to explore policies that make use of the additional information. Also, this does not cause any security issues in L4 because from the kernel data structures, managers cannot learn anything about applications other than their own clients, who must have implicit trust in them anyway.

4.5.3 Binding

The binding between kernel objects and their addresses can either be *static* or *dynamic*:

- **Static binding:** The kernel defines a function p which, for every potential kernel object o, computes an address range $p(o)$ where it is to be placed. In order to avoid conflicts, $p(o) \cap p(o') = \emptyset$ must hold for all pairs of objects (o, o') that can exist at the same time.

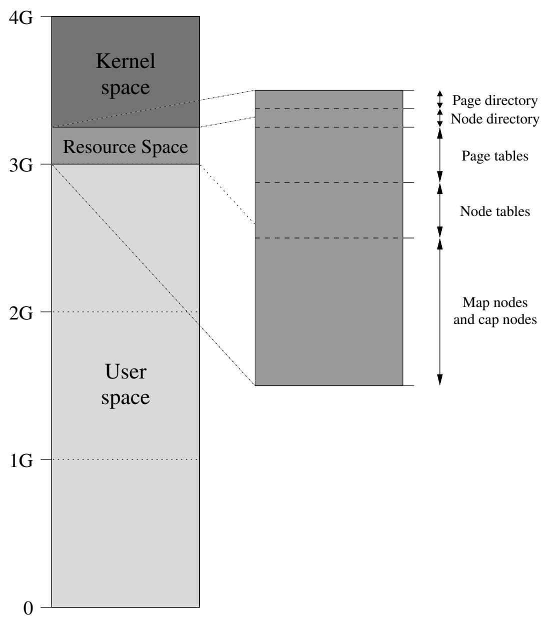


Figure 4.3: Address space layout. Due to static placement, page and node tables are allocated at a fixed offset, whereas map and cap nodes are dynamically placed in the designated region.

- **Dynamic binding:** Whenever a kernel object o is created, the kernel chooses a virtual address range p_o for it and stores the binding (o, p_o) throughout the lifetime of the object. In order to avoid conflicts, the address range must not overlap with existing objects.

The static scheme has the obvious advantage that no additional metadata is required to store bindings. Also, it avoids the problem of having to manage resource space explicitly, e.g. by keeping a list of free address ranges and a placement policy to choose one of them.

On the other hand, dynamic binding is often more efficient when the objects are allocated with a finer granularity than the memory used to back them. The difference is most evident for sparsely populated data structures, where static binding can lead to high internal fragmentation.

In our experiment, fragmentation particularly affects the mapping database. Consider the case where every frame in a particular address space has been mapped to every other address space. This would require $n_f \cdot (n_s - 1)$ cap nodes, where n_f is the number of frames and n_s the number of address spaces in the system. Hence, a static address range for cap nodes must be large enough to accommodate that many nodes. However, in a realistic system, only very few of those nodes will be in use at any one time, and they are unlikely to have contiguous addresses. In the worst case, every frame-size part of resource space will contain at most one node, and kernel memory utilization will drop to an unacceptable $\frac{4 \cdot 8}{4096} \approx 0.8\%$.

Therefore, we decided to use static binding only for the page and node tables. Because of the hardware-dictated page table format on IA-32, these data structures must be allocated with frame granularity anyway, so fragmentation cannot be prevented. For the mapping database, however, we used dynamic binding and a simple slab allocator to distribute new nodes to existing frames.

Like in any dynamic binding scheme, we also had to decide where to store the bindings between existing nodes and their addresses. In order to avoid the need for additional metadata, we use pointer swizzling, i.e. we replace all references to a preempted node with its address. Thus, the address of a preempted map node can be found by looking it up in the node table.

UTCBs already have addresses in the original system. These addresses are located in a special region of virtual address space, the UTCB area, which is considered to be part of resource space. KTCBs are not subjected to the paging scheme at this time (see Section 4.7.5); instead, their allocation is controlled with the `ThreadControl` system call.

4.6 Request and allocation

When the kernel detects that a principal needs additional kernel metadata to complete an operation, it must request the necessary memory from a user-level resource manager. In L4, this is accomplished by raising a page fault in the resource area of the faulting task, which is then handled by the corresponding pager. The faulting thread executes the following algorithm:

```
suspend faulting operation
determine fault address
find responsible pager
repeat
   send page fault message to pager
   wait for reply
   check supplied mappings
until mapping acceptable
import kernel objects from mapping
resume faulting operation
```

In this section, we describe the individual steps of this algorithm in more detail. We also discuss security and safety issues, and how these are resolved in our system.

4.6.1 Sending the request

In order to request a missing metadata object m, the kernel must first determine the virtual address a_m at which to raise the fault, i.e. the address where m is located in the resource area. If static placement is used, this can be achieved simply by using the placement function p to compute $a_m = p(m)$. In the case of dynamic placement, the address may be stored in another metadata object[2]. If this object is not present, it must be requested first.

Once the address has been established, the kernel must choose a pager to handle the request. There are many possible choices, for example:

- A global pager that provides kernel memory to all principals
- The standard pager of the faulting principal
- A special *k-pager* that provides kernel memory to the faulting principal

Using a single global pager is the simplest solution; however, we believe that this would only be a small improvement over having a built-in kernel policy. The second scheme is more restrictive than the third because it does not offer a simple way to manage user and kernel memory in separate tasks. Nevertheless, we decided to use the second scheme because we wanted to demonstrate that it is sufficient; the third scheme is an obvious extension and can be added where the additional flexibility is required.

After the kernel has identified the pager, it suspends the faulting operation and synthesizes a page fault message on behalf of the faulting thread. This message is sent via synchronous IPC, so the faulting thread may have to wait until the manager becomes ready to receive. After the message has been delivered, the thread waits until the manager responds by mapping the requested resource.

4.6.2 Checking the allocation

Once the request has been sent to the pager, the faulting thread waits for a resource allocation, i.e. for a reply message containing one or multiple `map` elements. However, not all resources can be accepted in this state. Obviously, the faulting thread should accept a memory frame if all of the following conditions hold:

1. The frame is supplied by its current pager
2. It backs the region in which the fault occurred

[2]The address is only stored if the object exists but has been preempted; however, unless the kernel *knows* that the object does not exist, it must check the address anyway.

3. The pager grants read and write privileges on the frame

For security reasons, the thread should never accept kernel memory from third-party threads because they could later preempt the resources and access the metadata that is exported in them. If the third condition would be omitted, memory could be mapped read-only to the kernel, e.g. to implement copy-on-write sharing; however, the kernel would then have to enforce the write protection, which complicates the implementation and exceeds the scope of our experiment. We consider this future work.

If the second condition is not mandatory, pagers can use prefetching to supply resources that are likely to be needed in the future; however, principals must then have complete trust in their managers because they are free to prefetch critical metadata, e.g. TCBs. The equivalent problem for user memory can be solved by using a *region mapper*, a pager thread which acts as a proxy for page faults [2, 17]; however, this approach does not work very well for kernel memory because the region mapper would depend on the same resources as its clients. Therefore, we decided to keep the second condition.

In a system that supports memory-mapped I/O, the kernel should also check whether the supplied frame is part of main memory. Otherwise, an attacker could use device memory to circumvent access restrictions on kernel metadata.

4.6.3 Accepting a resource

When the kernel decides to accept a certain frame for use as kernel memory, it must first ensure that the frame cannot be accessed from user level, e.g. by removing it from all page tables and invalidating all relevant translations in the TLB. This is necessary to maintain protection, but it is also a precondition for the following steps because it prevents race conditions. On an SMP system, these race conditions could lead to a Time-of-Check, Time-of-Use (TOCTOU) vulnerability.

Once the kernel has exclusive access to the frame, it must check its contents for exported kernel objects and re-import them where necessary. This import may fail for various reasons, e.g. when inconsistencies are detected or an invalid object is found. In this case, the kernel has multiple options:

- It can discard the frame and raise an error,
- It can discard the frame and re-send the page fault,
- It can postpone the import and proceed with the next object, or
- It can silently discard the faulty object and proceed

In cases where the problem clearly results from tampering, the kernel is probably under attack and should choose one of the first two options. If the object is not essential (i.e. not the cause of the original fault) and the problem is likely to be caused by missing metadata, the third option is applicable. In all other cases, the kernel should act *in dubio pro reo* and assume that the problem is due to changes in kernel state while the object was unavailable; thus, it should choose the fourth option.

Additionally, the kernel must store both the *fact* that the frame is allocated for kernel metadata and its *virtual address* in the resource area. The former is necessary to prevent the frame from being allocated twice, and to trigger the export of the metadata when it is reclaimed, whereas the latter is needed for localization[3]. Where static binding is used, the virtual address can also serve as a type identifier, e.g. to distinguish page tables from node tables.

In our experiment, we used a *frame table* for this purpose. The frame table has one entry for each physical page frame in the system. It holds virtual addresses; frames currently allocated to user level are labeled with an invalid address. Since the size of the frame table does not change at runtime, it can be preallocated during startup and need not be subjected to the paging scheme.

4.6.4 Deadlock avoidance

In the original L4 API, many operations were explicitly or implicitly defined not to cause page faults; for example, thread switches or message transfers with untyped elements were guaranteed to succeed. This is different in our system, where a thread may lack almost any metadata, e.g. a page table or even one of its TCBs. Since page faults always block one or multiple threads, care must be taken to avoid deadlocks and to resolve them where they do occur.

Fortunately, it is possible to avoid many deadlocks by arranging the pagers in a hierarchy. In such a system, page faults are always escalated to the next level, and a circular wait situation involving multiple pagers cannot occur. However, deadlock is still possible if a page fault is raised while a pager is blocked by one of its clients, e.g. during IPC (see Figure 4.4). This cannot be avoided by design because the pager must use IPC to receive page fault messages, and to send a response.

It is possible to design the kernel in a way that allows at least the request to be sent without additional resources, i.e. with the KTCB only[4]. However, the client

[3]The threadID of the owner need not be stored because it can either be inferred from the contents of the page – for example, the owner of a map node can be determined from the node table link – or is irrelevant because the metadata, e.g. a page table, is discarded anyway.

[4]Clearly, the thread must have caused the fault somehow; therefore, it must already be known to the kernel, i.e. its KTCB must be present.

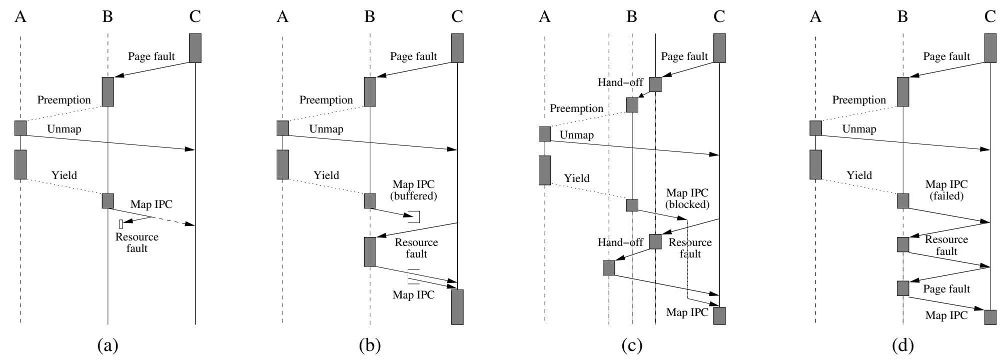


Figure 4.4: Deadlock caused by a resource fault (a) and different resolution techniques: Buffering (b), multi-threaded pager (c), abort and retry (d).

may require resources to receive the reply; for example, a page table can only be accepted if the page directory is already present. If another page fault is raised in this situation, deadlock occurs immediately.

In our system, we use *fault ordering* to avoid this problem whenever possible. If the kernel detects that it needs multiple resources $r_1, r_2, \ldots, r_n$ to complete an operation, it chooses an order $(i_1, i_2, \ldots, i_n)$ such that r_{i_j} does not depend on any r_{i_k} with $k > j$. Such an order always exists in our system because the resources form a hierarchy and therefore no circular dependencies can exist. The kernel then requests the resources in that order. Thus, deadlock can be avoided.

4.6.5 Deadlock resolution

Even when fault ordering is used, deadlock can occur due to third-party actions. Consider the example in Figure 4.4, where client C requests a new page table from its pager, B. While the request is being handled by B, a higher-level pager A revokes C's page directory. B cannot know this because when the message was delivered, the page directory was still available. Therefore it tries to supply a page table to C as requested, which leads to a deadlock situation.

Avoiding this problem by system design would require a disproportionate effort; basically, kernel memory would have to be pinned for the entire lifetime of its owner. Therefore, the situation must either be prevented or resolved when it occurs. There are several possible approaches, for example:

1. **Buffering:** The kernel accepts the useless resource on behalf of the client and keeps it in an internal buffer while the second fault is handled.

2. **Multi-threaded pager:** The pager uses different threads for receiving faults and sending responses; hence, its receiver thread never blocks.

3. **Abort and retry:** The deadlock is detected and resolved by aborting the IPC and delivering an error code to the pager. The client retries, i.e. it restarts the page fault handler. Fault ordering is used to avoid a second deadlock.

The buffering approach requires additional metadata to store waiting resources and introduces various special cases into the kernel. Multi-threaded pagers require an additional IPC[5] to do the hand-off; also, even the top-level pager σ_0 would have to be multi-threaded, and it is not easy to determine how many threads it would need.

Aborting the IPC seems dangerous because there is a possibility for starvation. However, the pager can detect this situation from the error code and then take adequate measures, e.g. resolve the second fault immediately. Therefore, we decided to use the third approach.

4.6.6 Faults during system calls

The main responsibility of the L4 microkernel is to maintain protection, i.e. to control the interaction between tasks. For this purpose, it offers several system calls that can be used by sufficiently privileged tasks to interact with one another. This interaction always involves resources; therefore, the kernel must be able to deal with a situation where either the caller or one of the affected tasks lacks a necessary resource.

This situation is different from an ordinary page fault in that it can involve multiple principals. If the standard page fault algorithm is applied, *all* of these principals must be suspended until the fault is resolved. Thus, if a malicious task colludes with its pager and prevents it from handling the fault, it can block the other tasks indefinitely. Therefore, all system calls must be checked for this vulnerability, and measures must be taken to remove it where it occurs.

The L4 Version 4 API defines twelve system calls. Four of these calls, `Kernel-Interface`, `SystemClock`, `ProcessorControl` and `MemoryControl`, involve only global metadata, which can be preallocated during startup. Two others, `ExchangeRegisters` and `Lipc`, can only be used on threads in the same task, which cannot be protected from each other anyway. The remaining six system calls can involve multiple tasks and thus require closer inspection.

The `ThreadSwitch` call can be used to yield the CPU, or to switch[6] to a

[5]Unfortunately, using a lazy thread switch [38] does not help much because the worker threads never reply to the receiver thread

[6]A similar argument can be applied to ordinary context switches, which occur e.g. when a higher-priority thread is unblocked, or when the current time slice expires. These switches can be thought of as implicit calls to `ThreadSwitch`.

specific thread t. This can cause page faults if t lacks essential resources such as its address space, which is required for execution. However, as long as t's KTCB is present, the page faults can be raised in the target context, and the caller is not blocked. Since KTCBs are not paged in our experiment, this is always possible, and the problem cannot occur.

The `Unmap` call is used to revoke memory mappings. In order to find transitively derived mappings, the kernel may need to traverse a subtree of the mapping database, which in principle could lead to page faults when preempted nodes are encountered. In our system, this problem is prevented by eagerly disabling mappings when their map node is preempted or disconnected from the main tree; therefore, preempted nodes can be safely skipped during `Unmap` (see also Section 4.7.3).

The `IPC` call already contains a timeout mechanism to handle preempted resources. In the original API, this is needed when a page fault occurs during message transfer; both sender and receiver can define a *transfer timeout* to specify how long they are willing to wait for the page fault to be resolved. This mechanism can easily be extended to resource faults.

For the remaining three system calls, `ThreadControl`, `SpaceControl` and `Schedule`, timeouts cannot be used because these calls must never fail. `ThreadControl`, for example, is used to delete threads; if a malicious task could reliably deflect this call, there would be no way to remove it from the system. Furthermore, the first two calls can only be executed by a privileged system task; if the calls were allowed to block, this task would be vulnerable to Denial-of-Service attacks.

In our system, we avoid this problem by moving all relevant metadata to the KTCB, which is not paged; therefore, the three calls can never raise a page fault. We are fully aware of the fact that this can only be a temporary solution. However, we believe that the problem is due to structural issues in the API, and that it can only be thoroughly solved by removing those issues. Section 4.7.5 discusses this problem in more detail.

4.6.7 Nested faults

A *nested fault* is a page fault that occurs while another page fault is being handled. There are two ways to handle a nested fault: One is to handle it *instead* of the original fault, the other is to *stack* it on top, i.e. to suspend the original fault while the nested fault is being handled. When faults are stacked, it is important to determine the maximum nesting depth and to allocate sufficient space (e.g. on the kernel stack) to store the corresponding contexts.

In our system, faults are never stacked, and each thread is allowed to request only one memory resource (user or kernel memory) at a time. If a nested fault occurs, the handling of the original fault is aborted. This may seem dangerous because information about the original fault is lost; however, if its cause persists, the fault will reappear once the faulting operation is resumed. Still, the work performed to handle the original fault is lost; yet, we believe that this situation occurs infrequently when fault ordering (Section 4.6.4) is used.

Therefore, the kernel only needs to be able to store two contexts: One for the faulting operation and one for the page fault handler. If the page fault handling code is short, it can be executed non-preemptively, and thus the second context can be very small; basically, only the address of the requested resource is needed. If separate k-pagers are used (Section 4.6.1), resource faults can be stacked onto user page faults, and the maximum nesting level rises to three.

Special care must be taken to prevent infinite nesting, which can occur when circular dependencies exist between resources. A principal that requests one of those resources would perpetually cause resource faults and could never make any progress. Hence, circular dependencies must be avoided.

In our system, `map` is the only primitive where such a dependency could exist. Specifically, if mapping memory to the kernel would require a map node, a fault that is caused by a lack of map nodes could never be handled. Fortunately, since the kernel only accepts physical memory, the corresponding information can be stored in the frame table, and a map node is not required.

4.7 External Representation

In order to allow for preemption of kernel memory, it must be possible to safely export kernel metadata to user level. In our scheme, this is done by converting the metadata into an external representation. The resulting state can be safely re-imported into the kernel once the memory is restored.

In Section 3.2.4, we already defined requirements for such an external representation, and Section 3.4 presented various conversion techniques. In this section, we apply these techniques to each kernel data structure in L4.

4.7.1 Page tables

Page tables define the translation from virtual to physical addresses. On IA-32, the architecture we chose for our experiment, the CPU uses a hard-wired algorithm to parse page tables; therefore, it is mandatory to use the format that is defined by the hardware.

IA-32 page tables have two levels. The first level consists of a single page frame, the *page directory*, which contains 1,024 word-size entries. Each of these entries corresponds directly to a 4MB region in the 4GB virtual address space and can either directly specify a translation (a so-called *superpage*) or point to another frame, a *page table*. Similarly, a page table has 1,024 entries, each of which specifies a translation for a 4kB region in virtual address space. Translations may be invalid; in this case, an attempt to access the corresponding region causes a page fault.

In addition to the actual translation, i.e. the mapping from virtual to physical address, a page table entry also contains various control bits, e.g. for privileges and cache behavior, and two status bits: One to indicate that the page has been accessed, the other to indicate that the content has been modified. The status bits are updated by the hardware.

External representation

With the exception of the status bits, page tables contain only redundant information; their entire state is also stored in the mapping database. For example, the physical address of a page can be found by locating the corresponding map node and tracing its parents back to σ_0, which has a direct virtual-to-physical mapping. Hence, page tables are *derived* data structures, which makes them ideal candidates for the discard and retrieve technique (Section 3.4.5).

In order to successfully apply this technique, we must be able to store the information in the status bits elsewhere. However, only two bits per entry are required, and these are easily stored in the corresponding map node, which can be located using the node directory. Of course, both the map node and the node directory must be present to do this; however, when either of them is preempted, the page table entry must be invalidated anyway, which causes the bits to be saved.

Therefore, we chose to discard the contents of preempted page tables, and to retrieve the corresponding state from the mapping database when it is needed again. Thus, the external representation of a page table entry is a constant, e.g. zero.

Efficiency

Because the Discard and Retrieve technique throws away kernel state, it is important to show that not too much work is wasted, i.e. that reconstructing the state is not prohibitively expensive.

In order to retrieve a missing translation, we use the node directory to look up the map node for the corresponding region. We then follow the parent link in the map node to trace the mapping back to σ_0, or to an address space that already has the translation – whichever comes first. If σ_0 is reached, the physical and virtual address of the last map node are equal, and the correct translation can easily be derived; otherwise, the translation is simply copied from the other address space.

To further reduce the necessary effort, we use lazy evaluation. When a page table is restored, it is filled with invalid translations; these are then replaced with the correct values when a page fault occurs. Thus, the above algorithm need not be executed for all valid entries, but only for the current working set.

Safety

As the external representation is constant, an attacker cannot gain any information from it. Also, since the external representation is overwritten when it is restored, modifying it does not have any effect.

4.7.2 Node tables

Node tables define a mapping from virtual memory regions to map nodes. For every virtual address, they can be used to either locate the map node that backs this address, or to establish that such a node does not exist. If the node exists but has been preempted, the node table contains its address in resource space instead of a direct reference. This information is necessary for sending a resource fault; it cannot be determined otherwise because dynamic binding is used for map nodes (see also Section 4.5.3).

Since node tables are 'shadow' data structures to page tables, they also share the same two-level structure. The first level consists of a single memory frame, the *node directory*, which can contain links to second-level *node tables*. Each frame consists of 1,024 entries that are in one of the following states:

- **Node reference:** The entry contains a direct reference to a map node, which is located in kernel memory.
- **Node address:** The entry contains an address in resource space, which can be used to send a resource fault for the node.
- **Link:** The entry contains a direct reference to a node table (node directory only)
- **Invalid:** The entry contains a well-known value to indicate that it does not hold any of the above.

The resource addresses are obtained by pointer swizzling when the corresponding map node is preempted. Once the map node is restored, they are eagerly replaced with direct references again. Therefore, it the kernel encounters a resource address during a lookup, there are two possible cases:

1. If the region in the resource area that contains the address is currently not backed with kernel memory, the kernel raises a resource fault in that area and retries after the corresponding page has been restored.

2. If the region is already backed with kernel memory, the kernel checks whether it contains the correct map node at the specified address. This map node may still be in external representation and may have to be imported first. If the node has a mapping for the correct memory region, the resource address is replaced with a direct reference; otherwise, the address is invalid, and the entry is reset to the Invalid state.

External representation

Together with the mapping database, node tables form a large composite data structure which is connected by references. As it is entirely infeasible to export this data structure as a whole, we apply the Partial Preemption technique (Section 3.4.2) and use pointer swizzling to localize all references.

Node table entries can be exported directly if they are either in the Invalid state or contain a resource address. Pointer swizzling is applied to direct node references; the result is a resource address which is local to the principal's address space can be exported without further modification. Because static binding is used for node tables, entries in the Link state need not be exported and can be reset to the Invalid state.

Therefore, a node table in external representation consists only of resource addresses and invalid entries. If the entire data structure is preempted, the result is a tree in the local resource area (see Figure 4.5).

Efficiency

In order to export a node table, each entry must be checked; links must be replaced with the well-known value for invalid entries, and node references must be converted to resource addresses. The latter requires only a lookup in the frame table (to determine the virtual address of the page) and a mask-and-set operation.

Importing node tables also requires inspection of each entry; however, the kernel only needs to ensure that all entries are either invalid or point to the resource area and are properly aligned. Link entries in the node directory are restored automatically when the kernel imports the corresponding node tables; resource addresses are converted to direct references when they are accessed, using the above algorithm.

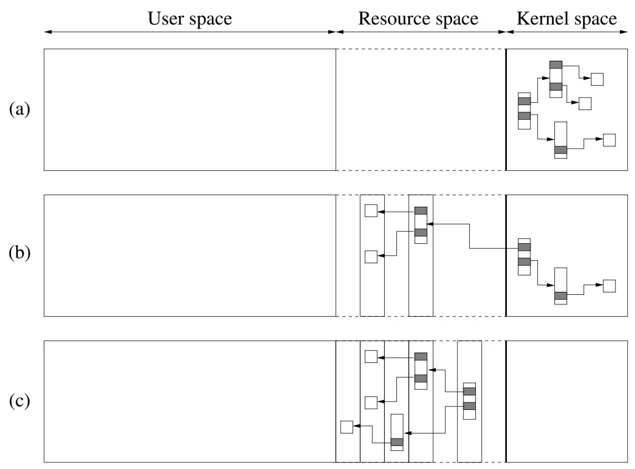
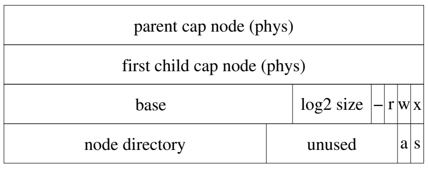


Figure 4.5: Node table and referenced map nodes in the kernel (a), partially preempted (b), and fully preempted (c).

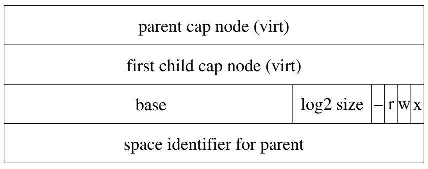
Safety

The external representation is safe because all values are fully localized to the resource area of the principal. No kernel addresses are exported. Therefore, only the following attacks need to be considered:

- Forging references to non-existent map nodes. This will be detected when the kernel attempts to dereference them, and they will be replaced with Invalid entries.
- Re-targeting references to different map nodes. This equals to intra-address space granting, which is irrelevant for protection. Principals with more than one thread can perform this operation anyway.
- Replacing references with invalid entries. This equals to a Flush operation (i.e. an `unmap` in the local address space), which the principal could have done anyway.
- Inserting misaligned references, or references outside the resource area. These can be detected during import and replaced with invalid entries.
- Inserting references to other data structures in the resource area, e.g. to TCBs. This can be detected if part of the resource area is dedicated to map nodes; references outside this area can then be discarded.
- Establishing a circular reference. This is impossible because Link entries are neither exported nor imported.



(a) Internal representation



(b) External representation

Figure 4.6: Map node

4.7.3 Mapping database

The mapping database is a tree-like data structure that keeps track of all memory mappings in the system. It is necessary because in L4, memory mappings are recursively derived from other memory mappings; therefore, when a mapping is revoked, all derived mappings must be found and revoked as well.

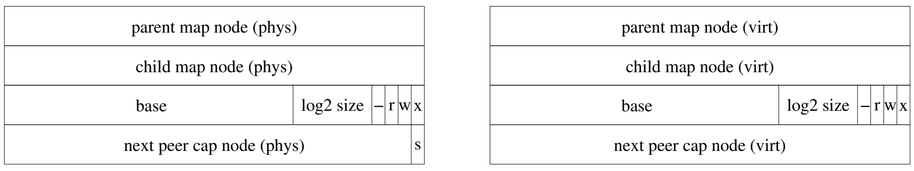
A mapping always exists between two regions, a *source region* and a *target region*. This fact is reflected in the structure of the mapping database, where the two regions are represented by different node types, cap nodes and map nodes (see also Section 4.3.1). As every mapping is derived from exactly one other mapping, every map node has exactly one parent cap node. However, different mappings may share the same parent mapping, and thus a single map node may have multiple child cap nodes (see Figure 4.2).

The map node we used in our experiment (see Figure 4.6) contains the following fields:

- An *uplink* to the parent cap node, i.e. the cap node that describes the source region of the mapping,
- A *downlink* to the first child cap node, if it exists. Such a node describes a mapping that is directly derived from the parent mapping,
- The *base* and *size* of the target region in the virtual address space,
- An *access bit cache*, which is used to store access information from preempted page tables (see Section 4.7.1),
- A pointer to the *node directory* of the address space which holds the mapping, and
- Several *control bits*, which are used internally by the kernel.

The corresponding cap node (see Figure 4.7) has the following fields:

- An *uplink* to the parent map node, i.e. the node that represents the mapping from which this mapping is derived,



(a) Internal representation (b) External representation

Figure 4.7: Cap node

- A *downlink* to the child map node, which describes the target region,
- The *base* and *size* of the source region relative to the parent mapping, which can be used to derive smaller mappings from larger ones;
- The *maximum privileges* conveyed by the mapping (the effective privileges also depend on the current privileges of the parent),
- A *sibling pointer* to the next cap node derived from the same map node, and
- A *control bit* which is used internally by the kernel.

In this implementation, every map node is associated with exactly one cap node. Hence, it is not possible to construct a large mapping from multiple smaller ones; if such a mapping is attempted by the user, multiple map nodes must be used. This restriction applies to every L4 implementation known to the author; since it is not relevant for our experiment, we do not address it here.

The subtree property

The mapping database is a large composite data structure; exporting it as a whole would be costly and ineffective. Therefore, it is advisable to use a more fine-grain scheme such as Partial Preemption (Section 3.4.2), which allows pagers to operate on small subsets of the database, down to individual map nodes. However, the fact that the mapping database is used for revocation makes it dangerous to apply this scheme directly.

Consider the following scenario: Pager σ_0 has mapped a page frame to B, who maps it on to a subsystem that consists of C, D and E (Figure 4.8). Assume C colludes with its pager B, who preempts the region of kernel memory that contains the association between B and C. As a result, the subtree below C is disconnected from the database and cannot be reached from B any more. Therefore, when σ_0 decides to revoke the page frame, the kernel can only locate the mapping in A's and B's address spaces; the other mappings will remain unaffected.

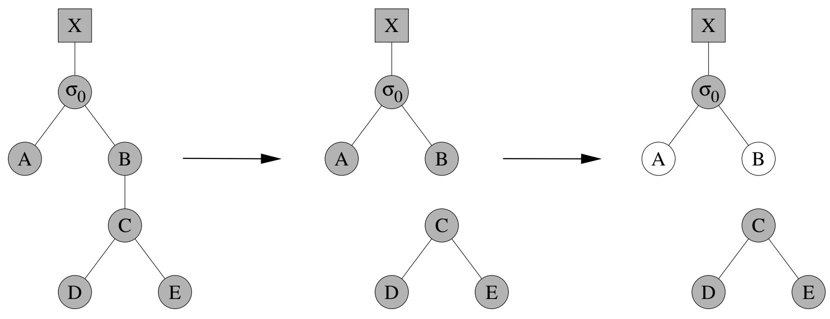


Figure 4.8: Detached subtree in the mapping database. In the first step, B preempts the mapping to C. If now σ_0 attempts to revoke permissions on X, the kernel cannot reach the subsystem below C any more.

Clearly, a scheme that allows subsystems to 'hijack' memory frames would compromise protection. Therefore, the kernel must make sure that it can reach all effective mappings in the system, i.e. the following *subtree property* must hold for the mapping database:

> *While the root node of a subtree in the mapping database is preempted, the mappings in that subtree must not convey any privileges.*

If this property holds, the kernel can safely skip preempted nodes in the mapping database when it encounters them during `unmap` because even if a disconnected subtree existed below one of these nodes, it cannot convey privileges any more, and there is nothing to revoke.

Cascading unmap; page fault storms

One possible way to enforce the subtree property is to simply revoke mappings when they are preempted. Because `unmap` is recursive, the corresponding subtrees in the mapping database disappear entirely, and the subtree property holds. Although this operation discards a lot of state, it is safe in L4 because pagers must always consider the possibility that a higher-level pager revokes one of their mappings; therefore, they must be prepared to handle page faults even on mappings they did not revoke themselves. Thus, a subtree that is destroyed during preemption can always be reestablished with page faults.

However, as map nodes are relatively small, a single frame of kernel memory can contain many of them (up to 1,024 in the worst case). If such a frame is preempted, *all* of the corresponding subtrees would have to be destroyed; later, an equivalent number of page faults would be necessary to restore them. Because pagers and their clients usually reside in different address spaces, this would require many expensive context switches.

Moreover, some of the affected mappings could back other frames of kernel memory, which might e.g. contain other map nodes. For the reasons stated above, these mappings would have to be revoked also, resulting in a cascade that can affect thousands of mappings, possibly the entire system; the subsequent 'page fault storm' can then be very expensive. For a pager, there is usually no way to tell *a priori* whether this will happen for a particular page of kernel memory; therefore, it is difficult to find a sensible policy.

Persistent mappings

Because of the subtree property, there is no easy way to avoid revoking all the mappings in a preempted subtree. However, the subsequent page fault storm is much more expensive (due to the many context switches), and there are two simple and effective techniques to reduce this cost:

1. **Exporting map nodes:** If map nodes are exported rather than discarded, it is possible to restore many of them with a single resource fault.

2. **Keeping disconnected subtrees:** Although they do not convey any privileges, these subtrees contain many 'dormant' mappings that can easily be restored once the root node is reconnected.

With these two techniques in place, it is possible to restore a preempted page of map nodes with a *single* resource fault. Ideally, the resource fault is handled before the preempted subtrees can cause any user page faults; in this case, the nodes in the page are re-imported and reconnected to their dormant subtrees – resulting in exactly the same situation as before the preemption. In a more realistic case, some mappings will be restored before the reconnection and must be replaced afterwards; however, this should still be far less expensive than raising page faults for each of the remaining mappings.

The disadvantage of this solution is that it is inconsistent with the existing L4 mapping semantics. Consider the scenario in Figure 4.9 where two memory frames, X and Y, have been mapped to several tasks. If B removes the mapping to C, X will also disappear from the virtual address spaces of D and E. Assume C now takes a page fault that causes B to establish a mapping to Y in the same location. In the current L4 model, D and E do not have access to this mapping and must send page faults to C; with our proposed changes, Y will appear in all three address spaces at the same time, and no further page faults will happen.

One possible solution is to treat the classical `unmap` primitive differently from the preemption of map and cap nodes. Hence, if B removed the mapping with the `unmap` primitive, the mappings D-E and D-F would be revoked as well, whereas if B simply preempted a map node, the mappings would remain 'dormant' and

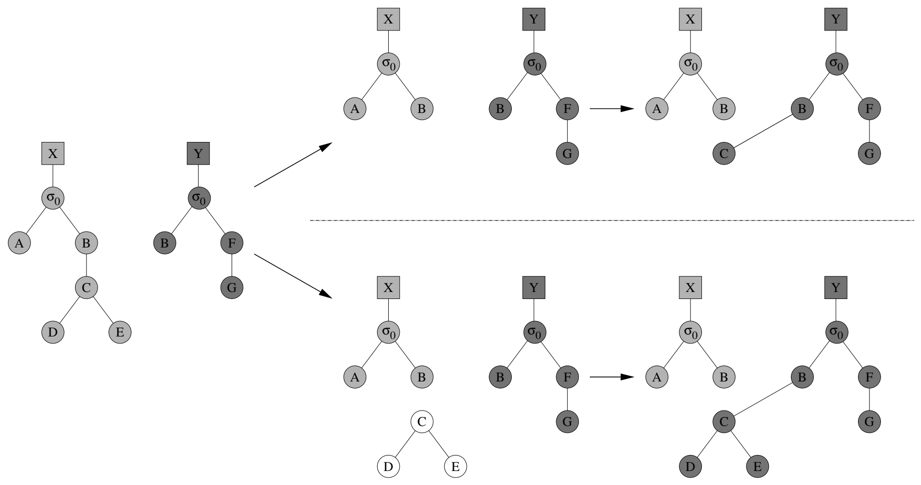


Figure 4.9: Difference between recursive unmap (above) and persistent mapping (below). In both cases, B first unmaps frame X and then maps frame Y to the same location.

could be reestablished. Both cases would look the same to D, yet they would have to be handled differently, which would require an extra protocol between B and D.

However, we think that such a distinction is unnecessary and that the modified semantics can be applied to unmap as well. We argue that the behavior of the overall system is not changed, because most of the 'dormant' mappings would have been reestablished in the current model anyway.

Consider a pager σ_P that implements a policy P on the memory it provides. This policy will be consulted only at certain points in time, the *decision points*. For example, if σ_P implements LRU, it will need to perform page aging at regular intervals; if it implements quotas, it will have to check them whenever a client requests additional resources.

However, the decision points of different pagers do not normally coincide. When, for example, σ_P is paged by another pager σ_Q which implements paged virtual memory, σ_Q may choose to revoke and swap out a page at any point in time. Now, if one of σ_P's clients needs that page, σ_P will see a page fault on a page it has already allocated. We do not see any reason why σ_P should consult its policy at this point; instead, it will probably remap the same page again. With persistent mappings, the result would have been the same, except that σ_P would not have been involved.

For this reason, and because it reduces the side-effects of map node preemption, we decided to implement persistent mappings in our experimental system. This leads to the following change in the semantics of map and unmap:

	Old semantics	New semantics
`map`	*Transfers* a memory region. The receiver is given access to all resources that are present in the source region *when the mapping is received*	*Links* two memory regions. The receiver is given access to all resources that are present in the source region *while the mapping exists*
`unmap`	Removes all directly derived mappings from a memory region. All indirectly derived mappings are *revoked*	Removes all directly derived mappings from a memory region. All indirectly derived mappings are *deactivated*

In the mapping database, unmapping a map node m does no longer remove the entire subtree below m, but only the cap nodes that are direct children of m. The map node m itself is only removed when the mapping is *flushed*, i.e. removed from the principal's address space as well.

External representation

In order to export the contents of the mapping database, we apply a combination of multiple techniques. As the mapping database is shared by all principals in the system, it inevitably contains some inter-domain connections, namely between cap nodes and their associated map nodes. Because such a connection can exist between principals that have only limited trust in each other, we use the Splitting technique to export one half of the connection to each of them.

The Splitting technique requires principal identifiers to implement the cross-domain references. In the current L4 Version 4 API, however, the principals in question (address spaces) do not have explicit identifiers; they are named implicitly by the ID of a thread they contain. Although this is not unproblematic (see Section 5.3.2), we adopt this scheme for our experiment. When an address space identifier is needed, the kernel randomly chooses the ID of a thread in that address space.

The remaining connections (between a map node and its first child cap node, or between a cap node and its next sibling) are intra-domain and can therefore be exported with Partial Preemption. Most of the actual contents in both nodes can either be exported unmodified or simply discarded.

More specifically, the individual fields of a map node are exported in the following way:

- The *uplink* to the parent cap node may cross a domain boundary; therefore it is represented by the threadID of the corresponding principal and a virtual address in the resource area of that principal.
- The *downlink* to the first child cap node is always local; thus, it is sufficient to convert it into a virtual address in the resource area.
- Both *base* and *size* are local to the principal's address space and can be exported unmodified.
- The *access bit cache* is exported unmodified because it is irrelevant for protection.
- The *node directory reference* always points to the node directory of the principal to which the node is exported; therefore, it can be discarded and later restored.
- The *control bits* can be regenerated by the kernel; thus, they can be discarded.

Note that, although the external representation has an additional field (the threadID of the principal that owns the parent node), the Monotony requirement is not violated because the discarded and unused parts yield enough free space.

The contents of a cap node are exported in the following way:

- The *uplink* to the parent map node and the *sibling pointer* are local and can be converted to virtual addresses.
- The *downlink* could actually be discarded because due to the subtree property, this link is never traversed downwards while the node is preempted. However, the threadID of the corresponding principal is needed for validation.
- Both *base* and *size* are local to the source region and can be exported unmodified.
- The *maximum privileges* are exported unmodified because the principal can gain nothing by raising the maximum beyond its own privileges.
- The *control bits* can be regenerated and are discarded.

It may be surprising that the virtual address part of the downlink can simply be discarded. However, consider that the downlink is only needed during `unmap`. If it is virtual, this means that the corresponding subtree is either preempted or has been freshly imported; in either case, the subtree property guarantees that it does not convey any privileges, and `unmap` can simply skip it. As soon as the first dormant mapping is touched, the page fault handler uses the uplinks to walk the tree in the reverse direction, restoring all downlinks to direct references along the way.

Efficiency

In order to export a page of map and cap nodes, each live node must be modified. Discarded fields must be invalidated, e.g. by overwriting them with a constant value, and references must be localized, i.e. physical addresses must be replaced with addresses in the resource area. For map node uplinks, the threadID of the other principal must also be determined and stored. Additionally, the backlink that is associated with each reference must be localized in a similar fashion.

Converting a physical forward link to a virtual address requires a lookup in the frame table; for backlinks, the virtual address is easily determined from the address of the page that is being exported. In order to find the threadID for a map node uplink, the kernel follows the link to the parent cap node and then proceeds to the next map node. This node contains a reference to a node directory, which in turn has a reference to a KTCB in the corresponding address space. The global threadID in this TCB is used[7].

Importing a page of map and/or cap nodes does not require any immediate work; instead, the nodes can be imported lazily when they are first used. To import an individual node, the kernel must convert the virtual reference back into a physical pointer, which requires a node table lookup; it must also regenerate the control bits and, if a map node is being imported, insert a reference to the current node directory. If any of the previous steps fails, the kernel must invalidate the node. Sanity checks for the exported values (base, size, and privileges of the mapping) are not required because they are relative to the corresponding values in the parent node.

Safety

The *base* field in a map node specifies the offset where the mapping appears in the recipient's address space. It is safe to export this value to the recipient because he can easily modify it by granting the mapping to another thread in the same address space. Invalid values (e.g. offsets outside of the user address space) can easily be detected during the import.

[7]If the thread whose ID is chosen here is migrated later, all map nodes containing this ID are broken and cannot be re-imported (see Section 5.3.2).

The *size* field in a map node specifies the size of the mapping in the recipient's address space. This value can be safely exported to the recipient because it is limited by the size of the mapping on the sender side, i.e. the recipient can gain nothing by increasing it beyond that limit. By decreasing this value, he can only reduce his privileges because he loses access to part of the mapping.

The *access bit cache* field is safe to export because the recipient can control it anyway (by accessing the mapping). Theoretically, a malicious principal could try to hide the fact that it has accessed a page by resetting this value. If this is a problem, the bits can be transferred to the parent mapping upon preemption.

The *base* and *size* field in a cap node specify which part of the sender's address space is accessible to the recipient of the mapping. By modifying the base address, the sender can change the location of this part. For the recipient, this means that the contents of the mapped region can suddenly change; however, the sender can achieve the same simply by writing to the region. By decreasing the size, the sender can make part of the mapping disappear in the recipient's address space; however, it can also do this by invoking the `unmap` primitive. Increasing the size does not have any effect because the size field is replicated in the recipient's map node, and the effective size of the mapping is determined by the minimum of the two.

The *maximum privileges* field in a cap node is safe to export because it only specifies an upper limit; thus, the sender cannot elevate its privileges by increasing this value. Decreasing it effectively revokes privileges from the recipient of the mapping, which can also be achieved with the `unmap` primitive.

The *cross-domain references* in map and cap nodes are safe to export because they are always paired, and the two parts are always controlled by different principals (except for intra-domain mappings, which are not relevant for protection anyway). An individual principal can gain nothing by modifying such a reference because the mapping is not effective unless the two parts match.

4.7.4 User thread control blocks

The user thread control block (UTCB) stores the user-accessible part of a thread context, including a message buffer and various Thread Control Registers (TCRs), which are considered a logical extension of the thread's register set. Some TCRs are used by the kernel, e.g. to deliver error codes, while others are controlled by the thread itself, e.g. to specify timeouts.

Unlike the other data structures discussed in this section, UTCBs do not contain protected kernel state; in fact, the API specifies that they can be freely accessed from user level, just like ordinary registers. Hence, it is unnecessary to take any special action during export because each thread can see the full contents of its UTCB anyway; also, the values need not be checked during import because the threads can write them at any time, and therefore the kernel must check them on every use.

4.7.5 Kernel thread control blocks

Kernel thread control blocks (KTCBs) are used to hold the protected part of a thread context, such as scheduling information, the current status of the thread, or its presence and position in various internal queues.

In our experiment, we did *not* attempt to page KTCBs because this would have required significant changes to the L4 API, which is beyond the scope of this work[8]. For example, the kernel is currently specified to contain an internal, priority-based scheduler, and therefore the KTCB contains scheduler-related information. If this information were preempted, the kernel would lose knowledge of the very *existence* of the corresponding thread, and thus would not generate page faults unless some system call was invoked on that thread. Also, the thread's association with its pager is currently kept in the KTCB; without this information, the kernel cannot even determine where to send the page fault.

As the L4 Version 4 API has been designed without a particular focus on kernel memory management, it is not surprising that these problems exist. However, we believe that some changes to the API, e.g. with respect to the scheduling model and the thread naming scheme, are sufficient to solve them. Our group is currently working on these changes, and we are confident that we can eventually page the *entire* kernel state, including KTCBs.

In the remainder of this section, we analyze the contents of the KTCB and provide suggestions how each item could be exported.

Register contents

On IA-32, the basic register context of a thread consists of the instruction pointer (EIP), the stack pointer (ESP), seven general purpose registers (EAX-EDX, ESI, EDI, EBP) and a status word (EFLAGS). The FPU and SSE units are both equipped with additional registers, but these need only be saved when the thread actually uses those units.

As the entire register context is freely accessible to the thread, it can be exported unmodified. With the exception of some bits in the EFLAGS register[9],

[8]Even with this restriction, it is possible to implement a comprehensive resource management scheme because KTCBs are only allocated during `ThreadControl`, which is a privileged system call that can only be invoked by special tasks in the trusted computing base

[9]In addition to the usual status bits, the EFLAGS register also holds the I/O privilege level (IOPL) and several system flags, which must not be written by application programs. See Section 3.4.4 of the IA-32 Manual [23] for further details.

validation is not required either. However, since the FPU and SSE contexts are rather large, we suggest that these be allocated on first use, rather than during thread creation. They can then be paged separately, although this means that a thread can take a resource fault just from accessing an FPU register.

Thread status

The status of a preempted thread indicates the operation the thread was executing when it was preempted (e.g. running in user level or performing a system call), as well as the status of that operation (e.g. blocked by another thread).

All previous L4 implementations known to the author use a per-thread kernel stack in each KTCB, along with some dedicated fields, to store this information. The resulting programming model is convenient because each kernel stack contains activation frames from all kernel functions the corresponding thread has called; thus, a kernel-level thread switch is essentially a stack switch. However, the stack content is very sensitive because it is interspersed with return addresses and local variables; therefore, it must not be exported directly.

Fortunately, in many kernels, the number of preemption points is quite small; it is therefore sufficient to export a *continuation*, i.e. a brief summary of the respective situation, to user level. The continuation need not contain constant values, such as return addresses, or values that can be obtained elsewhere. If necessary, the exact contents of the kernel stack can later be restored from the continuation.

Scheduling information

According to the Version 4 API, the kernel must contain an internal round-robin scheduler with a fixed priority scheme. Therefore, the KTCB contains the current priority of the thread, the length of its current and remaining timeslice, and its total time quantum.

This information is difficult to export because it cannot be validated. Unlike memory mappings, which are always relative to the parent mapping, priorities and time periods are absolute and can easily be forged.

One possibility would be to use Cryptographic Sealing; however, as the range of these fields is quite small, the kernel might become vulnerable to dictionary attacks. Another solution is to move to a hierarchical scheduling model in which a data structure similar to the mapping database would be used to convey time and priority. Finally, it might be possible to move the entire scheduler to user level, and to replace it with a simple dispatching mechanism; in this scenario, scheduling metadata could be restored by user-level schedulers (e.g. using ’time faults’).

Address space affiliation

Kernel TCBs contain a reference to the address space in which the thread currently executes. In principle, this reference should be discardable because we chose tasks as principals; KTCBs should be implicitly associated with the address space of the task that requests them. However, tasks are not first-class objects in L4, and thus page fault messages must actually be sent by threads; therefore, this reference must be exported anyway.

This problem is complicated by the fact that address spaces, being second-class objects, cannot be named directly; in space-related system calls such as `SpaceControl`, they are referenced by the ID of a thread that executes in them. Therefore it is impossible to simply export a name, even if the kernel could somehow validate it to prevent forgery.

We believe that this problem should be solved by promoting address spaces to first-class objects (see Section 5.3.2).

Global ThreadIDs

In addition to the global ID that is part of each thread's context, KTCBs also contain threadIDs as references to other threads, e.g. to identify their pager, exception handler, or partner in an IPC. Because these IDs have a system-global scope, they can easily be forged by guessing.

We believe that this can be accomplished by replacing global threadIDs with a local naming scheme. Such a scheme is currently being discussed by the L4 community.

Timeouts

The `IPC` system call allows various timeouts to be specified by the user; for example, threads can specify how long they are willing to wait for their partner to become ready, or for the message transfer to complete. Each thread can have at most one active timeout at any point in time; this information is also stored in the KTCB.

Exporting timeouts is not difficult (they can be fully controlled by the respective thread anyway), but it is not obvious what the semantics should be when a preempted timeout expires. If an immediate effect is desired, then the kernel must either retain enough information to detect it, which obviously requires some kernel memory, or notify some user-level entity, for example the thread's scheduler. This entity can then ensure that the timeout is re-imported before it expires.

We believe, however, that it is sufficient to make the timeout effective when the respective KTCB is re-imported. The only case where this actually makes a difference is when the thread uses the timeout to enforce some guaranteed timing behavior, e.g. in a real-time application. In this case, the thread must negotiate a pinning contract [37] with its pager anyway, and it should be easy to extend this contract to the KTCB.

Run queue affiliation

All runnable KTCBs are part of an internal queue, the *run queue*, which is consulted by the dispatcher when a time slice expires and a context switch to the next thread must be performed. This affiliation is important for two reasons: First, the *fact* that the thread exists and is running must be preserved, because otherwise the thread will never be chosen by the dispatcher and can run only if another thread switches to it. Second, the *position* of the thread in the queue must be preserved to maintain fairness[10]; if multiple threads are runnable at the same priority, each of them should be dispatched with the same frequency.

With a hierarchical scheduling model, the first aspect (presence) can easily be exported. The second aspect (position) could be represented either by the ID of the 'next' thread or by the point in time when the thread was last executed; both values would have to be protected against tampering because they are difficult to validate.

Another option would be to use a central, trusted pager for the run queue, but this would compromise the distributed design of our paging scheme; for example, pagers would no longer be able to obtain a complete 'snapshot' of the subsystems they manage.

Finally, the run queue could be managed completely by user-level schedulers that rely on the kernel only for performing context switches. It remains to be seen whether such a scheme can be implemented with acceptable performance.

Send queue affiliation

Since the IPC mechanism in L4 is synchronous, a thread that is trying to send a message to another thread may be blocked if the other thread is not ready to receive. In this case, its KTCB is placed in the *send queue* of the target thread. Whenever a thread performs a receive operation, the kernel checks its send queue for waiting threads and, if applicable, wakes the first one.

Again, both the *fact* that a KTCB is part of a send queue and its *position* in that queue must be preserved; the latter is necessary to prevent starvation. The first aspect can be represented by the ID of the owner of the queue, which is safe

[10]Apart from mentioning time slices and priorities, the current L4 specification does not give any details on scheduling behavior. Thus, fairness is not explicitly required; however, we consider this an important feature.

Structure	*Field*	*Technique used*
Page Tables	Physical Address	Discarded, retrieved from mapping DB
	Control bits	Discarded, regenerated
	Status bits	Saved to map node
Node Tables	References	Localized with pointer swizzling
Map Node	Uplink	Converted to threadID, virtual address
	Downlink	Localized with pointer swizzling
	Base + Size	Not modified
	Access Cache	Not modified
	Node Dir Ref	Discarded, regenerated
	Control bits	Discarded, regenerated
Cap Node	Uplink	Localized with pointer swizzling
	Sibling Pointer	Localized with pointer swizzling
	Downlink	Converted to threadID
	Base + Size	Not modified
	Max Privileges	Not modified
	Control bits	Discarded, regenerated
User TCB	All fields	Not modified
Kernel TCB	All fields	Not pageable

Table 4.1: Techniques used to export L4 kernel data structures

to export because the thread possessed it when it invoked the send operation. The second aspect can, again, be exported as a 'next' pointer or as the point in time where the send operation was invoked. Note that the ordering of the send queue may also depend on the priorities of the waiting threads, and that the position information may only be relevant within a particular priority level.

Care must be taken that the presence of preempted KTCBs in a send queue does not block other threads, because otherwise a malicious client could run a denial-of-service attack against a server by sending to it and then having its own KTCB preempted. To avoid this, we suggest that preempted KTCBs be skipped; however, the resource fault should be triggered anyway to ensure that the preempted thread gets a chance to complete its send operation.

4.7.6 Summary

Table 4.1 once again summarizes the L4 kernel data structures and the techniques we used to export the individual fields.

4.8 Preemption and revalidation

In our system, it is possible to gracefully decrease the allocation of kernel memory of any principal; this is done by *preempting* pages that have been allocated to the kernel. Once a preemption is triggered, the kernel performs the following steps:

1. It determines the type of metadata that is stored in the page, e.g. by consulting the frame table,

2. It converts the metadata to external representation, and

3. It restores user-level access to the page

The process of finding the external representation has already been described in the previous section; here, we discuss other related issues, e.g. by which events preemption can be triggered, and which resources are affected.

4.8.1 Trigger events

Obviously, preemption must occur when the kpager – or any higher-level pager – invokes `unmap` on a frame that has been previously allocated to the kernel. However, for security reasons, the kernel must also consider a page preempted when the page is merely *accessed* from user level.

Consider a system where a pager σ_1 backs another pager, σ_2, who in turn provides read/write memory regions to its clients. In an ordinary L4 system, σ_2 can rely on the fact that nothing its clients could do will ever affect its own privileges. Therefore, it need only be prepared to handle page faults that are caused by σ_1; if it has negotiated a pinning contract with σ_1, it can be certain that no page faults will happen.

In our system, however, a client can use part of its region as kernel memory, which makes the corresponding frames disappear from *all* other address spaces, *including* σ_2. Of course, σ_2 can still avoid unexpected page faults by unmapping each page before accessing it, and by remapping it afterwards; however, this is slow, cumbersome and also completely superfluous, since logically, σ_2 still has read and write privileges on its pages. Instead, we chose to preserve the original semantics by transparently preempting kernel memory that is being accessed from user level.

Note that pagers can use this scheme to perform simple system calls on their clients; for example, a loader could write to the TCB of a new task to initialize its instruction pointer.

4.8.2 Inter-resource dependencies

The sum of all kernel metadata can essentially be thought of as a huge directed graph, which is connected by references in individual metadata instances. Care must be taken that preemptions do not partition this graph; otherwise parts of it may become orphaned, and the kernel must take countermeasures, e.g. perform garbage collection.

For this reason, there are some intrinsic dependencies between different kernel resources, which are summarized in the following table:

Resource	*Depends on*
Page directory	Node directory
Page table	Page directory
Page table entry	Page table, Map node
Node directory	—
Node table	Node directory
Node table entry	Node table, Map node
KTCB	Node directory
UTCB	KTCB
Map node	Node table entry
Cap node	Map node

The dependency relation is transitive. Since tasks were chosen as principals, all resources have a direct or indirect dependency on the node directory, which ultimately represents a task in our system.

Obviously, page and node tables depend on the respective directories because they cannot be located without it. Map nodes do not depend on node table entries conceptually; however, without this restriction, re-importing node tables would be difficult because the referenced map nodes could still be in the kernel, but there would be no way to locate them, except by scanning the entire mapping database.

4.8.3 Cyclic dependencies

Because of the inter-resource dependencies, a single preemption event may lead to multiple pages actually being exported. Hence, if the dependency graph contains a cycle, the kernel might be deadlocked or caught in an infinite loop.

With the exception of the mapping database, all kernel metadata has a strictly hierarchical structure which does not allow for cycles. However, map nodes can indirectly back other parts of the mapping database and thus other map nodes, and therefore a cycle is possible. Such a cycle cannot be formed with `map` alone because there is a strict temporal ordering (a page must be mapped to the kernel *before* it can back other mappings); however, cycles can occur if the `grant` primitive is used.

Consider a system in which a page p is mapped from a task A to another task B, who chooses to map it as kernel memory to C, for use in the mapping database. Assume A now grants p to C. If C now revokes the new mapping, the derived mapping in B must also be revoked, which in turn revokes it from C, and so on.

One possible solution to this problem is to conceptually reduce p to ordinary memory *before* starting to export its contents. Hence, the kernel will not attempt another export when it encounters p again.

4.8.4 Pages with mixed content

Some metadata has a finer granularity than memory frames; in Section 4.5.3, we already mentioned that we collect several instances in each frame using a simple slab allocator [6]. However, in order to export this metadata correctly, the kernel must be able to detect whether each slab is in use or not, and to determine the correct data type.

The first problem is easily solved by marking empty slabs with a well-known pattern that is different from all valid metadata instances[11] (e.g. all zeroes). The second problem can be avoided if a) the kernel never co-locates objects of different types in the same page and b) the one type can be determined by other means, e.g. from the offset of the page in the resource area.

In our system, this solution is applicable for all kernel objects except for the map and cap nodes, which are co-located to reduce fragmentation. Hence, a single page of kernel memory can contain both node types. To allow the kernel to distinguish between the two, both nodes contain a special bit at a fixed offset, which is always set in a map node and always clear in a cap node. Empty nodes can easily be identified because all of their bits are zero.

[11]Obviously, such a pattern does not exist if *all* possible values correspond to a valid instance. In practice, however, a suitable pattern can usually be found, e.g. by breaking an alignment restriction.

Chapter 5

Analysis

In the previous section, we described the details of an experimental system in which our scheme for managing kernel memory was applied to the L4 microkernel. Now we discuss this system at a more general level, e.g. with respect to protection and security. We also analyze the differences between our system and the original L4 model, and we make various suggestions for future revisions of the L4 API.

5.1 Protection and security

One important goal of this work is to show that kernel memory can be managed at user level *without* weakening protection. Unfortunately, we are unable to provide a formal proof; to our knowledge, such a proof has never been attempted, not even for microkernels. Instead, we examine a number of typical attacks and show that they cannot be successful in our system.

When considering potential attacks, we use the system shown in Figure 5.1 as a reference. In this figure, threads are represented as circles, and address space boundaries are denoted by rectangles. The arrows indicate pager-client relationships. For example, the threads B, C and D reside in one address space; B pages its clients C and D and is itself paged by A.

We assume that the root pager σ_0 is part of the trusted computing base and cannot be compromised. However, an attacker may be able to control several subsystems (the shaded regions, i.e. the threads B-D and G), including the respective pagers (B and G). Some of the malicious pagers may collude with their clients (B with C and D) or other malicious pagers (B with G), while others may be able to gain control over other, well-behaving subsystems (G over H).

In order to maintain protection, the kernel must ensure the following:

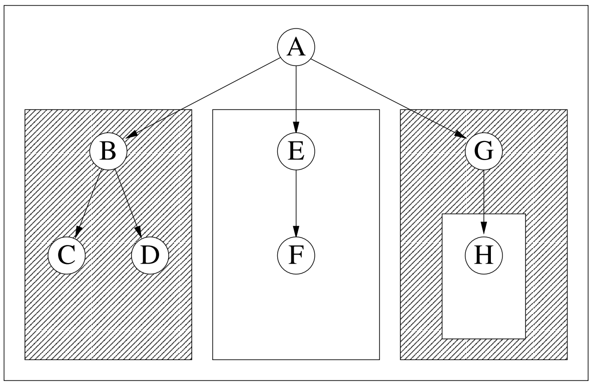


Figure 5.1: Reference model. The arrows indicate pager/client relationships; shaded parts are controlled by the attacker.

- Malicious clients must not be able to affect their pagers; for example, A must be protected from B.
- Malicious subsystems must not be able to affect other, independent subsystems; for example, E and F must be protected from G.
- Malicious servers must not be able to affect their clients, except by denial of service. For example, if E uses a service provided by B, then B must not be able to exert any control over E beyond that service.

However, the kernel *cannot* protect subsystems that obtain kernel memory from a malicious pager. In existing L4 systems, this restriction already applies to program code, since a pager that can control the code executed by its client can effectively do anything that client could do itself. Hence, the L4 protection model is not significantly weakened; clients can ensure their safety by obtaining their kernel memory from a trusted pager.

In the following, we distinguish between two classes of attacks: *metadata-based attacks* rely on the pager's ability to access the kernel metadata of its clients, while *page-fault-based attacks* exploit the fact that a pager can raise resource faults by unmapping pages of kernel memory.

5.1.1 Metadata-based attacks

A pager that provides kernel memory to a client can, by preempting that memory from the kernel, gain access to the contents in their external representation. It can then freely examine, modify, forge, copy, permute, invalidate or destroy the data in such a page. Afterwards, when the corresponding resource fault is raised, it can return the page to the kernel, where the contents are converted back to internal state.

Examining metadata

By *examining* a page that is exported vacant, i.e. a page directory or page table, the pager cannot gain anything. UTCBs are safe because they can be freely accessed by the client and therefore the pager could also examine them by modifying the client's code. The remaining state, node tables and the mapping database, convey information about the layout of the client's address space, including mappings provided by other pagers.

Most of this state, i.e. size and position of the mapping, refers only to the client and is therefore readily available to the pager. For mappings provided by another pager, some information can be gathered, namely the threadID of the other pager and the position of a cap node in its address space. Similarly, for mappings provided to clients, the threadID of the recipient can be determined. However, this information is mostly useless.

Modifying metadata

By *modifying* or *forging* metadata, neither the pager nor its clients can obtain elevated privileges because the external representation is safe (see Section 4.7). For example, neither B nor D can, by editing the exported part of the mapping database, change the privileges or the source area of existing mappings from A because they can only affect the map node; this is checked by the kernel against the corresponding cap node, which is controlled by A. For similar reasons, they cannot obtain new mappings either, e.g. from E, because the forged map node would have to contain a reference to a matching cap node in E's address space. Guessing a reference to an existing cap node, e.g. from a mapping E has provided to F, does not help because that cap node would contain F's threadID.

However, two malicious subsystems (e.g. B and G) could forge a matching pair of map and cap nodes if they know each other's threadID, and thereby establish a mapping *without using IPC*. This is not directly relevant for protection because the new mapping can be derived only from resources already present in either B's or G's address space; however, it can be used to circumvent IPC-based control mechanisms such as Clans and Chiefs [31] or IPC redirection [24] when they are used to monitor or to restrict communication [25].

This problem could be solved by handling re-imported mappings as if they were established using IPC, i.e. by submitting them to the monitor for inspection. Another option would be to replace the system-global threadIDs with a local naming scheme. In such a scheme, two principals that are not allowed to communicate directly would not even have a name for each other; thus, there would be no way they could forge a valid reference pair.

Copying metadata

In another potential attack, the pager copies a valid piece of kernel metadata to its own address space for later use; for example, if D has obtained a mapping from E, B could try to save the corresponding map node to recapture the mapping when E revokes it. This is called a *replay attack* and is a classical threat to cryptographic schemes. In contrast to forging, the attacker is presumed to already have access to a valid, albeit stale instance of metadata, which can be a considerable advantage e.g. if the scheme is protected by sparsity.

In our system, validation is used to detect stale references. Therefore, copying metadata is no more dangerous than forging or modifying.

Permuting metadata

Since pagers usually provide more than one page of kernel memory to their clients, they can preempt multiple pages and remap them in a different permutation. A malicious pager can swap pages from different clients, pages containing different data types, different instances of the same data type, and even data structures within the same page.

This attack is very similar to the modification attack. If a pager preempts two pages x and y, it can, instead of mapping x in place of y, simply copy the contents from x to y and vice versa. Neither the pager nor the clients can use this to increase their privileges because kernel metadata is always interpreted in the context of the client on whose behalf it is imported; hence, if the client did not have access to a resource before, it cannot gain it by importing the metadata of another client.

Consider the following example: A maps a memory object to C, and B preempts the page that holds the corresponding map node. The external form of the map node will then contain A's threadID and the virtual address of the corresponding cap node in A's address space. Now B attempts to remap the map node to D. During import, the kernel uses the reference in the map node to look up the cap node and checks if the two nodes match. However, since the cap node contains C's threadID, this is not the case and the kernel discards the mapping.

Protection can neither be compromised by swapping kernel objects of different types, e.g. by mapping a page of cap nodes instead of a node table. Since the kernel raises resource faults for a specific type, it assumes metadata of this type is supplied in return[1]. Therefore, metadata of a different type will be considered an

[1]This is safe to assume even if the pager has no concept of kernel data types, since the kernel identifies the requested metadata by the virtual address of the page fault.

invalid instance of the expected type and is subsequently discarded.

By swapping data structures of the same type, the attacker can exchange the user-visible state of two threads (by swapping UTCBs) or change the virtual addresses of mappings in its address space (by swapping node tables or map nodes). However, these operations can also be carried out using standard kernel primitives; therefore this does not constitute a security problem.

If metadata is permuted in external form, the physical address of individual instances of kernel metadata changes. This is *not* a security problem if the kernel does not retain any physical pointers to the metadata after it is exported (there is no reason to).

Invalidating or destroying metadata

If the attacker has access to the external form of a kernel object, it can damage it or remove it entirely. In both cases, the object cannot be re-imported, and the kernel must discontinue any services for which this object was required.

Obviously, this is not a security problem if the services in question are provided to the attacker, or to one of its clients. This is true for all the metadata we exported in our experiment, except for cap nodes. By destroying a cap node, the pager can effectively revoke the mapping that is conveyed by that node. However, if that mapping was also provided by the pager, it can achieve the same effect by invoking the `unmap` primitive; if the mapping was provided by another pager and only forwarded by the client, the pager can easily force the client to perform the `unmap` for him, e.g. by replacing one of its code pages.

Inferring kernel internals

By inspection of exported kernel metadata, an attacker might be able to draw certain conclusions about the protected *internal state* of the kernel. For example, assume UTCBs are allocated with a global placement scheme, i.e. two different UTCBs are never placed at the same location in the resource area, even if they reside in different address spaces. In this case, the attacker may be able to estimate the total number of threads in the system by creating a thread and observing the location of its UTCB. In our experiment, this is not possible because the resource areas in different address spaces are completely independent from each other.

Also, an attacker can, by observing metadata and its development over time, infer some of the *policies* used by the kernel. For example, by checking the mapping database periodically, the attacker may be able to determine the strategy used to allocate new map or cap nodes. This information might be helpful in an attack.

5.1.2 Page-fault-based attacks

A kpager receives resource faults from its clients. It can respond to these faults by sending a mapping, or it can choose not to respond. A kpager that already provides memory to the kernel can also induce resource faults in its clients by unmapping that memory at any time.

Page hijacking

Since the kernel runs in supervisor mode, it has full access to all physical memory; in particular, it can write pages that are mapped read-only in user level. Therefore, a kpager could try to circumvent access restrictions by mapping its memory to the kernel.

Consider the following scenario: A maps a code page read-only to both B and E. Now B provokes a resource fault in one of its clients, e.g. in D, and maps the code page in response. If the kernel permits this, it will protect the page and attempt to convert the contents into kernel metadata; subsequent accesses in E will then preempt the page and cause the metadata to be exported again.

Since the original content – a page of code – is extremely unlikely to coincide with a valid instance of metadata, the kernel must, in most cases, partially discard it; also, the metadata might change while in the kernel. Thus, the re-exported content will differ from the original, and B has effectively modified a read-only page.

However, this attack is not possible in our system because in response to a resource fault, the kernel only accepts mappings that convey at least read and write privileges. In this case, giving the page to the kernel is unproblematic because the attacker can destroy the content of the page also by overwriting it directly.

Withholding metadata

By design, the kernel does not impose any deadlines for handling kernel page faults; hence, a kpager may delay its answer indefinitely or even not reply at all, e.g. because the client has exceeded its allocation. While this inevitably blocks the client requesting the metadata, other, independent subsystems must remain unaffected.

An attack based on withholding metadata can only be effective for metadata whose in-kernel representation contains references to resources held by other principals. Therefore, page and node tables need not be considered since they are local data structures used exclusively by the respective principal.

UTCBs contain two threadIDs which designate the pager and the exception handler of the thread. When the thread takes a page fault or an exception, these threadIDs are used to send a message to the respective handler. In those situations, however, the faulting thread depends on the handler thread to resolve the fault; even if the reference is valid, the handler can easily block the thread by not sending a response.

The mapping database contains many cross-principal references; an attacker can revoke part of it, which causes some references to be disconnected. This could be a problem during unmap, when the kernel must traverse a subtree of the mapping database in order to find all transitively derived mappings, starting from the mapping that is being revoked. However, the subtree property (Section 4.7.3) guarantees that active mappings can only exist in the part of the mapping database that is connected to the original mapping in σ_0; therefore, the kernel need not raise any page faults during unmap, and the requestor is never blocked.

Note that withholding metadata will be a far more important problem in future systems where queues, e.g. the send and ready queues, are exported from the KTCB. In such a system, care must be taken to allow the kernel to parse these queues even if some of its members are preempted; see Section 4.7.5 for further discussion.

Flooding

A malicious client can perform a classical denial-of-service attack against its pager by continuously preempting some of its kernel memory and thus flooding the pager with a large number of resource faults.

This attack is similar to a flooding attack based on ordinary memory and can be handled in the same way, e.g. by counting page faults and throttling clients that cause an excessive amount. However, resource faults are considerably more expensive than ordinary page faults since the kernel must re-import the metadata in the page. Therefore, it is important to internally account this operation to the faulting principal, e.g. by using its time slice rather than that of the pager.

5.2 Differences to L4

By introducing our scheme for kernel memory management into L4, we have changed several aspects of the original L4 model [30]. Most of these aspects have already been briefly mentioned in previous sections. In the following section, we explain the changes in more detail and describe their consequences.

5.2.1 Resource faults

In the original L4 model, page faults can only occur in user space, i.e. in the user-accessible part of virtual address space, as a result of a thread's reading, writing or executing the contents of a page, or during IPC. Furthermore, the UTCB area and the Kernel Interface Page (KIP) are exempt from page faults since these objects are provided by the kernel and are backed with kernel memory.

In our system, the first exception does not hold any more, since UTCBs are now provided by user-level pagers. Additionally, page faults may now also occur in the resource area, where page tables, node tables and the mapping database are located.

Reading, writing and executing can still trigger page faults, but they are no longer limited to non-existent mappings; page faults may now also be used to request missing page tables, node tables or map nodes.

All page faults that could happen during IPC in the original system are still possible; however, some additional faults can now occur. The L4 API [30] classifies these faults in three categories:

- *Pre-send page faults* happen in the sender's context before the message transfer has started, without involving the receiver thread;
- *Post-receive page faults* happen in the receiver's context after the message has been transferred, without involving the sender thread;
- *Transfer page faults* happen while a message is being transferred and involve both sender and receiver.

The additional faults are caused by accesses to resources that either the sender or the receiver requires to complete IPC but does not currently hold:

	Sender	Receiver
UTCB	Pre-send	Xfer
Page directory	–	–
Page table	–	–
Node directory	Xfer	Xfer
Node table	Xfer	Xfer
Map node	–	Xfer
Cap node	Xfer	–

Faults in the receiver can cause a deadlock, which is resolved by aborting the IPC (see Section 4.6.5). Since the pager must be able to detect this, a new error code is required.

Furthermore, resource faults can now occur during certain system calls that were previously defined not to raise any faults, namely the `ExchangeRegisters` and `Unmap` primitives. `ExchangeRegisters` needs access to the UTCB of the destination thread, which may not be available; `Unmap` needs to update or remove the map and cap nodes of the mappings that are being revoked[2].

5.2.2 Persistent mappings

In the original system, the `unmap` primitive recursively revokes all derived mappings; this involves erasing the corresponding subtree in the mapping database. We avoid this behavior in our system because it can lead to a cascading effect (see Section 4.7.3). Instead, only directly derived mappings are revoked; the remaining, transitively derived mappings are deactivated and remain dormant until the respective subtree in the mapping database is reconnected to the main tree.

This substantially changes the semantics of the map primitive. Previously, `map` conveyed privileges only on mappings that existed at the time when `map` was invoked. With the above change, however, map effectively 'links' two regions in the sender's and receiver's virtual address spaces; therefore it can also convey privileges on mappings the sender acquires *after* the mapping was established. Moreover, this does not only affect the direct receiver of the mapping, but all transitively derived mappings as well. See Section 4.7.3 for further details.

5.2.3 Extended pager privileges

In the original system, the influence a pager can exert on its clients is limited to the memory regions it provides. It can modify the contents of those regions or unmap them; however, it cannot directly influence regions provided by other pagers.

In our system, the kpager has access to map and cap nodes, UTCBs, and the node table. Therefore, it can freely edit the entire address spaces of its clients, e.g. by removing or permuting mappings. If it has access to other spaces, it can even install new mappings by forging a matching pair of map and cap nodes.

However, any of the above operations could obviously be performed by the client itself. Even in the original system, a pager that provides code segments to the client has similar powers, since it can modify the code and thus force the client to act as its proxy. Hence, kpagers require a trust level similar to that of a code pager.

[2]Due to the subtree property (Section 4.7.3), access to map or cap nodes in other address spaces is *not* required

5.2.4 Role of σ_1

The original L4 model includes a trusted kernel pager, σ_1, that is intended to provide orthogonal persistence [33, 48]. This pager has read/write access to the KTCBs, which it can use to obtain consistent snapshots of the entire system.

In our system, σ_1 is obsolete as soon as we succeed in exporting the entire contents of the KTCB to user level. Its role can be taken over by ordinary pagers, who can implement orthogonal persistence for their own subsystems. This scheme is considerably more flexible, since it is not centralized and the respective pagers do not need to be trusted by the kernel.

5.3 Lessons learned

The L4 Version 4 API in general, and its virtual memory model in particular, have turned out to be an excellent platform on which to implement kernel memory management. Yet, our work with the API has sparked some ideas for further improvement.

5.3.1 No global identifiers

On the one hand, some resources in L4 are fully virtualized. One example is memory, where mapping can be used to associate a virtual memory address with almost any physical memory address. On the other hand, however, some resources are global to the entire system, notably thread identifiers and priorities. ThreadIDs, for example, are valid for every principal in the system.

In our experience, global identifiers are a major obstacle when exporting kernel state to user level because they are easy to guess and hard to validate.

For example, a thread can easily guess a higher priority than its own (a lower integer number) and, by colluding with its pager, it can install this priority in its own TCB. Similarly, it can guess and install a larger time slice (a higher integer number). Because these attributes are not conveyed hierarchically like memory access privileges, they cannot be validated easily. This is the reason why we were unable to export the full content of the KTCB in our experiment.

From personal communication, we know that a revised naming scheme for threads is already being developed in the L4 community. We hope that, additionally, the scheduling model will be virtualized or replaced.

5.3.2 Address spaces as first-class objects

In the current version of the API, address spaces are, in a way, second-class citizens. Since they have visible properties (the KIP and UTCB area, which can be read and written with the `SpaceControl` system call), they clearly constitute a core abstraction. Yet, they do not have proper names; they are named indirectly by the ID of an arbitrary thread that is executing in the designated space.

This naming scheme is awkward for user-level applications since they must explicitly ensure that their names remain valid. Otherwise, the thread whose ID is used as a name might accidentally be migrated out of the designated space, and further system calls using that name might have unforeseen consequences.

Moreover, exporting this kind of name to user level can cause consistency problems. In our system, address spaces are used as principals, so there is often a need for naming a particular principal, e.g. in map and cap nodes. However, if a thread migrates while its threadID is used as principal identifier in some external state, that state cannot be re-imported into the kernel. Due to validation, this is not a security problem; however, the corresponding kernel state is lost.

These problems can easily be solved by promoting address spaces to first-class abstractions, and by giving them proper names.

5.3.3 Symmetric API

Some of the system calls in the current API are *asymmetric* because they contain special cases for some threads or tasks:

- The privileged system calls `SpaceControl` and `ThreadControl` can only be invoked by a few special tasks[3].
- The mapping feature works only if the recipient of the mapping resides in a different address space.

Both asymmetries have turned out to be major obstacles for application designers. The mapping asymmetry, for example, has forced L^4Linux developers to include the infamous *ping-pong* task. The sole purpose of this task is to reflect any mapping it receives; this is necessary to implement the `vm_remap` primitive, which relocates a memory region within an address space.

However, these restrictions exist mainly in order to prevent malicious tasks from consuming large amounts of kernel memory[4]. With our scheme, however,

[3]The use of privileged system calls is restricted to the initial servers, i.e. the ones created by the kernel at boot time (σ_0, σ_1 and the root server).

[4]The other reason why these restrictions exist is that the global threadID namespace must be managed by a trusted entity. If global IDs are removed, this becomes a nonproblem.

it is possible to account kernel memory usage to individual tasks; hence there is no reason any more why ordinary tasks should be barred from creating threads or mappings as they please. The above two asymmetries can thus be removed.

Chapter 6

Implementation

We have implemented the scheme described in the previous two chapters in an experimental microkernel, L4/Strawberry. The following chapter describes details of our implementation and reports our experiences. Also, it contains a short performance evaluation.

6.1 Goals

Like many operating system kernels, existing L4 implementations implicitly assume that kernel metadata is available when it is accessed. In our scheme, however, all variable-size kernel metadata is pageable, and the kernel must be prepared to handle a fault on virtually every metadata access. We believed this would require nontrivial changes to the internal design of the kernel. Therefore, we decided to validate our approach by attempting a 'real' implementation. Our objectives for this experiment were the following:

- Establish that the scheme works in a realistic setting. Our goal was to be able to run a large application on top of the experimental kernel.
- Explore implications on kernel design. Our goal was to compare our experimental kernel to existing implementations, and to identify any secondary changes that our scheme might require.
- Evaluate the effect on performance. Our goal was to keep the additional overhead on important kernel primitives below 5%, and to determine the effective cost for a simple user-level allocation policy, e.g. Quota.

We did *not* investigate which policies are best suited for managing kernel memory. Virtual memory management is a well-understood problem, although the issues are slightly more complicated for kernel memory because of the inter-resource dependencies and the higher preemption cost; we consider this future work. We also did *not* demonstrate that our scheme can be used to prevent Denial-of-Service attacks, or to enforce subsystem isolation. Related work [49] has shown that this is feasible with relatively simple policies like Quota, which can obviously be implemented in our system.

6.2 L4/Strawberry

At the time we started our experiment, an implementation of the L4 Version 4 API was already being developed in our group. Our original plan was to adapt this kernel to support our modified API (Section 5.2). However, we quickly found that this was infeasible. The existing kernel allocated its metadata from an in-kernel memory pool, and it was assumed throughout the code that this metadata could be accessed at any time, and without page faults. To change this behavior, we would have had to identify all of these assumptions and reimplement most of the corresponding code. Since microkernels are generally small, we decided that it would probably be easier to start from scratch.

Our new kernel is called L4/Strawberry[1]. It implements the basic L4 Version 4 API for the Intel IA-32 architecture and is binary compatible with the existing implementation. Thus, we were able to reuse most of the framework (bootloader, IDL compiler, initial servers) and run existing applications with only small modifications. Advanced features like SMP or Small Address Spaces are not supported.

The kernel consists of 5.860 lines of C code, 37 lines of IA-32 assembler code, and another 2.233 lines of C code for an optional kernel debugger. It is stable and complete enough to run L^{4}Linux [22], a variant of Linux 2.4 that runs as a user-level application on top of L4 microkernels. Since the original L^{4}Linux does not support resource faults, we had to modify its internal pagers (a 119 lines patch). Afterwards, we were able to boot a standard Debian distribution.

The kernel sources have been released under the GNU General Public License; they are available for download at `http://www.l4ka.org/`.

[1]L4 implementations are traditionally named after nuts. Many people have asked why we did not use this scheme for our kernel. In fact, however, the red fruit grown by the cultivated strawberry (*fragaria* × *ananassa*) is technically an enlarged pulpy receptacle; the actual fruit are the tiny *achenes*, each of which houses a single seed. Hence, strawberries are nuts and not berries – at least in the botanical sense.

6.3 Kernel design

In order to support pageable kernel metadata, we had to make some design choices that differ substantially from the classic L4 design as described in [34].

6.3.1 Continuations

The probably most striking difference between our kernel and existing implementations lies in the execution model, i.e. in the way blocking and preemption are handled. There have been two contrasting approaches to this problem: Many monolithic kernels such as BSD, Linux, and Windows NT implement the *process model*, where each thread has its own kernel stack; when the thread is not running, most of its state is implicitly encoded in this stack. In the *interrupt model*, which has been used e.g. in V [8], QNX [21], and Fluke [16], the kernel uses only one kernel stack per processor. Threads are required to record their state in an explicit kernel object, a *continuation*, before blocking.

Traditionally, L4 implementations have used the process model, i.e. per-thread kernel stacks. However, as described in Section 4.7.5, the contents of this kernel stack are difficult to export to user level. Therefore we decided to use continuations, and to build an interrupt-style kernel.

For the kernel designer, the interrupt model has two major disadvantages. First, it requires more programming effort because in contrast to the process model, a context switch is not equivalent to a simple stack switch; instead, explicit code is required to transfer state to and from the continuation. Second, it imposes constraints on when a kernel-level thread can block. For example, care must be taken that threads cannot be preempted by unexpected events, e.g. by a hardware interrupt or by taking a page fault while in the kernel.

However, we were surprised to find that the interrupt model also has considerable advantages:

- **It requires less kernel memory**, since the continuations are much smaller than kernel stacks. Process-style L4 kernels for the IA-32 typically use 1kB TCBs, while our kernel only needs 176 bytes.

- **It reduces the stack-related cache and TLB working set** of the kernel. This is especially important since stack content is frequently modified and must be written back to main memory afterwards, even though it may already be obsolete. Also, the single per-processor stack is more likely to remain in the cache between subsequent kernel invocations.

- **It simplifies thread management**, since all of the thread's state is explicit. There is no need to examine its stack, whose layout may change between platforms, compiler versions, or even different kernel configurations.

- **It enables further optimization**, since a thread can, by inspecting the continuation, easily find out what another thread will do when activated. Thus, it is possible in many cases to avoid using the continuation altogether.

Our findings are similar to the ones reported by Draves et al. [13]. By changing the execution model in Mach 3.0 from process-style to a hybrid model, they were able to speed up cross-domain RPC by 14% and to reduce the kernel space per thread by 85%. Certain operations even ran 60% faster.

6.3.2 Reference checking

Because our kernel is purely interrupt-style, we must avoid page faults in the kernel. Fortunately, the L4 API does not require the kernel to touch any user-backed objects[2] except during IPC, and even then only while copying strings. Thus, page faults can only happen a) during IPC and b) when first accessing an instance of kernel metadata.

Our kernel avoids the first type of page fault – during IPC string transfer – by eagerly checking how much of the string and the corresponding buffer is currently mapped. This part is then safe to copy; afterwards, a page fault message is synthesized if necessary. Although this requires the kernel to parse the page tables, the performance penalty is small because the physical addresses become available as a side effect. The kernel can use these addresses to bypass the traditional Copy Window technique [32] and copy the string directly using physical addresses.

The second type of page fault can be prevented by checking references before using them. References to non-existent or preempted resources have a special value that can be easily detected. When this value is encountered, the kernel leaves the fast path, saves a continuation, and synthesizes a resource fault on behalf of the corresponding principal.

The performance penalty incurred by reference checking is usually small. However, it can be considerable where a short, frequent operation requires many of those checks. We optimize these operations for the common case (all resources available) by adding a *hazard field* to the TCB. This field contains one bit for every resource, which is set when the resource is not available. On the fast path, the kernel checks the hazard field for the expected value (zero) and branches to the slow path if the check fails. With this optimization, only one simple check needs to be added to the IPC fast path.

[2]Although UTCBs are fully accessible from user space, they are actually kernel objects.

6.3.3 Preemption points

The interrupt model also complicates the handling of hardware interrupts. If an interruption occurs at a point where no consistent state is available, the kernel must perform recovery, e.g. by rolling the thread back or forward to the nearest continuation.

We avoid this problem by disabling interrupts in kernel mode. Instead, we add *preemption points* to potentially long-running operations such as `Unmap` and `IPC`. At each preemption point, the kernel checks whether an interrupt has occurred; if necessary, it writes a continuation and is then ready to perform a clean switch to the interrupt handler.

This method obviously increases interrupt latency. However, the latency is bounded by the maximum distance between adjacent preemption points and can thus be reduced by placing the preemption points densely and uniformly.

6.3.4 Physical mapping

In order to store metadata in memory frames supplied from user level, the kernel must have the frames mapped somewhere in its virtual address space. However, if these mappings are established separately for each 4kB frame, the kernel needs additional page tables. These meta-page-tables constitute yet another kernel resource that must be managed and allocated from the user-level resource manager.

Our experimental kernel avoids this problem by accessing kernel resources by their physical address. For this purpose, it keeps an internal linear mapping of all physical memory. Thus, all memory frames that can potentially be supplied as kernel resources are accessible *a priori*.

This approach has two advantages: First, the internal mapping can be composed of 4MB superpages and thus avoids separate TLB entries for each kernel resource, and second, the mapping is also useful during IPC string transfer (see Section 6.3.2). The disadvantage is that it limits the amount of physical memory that can be used (1GB in our experimental kernel) because the physical mapping must fit entirely into the virtual address space. Obviously, this problem disappears in 64bit systems.

6.4 Performance

In this section, we present a short performance evaluation of our experimental microkernel. Specifically, we examine kernel memory usage, the overhead for a simple user-level allocation policy, and the impact our scheme has on the IPC path, a critical fast path within the kernel.

For our benchmarks, we used a dual Pentium II/400 system with 192 MB of main memory and a current version of L4/Strawberry. For all timing benchmarks, the kernel debugger and all runtime checks were disabled. Note that the experimental microkernel does not have multiprocessor support, and thus the second CPU remained unused.

In addition to our own benchmarking applications, we used a development version of L^{4}Linux 2.4.20, which we had to modify slightly in order to add resource fault handling. No other modifications were made.

6.4.1 Kernel memory usage

In order to determine the amount of kernel memory used by typical applications, we booted a Debian distribution on top of L^{4}Linux. After opening an `emacs` session and starting a compile job, we entered the L4/Strawberry kernel debugger to obtain a snapshot of the current memory usage (see Figure 6.1).

For every address space in the system, we determined the number of threads it contains, the amount of user memory mapped to it, and the size of the corresponding metadata. Since the metadata is allocated with frame granularity, the table shows the number of 4kB frames for each metadata type (page tables, node tables, nodes in the mapping database, and UTCBs). For comparison, it also gives the total size of the allocation in kilobytes.

We found that a typical application consumes approximately 48kB of kernel memory, which is nonnegligible when compared to typical resident set sizes of around 100kB. Surprisingly, however, the numbers do not vary much between small and large applications. This suggests a high degree of internal fragmentation, and a closer inspection is warranted.

The minimum kernel memory consumption of an L^{4}Linux task is as follows:

1. One page directory (P) and the corresponding node directory (N)
2. Four page tables (P), one each for code region, library region, UTCB area and KIP area, and an equal amount of node tables (N)
3. One page for map and cap nodes (M)
4. One page for user TCBs (U)

The page and node tables are sparsely populated. For a resident set of 100kB, only 25 of the 4,096 entries (0.6%) are used, and thus the resident set can be increased drastically without a need for additional tables. Similarly, a UTCB frame can accommodate eight threads, and one frame in the mapping database

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

Figure 6.1: Kernel memory consumption 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.

can hold up to 256 map or cap nodes, which is sufficient to map a 1MB resident set.

However, little of this fragmentation is actually caused by our memory management scheme. In the case of page and node tables, the layout is fixed by the IA-32 hardware, and the fragmentation in the UTCB pages is inherent because the API does not allow UTCBs to be visible in other address spaces. Thus, the effective overhead of our scheme amounts to only 1.5 frames (6kB) per address space. One frame is needed for the node directory, and half a frame is typically wasted because map and cap nodes are stored in private memory rather than in a central pool.

We conclude that a) there is enough per-task metadata to justify the effort of controlling its allocation, and that b) the spatial overhead induced by our scheme is sufficiently low.

6.4.2 Policy overhead

In order to determine the temporal overhead for a simple user-level allocation policy, we first measured the time required to handle a resource fault. To this end, we modified our kernel to support an optional in-kernel memory pool. When this pool is in use, the kernel acts like a conventional L4 kernel and does not generate any resource faults.

Then we ran a simple test application that caused a page fault in a previously untouched memory region. This memory region was chosen so that the corresponding page and node table could not be present and had to be requested from the manager, which implemented a simple Quota policy. Thus, with resource faults enabled, the following happened:

1. The task requested a new node table
2. The task requested the 4kB page that was touched
3. The task requested a new page table

With the in-kernel memory pool, only the second fault was generated because the translation tables were allocated internally.

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

Figure 6.2: Cycles required to handle a triple resource fault.

We then used the performance counters of the Pentium II to measure the cycles required in both cases (Figure 6.2). The difference of approximately 3,363

	Pentium II/400		Pentium III/800	
	Pistachio	Strawberry	Pistachio	Strawberry
Intra address space	184 cycles	148 cycles	181 cycles	145 cycles
Cross address space	426 cycles	328 cycles	363 cycles	328 cycles

Figure 6.3: IPC cost on two different machines.

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 resource fault on this machine.

In the previous section, we demonstrated that a typical L^{4}Linux task consumes around 48kB of metadata. This is equivalent to 12 frames. We estimate that requesting these frames from a user-level manager causes an additional one-time overhead of $12 \cdot 1,700 = 20,400$ cycles or 51μs, which we consider acceptable, especially given that our microkernel is completely unoptimized.

6.4.3 Fast path overhead

The use of our scheme causes nontrivial changes in the kernel. Specifically, additional checks may have to be added to ensure that metadata is available, and important data structures may have to be changed. These changes come at a certain cost in terms of performance, particularly if they affect a fast path inside the kernel. Thus, one important goal of our implementation effort was to demonstrate that this cost is reasonable.

The most important fast path in an L4 microkernel is the IPC system call, which has traditionally been used as a performance metric in the L4 community [22, 32, 38]. Hence, we decided to use this system call in our evaluation.

For measurement, we used the canonical `pingpong` benchmark, which has been used with most of the recent L4 implementations. The benchmark creates two threads that continuously send tiny IPC messages back and forth, using the simplest possible set of parameters. With the CPU cycle counter, the time for a series of eight round trips is measured; the result is used to estimate the time required for a single IPC.

We had to make small modifications to the benchmark application in order to add support for resource faults; none of them affected the actual measurement process. Afterwards, we ran identical binaries on both L4/Strawberry (our experimental kernel) and L4/Pistachio, a previous implementation of the L4 Version 4 API. To preclude hardware effects, we used two different machines: The dual Pentium II/400 mentioned above and a dual Pentium III/800.

Figure 6.3 shows the results. Surprisingly, L4/Strawberry performed about 20% *better* than L4/Pistachio in most cases.

While these results are certainly encouraging, they must be interpreted with care. They obviously *do not* indicate that kernels with resource management are generally faster than those without, since our experimental kernel undoubtedly contains additional checks that exist for the sole purpose of resource management. We suspect that the difference is due to structural differences between the two kernels, especially with respect to the execution model; this would confirm the results in [13], but contradict findings in [32]. This issue needs further investigation.

However, these results *do* indicate that the overhead caused by resource management cannot be dramatic. A more exact, quantitative analysis could be done with a microkernel that has resource management as a configuration option; however, building such a kernel is not a goal of this work and is certainly beyond its scope.

Chapter 7

Conclusions and Future Work

The objective of this thesis is to address limitations and deficiencies in existing schemes for kernel memory management. The thesis argues that these limitations can be overcome by implementing the management policy outside of the kernel. For this purpose, it introduces a kernel mechanism that securely exports control over kernel memory resources to user level.

7.1 Limitations in existing schemes

Previous schemes for kernel memory management suffer from at least one of the following four weaknesses:

- They do not offer predictable control over *all* variable-size kernel state,
- They do not fully isolate subsystems and thus cannot be used to prevent Denial-of-Service attacks,
- They do not allow an allocation of kernel memory to be decreased gracefully and without damaging the principal, or
- They do not permit untrusted subsystems to profit from their own custom management policy.

7.2 Contributions of this thesis

This thesis introduces a kernel mechanism that addresses all of the above limitations. By applying the concept of paged virtual memory to the kernel, we obtain a scheme in which user-level applications can page their own metadata. The metadata is logically part of the application's address space and can be backed with ordinary memory. All of this memory can be preempted at any time; the corresponding metadata is exported to user level in a safe representation and can later be re-imported by the kernel.

7.3 Future Work

The work described in this thesis can serve as a starting point from which new management policies for kernel memory can be explored. We believe that many existing virtual memory management policies, such as LRU or Working Set, can easily be adapted to kernel memory, although the higher preemption cost and the inter-resource dependencies must be considered carefully and may require changes in some cases.

Our scheme gives user-level applications an unprecedented level of control over kernel memory, which could be exploited in many ways. For example, it allows applications to control the physical placement of kernel metadata. This could be used to implement cache coloring [28] and thus increase predictability in real-time applications, or to improve locality in SMP systems.

In our system, managers have access to the entire state of the subsystems they control, including kernel metadata. This could be used to implement persistence at user level. Since managers do not have to be trusted by the kernel, the resulting system would be very flexible, allowing different persistence policies to co-exist and new policies to be introduced dynamically.

Finally, the fact that our scheme allows kernel metadata to be edited from user level could be used to simplify and optimize the kernel API. For example, system calls that only serve to manipulate kernel state could be omitted entirely; complex operations, such as loading a new application, might be performed at user level and without invoking any kernel primitives. This could increase system efficiency and further decrease the complexity of the kernel.

Bibliography

[1] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 96–107. ACM Press, Apr 1991.

[2] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The SawMill framework for virtual memory diversity. In *Proceedings of the sixth Australasian Computer Systems Architecture Conference*, pages 3–10. IEEE Computer Society Press, Jan 2001.

[3] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 45–58. USENIX Association, Feb 1999.

[4] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN: An extensible microkernel for application-specific operating system services. In *Proceedings of the sixth ACM SIGOPS European workshop*, pages 68–71. ACM Press, 1994.

[5] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles*, pages 267–284. ACM Press, Dec 1995.

[6] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer Technical Conference*, pages 87–98, Jun 1994.

[7] Christian Ceelen. Implementation of an orthogonally persistent L4 μ-kernel based system. Study thesis, University of Karlsruhe, Feb 2002.

[8] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988. ISSN 0001-0782.

[9] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the sixth ACM SIGOPS European workshop*, pages 88–91. ACM Press, 1994.

[10] David R. Cheriton and Kenneth J. Duda. Logged virtual memory. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles*, pages 26–38. ACM Press, Dec 1995.

[11] Alan Dearle, Rex di Bona, James Farrow, Frans Hensken, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.

[12] Alan Dearle and David Hulse. Operating system support for persistent systems: Past, present, future. *Software - Practice and Experience*, 30(4):295–324, 2000.

[13] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the thirteenth ACM Symposium on Operating systems principles*, pages 122–136. ACM Press, 1991.

[14] 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.

[15] Dawson R. Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM: Application-level virtual memory. In *Proceedings of the fifth Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995.

[16] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 101–115. USENIX Association, 1999.

[17] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathan E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, Sep 2000.

[18] Andreas Haeberlen. User level management of L4 kernel memory. In *Proceedings of the Second International Workshop on Microkernel-Based Systems*, Lake Louise, Canada, Oct 2001.

[19] Steven M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 73–86. USENIX Association, Feb 1999.

[20] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 187–197. ACM Press, Oct 1992.

[21] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Apr 1992.

[22] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77. ACM Press, Oct 1997.

[23] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 1.* Order No. 245470-007.

[24] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *Proceedings of the seventh Workshop on Hot Topics in Operating Systems*, Mar 1999.

[25] Trent Jaeger, Jonathan E. Tidswell, Alain Gefflaut, Yoonho Park, Jochen Liedtke, and Kevin Elphinstone. Synchronous IPC over transparent monitors. In *9th SIGOPS European Workshop*, Sep 2000.

[26] The K42 Team. Memory management in K42. White paper, IBM T.J. Watson Research Center, Aug 2002. URL `http://www.research.ibm.com/K42/`.

[27] John P. Kearns and Samuel DeFazio. Diversity in database reference behaviour. *Performance Evaluation Review*, 17(1):11–19, May 1989.

[28] Richard E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, Nov 1992.

[29] 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.

[30] The L4Ka Team. L4 experimental kernel reference manual, version X.2. URL `http://www.l4ka.org`. Feb 2002.

[31] Jochen Liedtke. Clans & chiefs. *Architektur von Rechensystemen*, Mar 1992.

[32] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the fourteenth ACM Symposium on Operating systems principles*, pages 175–188. ACM Press, Dec 1993.

[33] Jochen Liedtke. A persistent system in real use: Experiences of the first 13 years. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Dec 1993.

[34] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles*. ACM Press, Dec 1995.

[35] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9), Sep 1996.

[36] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proceedings of the sixth Workshop on Hot Topics in Operating Systems*, May 1997.

[37] Jochen Liedtke, Volkmar Uhlig, Kevin Elphinstone, Trent Jaeger, and Yoonho Park. How to schedule unlimited memory pinning of untrusted processes or provisional ideas about service-neutrality. In *Proceedings of the seventh Workshop on Hot Topics in Operating Systems*, pages 153–159, Mar 1999.

[38] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proceedings of the eighth Workshop on Hot Topics in Operating Systems*, pages 15–18, May 2001.

[39] 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.

[40] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the second symposium on Operating systems design and implementation*, pages 153–167. ACM Press, Oct 1996.

[41] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, Aug 1992.

[42] Richard Rashid, Avadis Tevavian Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8), Aug 1988.

[43] John Reumann, Ashish Mehra, Kang G. Shin, and Dilip Kandlur. Virtual services: A new abstraction for server consolidation. In *Proceedings of the 2000 USENIX Annual Technical Conference*, Jun 2000.

[44] Rik van Riel. Page replacement in Linux 2.4 memory management. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Jun 2001.

[45] Mahadev Satyanarayanan, Harry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1), Feb 1994.

[46] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. State caching in the EROS kernel. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, pages 88–100, 1996.

[47] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the seventeenth ACM Symposium on Operating systems principles*, pages 170–185. ACM Press, Dec 1999.

[48] Espen Skoglund, Christian Ceelen, and Jochen Liedtke. Transparent orthogonal checkpointing through user-level pagers. In *Proceedings of the 9th International Workshop on Persistent Object Systems (POS9)*, Sep 2000.

[49] Oliver Spatscheck and Larry L. Peterson. Defending against denial of service attacks in Scout. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 59–72. USENIX Association, Feb 1999.

[50] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, Jul 1981.

[51] Sun Microsystems Inc. Solaris resource manager 1.0. White paper, Palo Alto, California. URL `http://www.sun.com/software/white-papers/wp-srm/`.

[52] Cristan Szmajda. Calypso: A portable translation layer. In *2nd International Workshop on Microkernel Based Systems*, Sep 2001.

[53] Patrick Tullmann, Jay Lepreau, Bryan Ford, and Mike Hibler. User-level checkpointing through exportable kernel state. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, Seattle, WA, Oct 1996. IEEE.