Diplomarbeit

Design and Implementation of the L4.sec Microkernel for Shared-Memory Multiprocessors

Torsten Frenzel
frenzel@os.inf.tu-dresden.de

August 4, 2006

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Bernhard Kauer

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den August 4, 2006

Torsten Frenzel

Acknowledgments

First, I would like to thank Prof. Hermann Härtig for the opportunity to work on this project in the Operating Systems Group at the TU Dresden. My special thanks go to my supervisor Bernhard Kauer for the long and clarifying design discussions and the time he spent for providing me with many insights about the internal concepts of L4.sec. Udo Steinberg and Micheal Peter deserve many thanks for the numerous inspiring discussions about various problems of multiprocessor operating systems. They focused my thoughts to find a way through the jungle. Furthermore I would like to thank Lenin Singaravelu, Björn Döbel, Marcus Völp and Udo Steinberg for proofreading chapters of this thesis. Their valuable comments helped me improve the style of this document tremendously. Finally, many thanks goes to my family for their constant support and their encouragement and to my friends, who helped me recreating after some stressful days.

Contents

List of Tables

List of Figures

1 Introduction

1.1 About this work

Microkernels have steadily evolved since early research prototypes. The result was a generation of fast and stable kernels, collectively called second-generation microkernels. The L4 microkernel is a member of this family [1]. Beginning with the L4v2 specification [11] as most widely used and implemented microkernel interface, other specifications were designed to resolve revealed shortcomings. The L4x0 interface [12] introduces local names for threads and the L4x2 interface [3] aims at a more hardware-independent interface. But despite this success, the important issues of communication control and security have only been partly solved.

Therefore in the last two years a new microkernel specification, called L4.sec [19], was designed and implemented [8] at the TU Dresden Operating Systems Research Group to address the problems of communication control and security. L4.sec tries to avoid the security weaknesses of the second-generation microkernels while trying to retain the key properties of microkernels, like low complexity and fast inter-process communication (IPC).

L4.sec is a member of the L4 microkernel family. This means that L4.sec preserves the L4 paradigm, which declares address spaces, threads and IPC as the minimal set of abstractions that a microkernel has to implement. Beside these well-known abstractions, L4.sec adds new concepts of capabilities, endpoints and kernel-memory objects (KMO).

All kernel objects in L4.sec are referenced by capabilities. Capabilities are local names for kernel objects and reside in the capability space of a task. The capability space is managed the same way as a memory space. Each capability in the capability space is associated with access rights for the kernel object. Capabilities can be mapped from one capability space to another and revoked, constructing a recursive capability space model.

Endpoints are communication channels for a set of senders and receivers. They can be created and their capabilities can be mapped by applications, like other kernel objects. An IPC does not address a thread directly instead an endpoints has to be referenced.

L4.sec represents kernel memory as KMOs. Applications have to provide the memory for kernel objects by specifying a KMO. On one hand, this allows them to manage the kernel memory. On the other hand, it is guaranteed that the kernel memory consumption does not exceed the limits of the application.

Together these new features are the basis to build a secure operating system on top of L4.sec.

The current design and implementation of L4.sec focusses on uniprocessor architectures. However, multiprocessor architectures become more and more available not only in server-side systems but also in client-side systems. Even desktop computers are being equipped with multiple processors. So support for multiprocessor architectures in operating systems becomes a standard feature.

This thesis focuses on the design of necessary mechanisms to support multiprocessor architectures in the L4.sec microkernel. I will discuss extensions of the existing L4.sec model and outline their advantages and disadvantages. The goal of this work is to find an L4.sec multiprocessor

[1] Another second generation microkernel is QNX [21]

model, which provides sufficient support to build scalable and flexible operating systems. To reach these goals, this work focuses on two points:

1. The design of an IPC primitive based on the concept of endpoints. The resulting design optimizes communication, which does not cross processor boundaries while not restricting communication between different processors.

2. The synchronization of kernel memory. It turned out that the non-type-stable kernel memory complicates synchronization and adds overhead. This thesis attempts to solve this problem with the help of read-copy update techniques without changing the handling of kernel memory.

To reach these goals, low-level synchronization primitives have to be designed properly. It turned out that optimized locks for specific kernel operations provide an adequate solution to reduce the synchronization overhead.

The design is implemented and evaluated in a prototype. I compare the performance of kernel operations with the uniprocessor kernel and examine the impact of the synchronization overhead.

1.2 About this document

This thesis is organized as follows. Chapter 2 introduces the underlying concepts of L4.sec: kernel objects and kernel operations. In this chapter, I also discuss the most relevant points about multiprocessor operating-system kernels and related work, like synchronization. In Chapter 3, I outline and discuss different designs of the communication model and the kernel-memory model in detail. The investigation of low-level aspects of kernel design leads to efficient synchronization primitives which allows an efficient implementation of the multiprocessor microkernel. Chapter 4 presents the prototype and interesting parts of the implementation. At the end of this work, I evaluate the performance of the new L4.sec multiprocessor kernel and verify the decisions made in Chapter 3. The thesis concludes with suggestions for future work.

2 Fundamentals and Related Work

This chapter introduces the concepts of L4.sec in more detail. I describe the model of the multiprocessor architecture on which the new L4.sec microkernel is based and outline relevant techniques for synchronization in multiprocessor systems.

2.1 L4.sec Concepts

In addition to the primitives of tasks, threads and address spaces known in L4, L4.sec adds capabilities, endpoints and kernel-memory objects to provide control of communication and kernel-memory usage and increase the level of security.

2.1.1 Capabilities and Capability Spaces

Capabilities refer to kernel objects, like tasks, threads and endpoints and have permissions attached to them. Capabilities reside in a task-local name space, called capability space, which is responsible for the translation from a local name to a capability. The concept of the capability space has analogies to a virtual address space. A virtual address is translated to a page table entry to gain access to a resource.

When an application references a kernel object, it has to provide a local name for the object. The kernel looks up the capability space of the application to find the kernel object. If the capability does not exist or the check on access rights fails, the operation is aborted and a capability fault is returned to the application.

Creating a new kernel object or mapping a capability from one capability space to another inserts a new capability in the capability space, which can be referenced with a local name. The unmap operation revokes capabilities. The kernel object gets implicitly destroyed, if the last capability referencing the kernel object is removed.

2.1.2 Endpoints and Communication Control

Endpoints provide a communication channel for threads. A thread can send a message to an endpoint or receive a message from an endpoint. For example, if two threads have the right to receive a message from an endpoint, and third thread has the right to send to an endpoint, the sending thread can not determine which of the two receiving threads received the message.

Endpoints are adequate abstractions for the implementation of multi-threaded server systems, where the server has a set of worker threads which are bound to a service endpoint. Every client that is allowed to access the server has mapped the service endpoint with send permissions in its capability space. The server can dynamically adapt the number of service threads to the number of clients.

But endpoints also have shortcomings, which make them inappropriate for some use cases. For example in the implementation of a stateful protocol it is necessary to provide some way of sender identification because the server needs to track the state of the sender. Another problem is that IPC-operations with call semantics can not be implemented using a single endpoint. When a client requests a service from the server and has to wait for a reply, it has to use a second

dedicated endpoint. Thus every client thread has to provide its own endpoint to accept a reply from the server.

2.1.3 Kernel Memory Management

Kernel resources typically need kernel memory to store their data structures. A kernel can manage its resources with and without charging the user application that caused the resource request. For example, in many operating systems the kernel uses a reserved, fix-sized memory area where it allocates memory for kernel objects. In situations when the kernel memory is exhausted, the kernel suffers an out-of-memory error and the operation has to be aborted. Malicious applications can exploit this weakness to launch an exhaustion denial-of-service attack by creating kernel objects continuously to stall the kernel. To protect against this attack, quotas for user-level applications can be introduced, which restrict the number of kernel objects for an application.

L4.sec exposes kernel-memory management to the application. All memory that the kernel needs for creating kernel objects, has to be provided by user-level applications. Every application therefore has to choose, if it uses its memory for user-level objects or kernel objects. The process of changing a memory page from user-level to kernel-level is called conversion. The reverse process is called reconversion and implicitly destroys all kernel objects allocated in this page.

The above solution has several advantages. On one hand, user-level applications can manage their resources in the most effective way. On the other hand, if there is not sufficient kernel memory available to execute the operation, the application is responsible for resolving this conflict. Therefore the kernel does not need to implement a built-in strategy to resolve such situations and can not be compromised.

In its current implementation L4.sec has no fixed layout for the kernel address space and there are no static regions for one type of kernel objects. For example a virtual address may refer to data of a task but in the future the same address may refer to data of an endpoint. The type of kernel memory can change dynamically in time when creating and deleting kernel objects. Thus the L4.sec microkernel implements non-type-stable kernel memory. This property of kernel memory has negative implications for synchronization in a multiprocessor model. For example it rules out common lock-free synchronization approaches.

2.1.4 Kernel Objects

L4.sec classifies kernel objects into named and unnamed kernel objects. Named kernel are explicitly created by applications and are referenced by capabilities. The following list describes the internal structure of the relevant named kernel objects.

Tasks

Tasks are resource containers and protection domains. Every task has one memory space and one capability space. Threads can be bound to a task, which means that the thread runs in the address space of the task and has access to all resources owned by the task.

Threads

Threads represent activities and run in the context of a task. A thread executes either a user-level program or a kernel operation requested by the application. During a kernel operation a thread can be associated with another thread. The most obvious example is the IPC-operation, where threads communicate with each other.

The state that characterizes a thread, comprises a state field, processor registers and virtual registers. The state field holds general information, for example if the thread is running or halted. The set of processor registers, like the instruction pointer, the stack pointer and general purpose registers, is defined by the hardware architecture. Virtual registers are caches, which hold information that are needed to execute a kernel operation, for example a timeout for an IPC-operation.

Endpoints

Endpoints are abstractions of communication channels. They contain a wait queue used for IPC-operations to block waiting senders and receivers. In the wait queue either senders or receivers are blocked but not both. An endpoint capability can be mapped into multiple capability spaces allowing threads of different tasks to communicate.

2.1.5 Kernel Operations

This section gives an overview of the most important operations of L4.sec. The kernel operations can be arranged into two groups. The IPC-operation, cancel-operation, exregs-operation and destroy-operation [1] are considered thread-related operations, because they manipulate the thread state. The map-operation and the unmap-operation, which manipulate the mapping database (MDB), are called mapping-database-related operations.

IPC-operation

The IPC-operation transfers a number of values from a sender to a receiver. It consists of three phases. In the first phase both partners have to arrive at the endpoint and wait for a rendezvous. If the sender arrives before the receiver, it has to block and vice versa. After this phase, both partners are in an appropriate state for arranging the data transfer in the next phase. This implies copying the values from the sender to the receiver. After the second phase has completed, the partners do the post-processing which includes error checking, state correction and waking up the passive partner to finish the IPC-operation.

cancel-operation

A thread, which executes an IPC-operation and is waiting in the endpoint for a partner, can be canceled, with the cancel-operation.

exregs-operation

The exregs-operation [2] is considered an explicit synchronization operation on a target thread to read or change the thread state. The operation is split into three phases: a halt phase, where the thread is stopped (i.e. not running anymore), a work phase, where data are exchanged and a resume phase to restart the thread. The resume phase is optional and can be omitted, if the thread should not be restarted.

[1] The destroy-operation is only considered, if a thread is destroyed.
[2] Exregs is an abbreviation for exchange registers.

create-operation

Creating new kernel objects is done implicitly for unnamed kernel objects and explicitly with the create-operation for named kernel objects. The kernel uses a simple next-fit allocation strategy to find available kernel memory in the provided capability set. It also allocates a root-mapping node for this object and inserts a capability in the capability space of the creating task.

A special case of the create-operation is the conversion of a user-level memory page to a kernel memory page. As a result of a successful conversion a kernel-memory object (KMO) is instantiated, which describes the kernel memory region. The KMO is a named kernel object and therefore is accessible through a capability in the capability space. The first KMO for the initial task is instantiated by the kernel.

map-operation

The right to access a kernel object or a memory page can be transfered from one space to another with the map-operation. A map-operation creates a new mapping node and inserts a capability in the target space. All mapping nodes of one kernel object or memory page assemble the mapping tree of this object. The mapping database consists of all kernel objects and memory pages. It has for kernel objects a more complex structure than for memory pages.

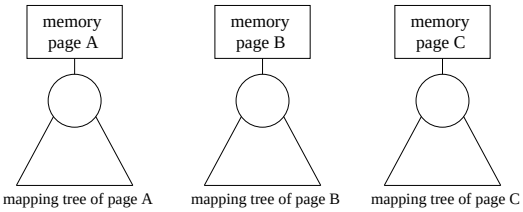

Figure 2.1: Example of a mapping database for memory pages

Memory pages cannot be created or destroyed and no relationships exist between them, which leads to independent mapping trees (Figure 2.1). Therefore the structure of the mapping database is a set of mapping trees.

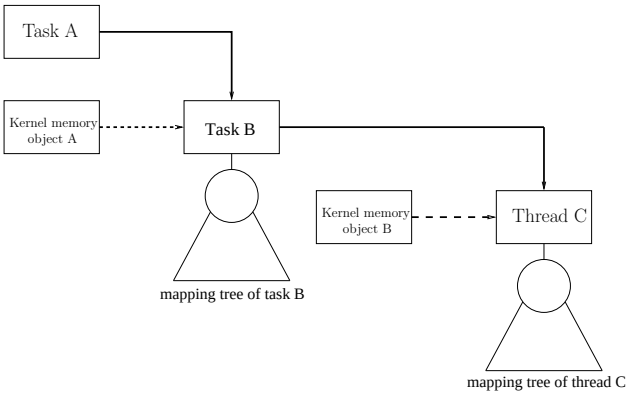

Figure 2.2: Example of a mapping database for kernel objects

As tasks provide the kernel memory for kernel objects, especially for other tasks, every kernel object has a resource dependency. The mapping trees of kernel objects are linked together by

these resource dependencies. In the example (Figure 2.2) the task A has created a task B and the memory resource is provided by KMO A. Task B can itself create other kernel objects, like thread C. This example demonstrates that the mapping database for kernel objects has a directed graph structure.

unmap-operation

An unmap-operation revokes rights from capabilities in a subtree of the mapping database. The capability is removed if no rights remain. When the whole mapping tree of kernel object is deleted, the kernel object is destroyed. Destroying one kernel object can result in the destruction of other kernel objects. If in the previous example task B is destroyed, thread C is also destroyed. The duration of an unmap-operation is only bound by the amount of available resources for building up the mapping tree.

destroy-operation

The L4.sec kernel interface does not provide a destroy-operation. Instead a kernel object is destroyed implicitly on the following two events:

- the root-mapping node of the kernel object is deleted
- the kernel-memory page in which the kernel object is allocated is reconverted

After completion of the destroy-operation the object cannot be accessed anymore and no other kernel operation is allowed to have a dangling reference to this object. The freed kernel memory can be reused to create new kernel objects.

The previous example illustrates that the destroy-operation is always executed in the context of an unmap-operation. Thus during an unmap-operation several objects can be destroyed. The unmap-operation invokes the destroy-operation when an object is deleted.

Unmapping a subtree of the mapping database is done in two steps. In the first phase the tree is traversed from the root node to the leaf nodes and a function is executed onto every object, to prepare it for the destroy-operation. In the second phase the tree is traversed in reverse order. In every step the mapping node is deleted; if the root node of an object is reached, the object is destroyed.

2.2 Multiprocessor Architectures

Multiprocessor systems can be classified by various aspects. For example they can be distinguished by the way the processors are connected to the memory [25]. In terms of memory-processor organization three main groups of architectures can be distinguished. In the following these approaches are described.

2.2.1 Shared-Memory Architectures

The main property of shared memory architectures is, that all processors in the system have access to the same memory, there is only one global address space. In such a system, communication and synchronization between the processors can be done via shared memory variables. An interconnection network connects the processors to the memory modules.

A big advantage of shared memory computers is, that programming a shared memory system is very convenient due to the fact that all data is accessible by all processors. There is no need to

copy data. However, it is very difficult to obtain high levels of parallelism with shared memory machines. This limitation stems from the fact, that a centralized memory and the interconnection network are both difficult to scale.

2.2.2 Distributed Memory Architectures

In case of a distributed memory architecture, each processor has its own, private memory. There is no common address space, i.e. the processors can only access their own memory. Communication and synchronization between the processors is done by exchanging messages over the interconnection network.

In contrary to a shared memory architecture, a distributed memory machine scales very well, since all processors have their own local memory, which means that there are no memory access conflicts. But scalability is limited by the capacity of the interconnection network. Using this architecture, massively parallel processors (MPP) can be built, with up to several hundred or even thousands of processors.

2.2.3 Distributed Shared-Memory Architectures

To combine the advantages of the architectures described above, ease of programming on the one hand, and high scalability on the other hand, a third kind of architecture has been established, called distributed shared memory architectures. Here, each processor has its own local memory, but contrary to the distributed memory architecture, all memory modules form one common address space, i.e., each memory cell has a system-wide unique address. In order to avoid the disadvantage of shared-memory computers, namely the low scalability, each processor uses a cache, which keeps the number of memory-access conflicts and the network contention low. However, the usage of caches introduces a number of problems, for example, data in the memory and the copies in the caches must be kept up-to-date. This problem is solved using sophisticated cache coherence and memory consistency protocols.

The multiprocessor design developed in this thesis is based on the shared memory architecture with local caches and a simple signaling service. The Intel IA32 architecture [1] and the Alpha architecture [2] are examples of such shared memory architectures.

2.3 Locking and Synchronization

Uniprocessor systems have only quasi-parallel execution, meaning exactly one thread is executing at a time. A thread which enters a critical section only has to prevent preemption to achieve mutual exclusion. In multiprocessor system synchronization is more complex as two threads may be executed in parallel on different processors and therefore preventing preemption is not sufficient to protect a critical section.

There are various synchronization primitives discussed in the literature. They are classified into lock-based synchronization primitives and lock-free synchronization primitives.

2.3.1 Lock-based synchronization

For operating systems the most relevant synchronization primitives are locks. The concept of a lock is classified into blocking locks and non-blocking locks. The following list discusses some locks appropriate for multiprocessor systems and their advantages and disadvantages.

1. Race-based Spin-locks

   Spin-locks are most widely used in shared-memory architectures [4]. All lock requesters spin on the same location and race for the lock. The advantage of spin locks is the low overhead for acquiring the lock in the non-contented case. Two major disadvantages are the usurpation of the processor and the lack of fairness [3]. Spin-locks are feasible for short critical sections.

2. Queue-based Spin-locks

   The class of queue based spin-locks provides better scalability if the lock is contented and ensures fairness, an example are MCS-locks [7]. Every thread, which tries to grab the lock enqueues itself into a wait queue and spins on a separate location. The lock holder releases the lock and frees the next lock requester in the wait queue.

3. Semaphore-locks

   Semaphore-locks avoid the blockade of the processor. If the lock requester cannot obtain the lock, it enqueues itself into a wait queue and blocks. When the lock is released by the lock holder, it also wakes up the first waiting thread in the wait queue. The advantage of blocking locks is that the CPU [4] is released voluntarily by the lock requester. But on the other hand the synchronization overhead compared to spin-locks is high because of the extra scheduling operation when a lock requester blocks. Thus semaphore-locks are appropriate for long critical sections.

4. Adaptive Locks

   Adaptive locks combine the advantages of spin-locks and semaphore-locks [13]. When a lock requester attempts to acquire a lock that is held by another one, it checks if the lock owner is active on another processor. If the lock owner is running, the lock requester spins. If the lock owner is blocked, the lock requester blocks as well. The goal of is strategy to avoid synchronization overhead of semaphore locks, because a running lock owner releases the lock as soon as possible and so spinning is faster and the expensive blocking operation is avoided.

5. Reactive Locks

   Reactive locks combine the advantages of racing-based spin-locks and queueing-based spin-locks [5]. Depending on the contention of the lock, the appropriate mechanism is chosen. If the lock has low contention the simple spin-lock strategy is used. If the contention increases, a queue-based spin-lock is used.

6. Reader-Writer Locks

   Reader-writer locks impose different lock semantics for readers and writers. As readers do not interfere with each other multiple readers can enter a critical section at the same time, but only one writer can change the data at a time. This yields improved performance for a system with many readers and few writers [17]. A reader-writer lock can prioritize the readers or the writers or handle both fair [9].

7. Read-Copy Update Synchronization

   Read-Copy Update (RCU) is a multiprocessor compliant synchronization mechanism and similar to a reader-writer lock [15]. The writer is responsible for all synchronization, while

[3] Ticket locks are a class of spin-locks that assure also fairness.
[4] CPU is used as a synonym for a processor

readers have no synchronization overhead. This is feasible for a system with many readers and few writers. RCU relies on out-of-place updates that can be performed in an atomic way and short read-side critical sections. Changing a data structure is done in two steps: First the new element is inserted and the old one is removed from the data structure. Before the old element can be deleted, the writer has to assure that no references point to this element. This is achieved by waiting, until all processors have passed a quiescent state. The quiescent state of a processor is defined as a state, where no thread is in a critical section and references to stale kernel objects do not exist. Thus after all processors have passed such a state no references to the old element exist anymore. This period is called grace period or epoch.

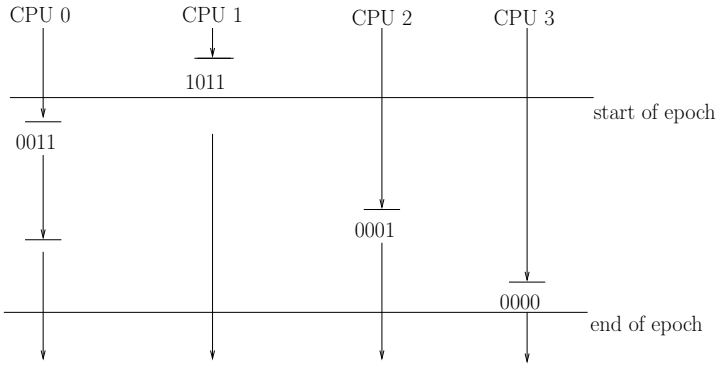

Figure 2.3: Measurement of on RCU epoch

An RCU epoch can be actively enforced or measured passively [16]. An example for measuring epochs is shown in Figure 2.3. CPU1 one starts the epoch by setting a global bitmask. Every CPU that passes though the quiescent state(, denoted by a horizontal line in the example), deletes the corresponding bit in the bitmask. After the last CPU has passed the quiescent state, the bitmask is zero and the epoch is over. Now all RCU requests, which arrived before the start of the epoch can be executed.

2.3.2 Lock-free Synchronization

As the name suggests lock-free synchronization uses no locks [24]. Critical sections have to be designed in such a way that they prepare their results out-of-place and then commit the new data using an atomic memory update. If a conflict is detected, the operation has to be restarted. Because there are no locks, lock-free synchronization avoids deadlocks and is multiprocessor safe. To the knowledge of the author all currently available lock-free algorithms assume type-stable kernel memory.

2.4 Multiprocessor Implementations of the L4 Microkernel

There are various implementations of the L4 microkernel with support for multiprocessors. For three L4 kernels the design goals are outlined and the achieved results are summarized. This discussion provides a foundation for deriving the design goals for the L4.sec multiprocessor model.

2.4.1 L4/Alpha Microkernel for the Alpha Architecture

The L4/Alpha microkernel was implemented for the Alpha processor family [20]. This kernel is designed to be scalable to a large number of processors, which is supported by keeping kernel data processor-local as much as possible, and by minimizing the used of cross-processor locking primitives and inter-processor messages. Thus the most important design goals of this kernel were first to minimize the overhead of processor-local operations and second, to minimize communication between different processors.

The results show that the performance of the IPC-operation between threads on the same processor is comparable to the performance of IPC of a uniprocessor kernel. Whereas inter-processor operations tend to be much slower in terms of sending two inter-processor messages.

2.4.2 Fiasco Microkernel for the IA32 Architecture

The Fiasco microkernel has been ported to support small scale multiprocessor systems [18]. The goal of this work was to extend current design of the kernel to a multiprocessor system, by adapting the locking primitives.

The Fiasco microkernel uses non-blocking synchronization implemented as a helping mechanism [6]. This approach forces that all synchronization uses inter-processor synchronization primitives.

The evaluation showed that this approach is not very scalable and results in increasing synchronization overhead for a small number of processors.

2.4.3 Pistachio Microkernel for the IA32 Architecture

The Pistachio microkernel is an implementation of the L4.x2 interface. The aim of the multiprocessor design is to explore full scalability and flexibility of a microkernel-based system [23]. The three main goals are: preserving parallelism of applications, bounding the overhead of specific operations and preserving locality of applications.

To attain these goals a processor mask for every kernel object is introduced which tracks the usage of every kernel object. Furthermore, a dynamic locking scheme is implemented, based on an adaptive locking mechanism. For objects which are accessed by one processor fast CPU-local synchronization is used, otherwise slower inter-processor synchronization is used. For complex data structures, like the mapping database, the kernel can adapt the lock granularity.

The results of this work show that scalable microkernels with acceptable performance are possible.

3 Design

The design section has two parts. The first part specifies and reasons an L4.sec high-level multiprocessor model. The communication model and the kernel memory model are investigated in detail for this purpose. The second part focuses on the low-level synchronization of kernel operations and the requirements and design of synchronization primitives.

3.1 Design Goals

The design goals of this work are based on the following observation: Communication in multiprocessor systems can be classified into CPU-local and xCPU operations [1]. CPU-local operations can use fast CPU-local synchronization, whereas xCPU operations uses slow xCPU synchronization primitives. Multiprocessor applications tend to avoid xCPU communication and emphasize CPU-local communication, to maximize performance and scalability. An application with only CPU-local operations and communication and no xCPU communication is scalable, because every CPU can execute its operation independently of other CPUs. A design of the multiprocessor microkernel has to take this requirement of applications into account and should provide fast CPU-local communication primitives [22].

On one extreme the microkernel could only provide CPU-local operations, assuring performance of kernel operations. But in this approach, the application is responsible for all xCPU synchronization without any support from the kernel.

In the other extreme the microkernel could only xCPU operations, which assures flexibility for kernel operations. This approach does not respect the above mentioned requirement of applications as it slows down CPU-local communication, and therefore such a system has unacceptable performance.

A better solution combines both approaches and therefore provides scalability and performance for kernel operations in a reasonable way. The kernel offers one set of kernel operations with fast CPU-local implementations and another set of kernel operations with slow xCPU synchronization. This design has to take into account that fast CPU-local synchronization can be compromised by xCPU synchronization and slow down. Therefore it has to focus on how to integrate fast CPU-local kernel operations with xCPU operations.

The evaluation of different models uses the criteria of flexibility and performance. This raises the question of how scalability and performance of a system can be measured. The measurement of scalability requires support of scalable hardware, which means in the context of multiprocessor system, that the number of CPUs can be increased or decreased according to the needs. As such support was not available during my work, theoretical considerations have to be made to estimate effects of a design decision. To measure the performance of the microkernel, evaluating the performance of an application with a typical parallel workload is reasonable. But currently there are not such applications available for L4.sec, so the evaluation will be carried out with the help of micro-benchmarks.

The next two sections refine and narrow the design space for the multiprocessor kernel objects and kernel operations in L4.sec.

[1] A xCPU operation is a kernel operations that accesses global data and therefore requires xCPU synchronization

3.1.1 Kernel Objects

In a multiprocessor kernel, kernel objects have to be CPU-aware. The additional dimension adds more complexity in the structure and behavior of a kernel object.

Tasks

Tasks are resource containers for other kernel objects and therefore also have to control the propagation and accessibility of resources over CPUs boundaries. It is reasonable that in a multiprocessor system tasks can span over a set of CPUs. Thus the contained resources are accessible from this set of CPUs. For example a tasks can allow threads from different CPUs to access an endpoint and to communicate with each other.

The proposed model defines a task hierarchy according to the following relationship. A task controls another task, if it has provided the memory resources for the instantiation of the associated kernel object. When a new task is created, a set of CPUs for this task has to be specified, which it has access to. There are two useful restrictions, for this model: The first restriction is that the initial task created by the kernel at boot time, should span all CPUs. The second restriction is that a created task can only span over a subset of CPUs of the controlling task. The resulting structure is a tree-like resource graph, where initial task controls all CPUs and subsequent tasks have only access to a specific subset. This approach also supports partitioning of the whole system into independent subsystems.

Threads

A thread represents a state of execution and is bound to a single CPU at a time. Threads are related to a scheduling model. Evaluating multiprocessor scheduling models is not in the scope of this thesis, instead a pragmatic approach is chosen.

Scheduling in L4.sec uses a priority-aware round-robin scheduler. Priority-based scheduling suffers from an anomaly known as priority inversion, meaning that a thread with a high priority is blocked by a thread with a lower priority. An example is that a high-priority lock requester waits for a low priority lock owner to release the lock. A feasible multiprocessor design with priority-based scheduling has to avoid priority inversion or at least bound the time of priority inversion for a thread. This makes lock primitives complex. Due to that fact a simplified scheduling model is chosen, which uses priority-less round-robin scheduling.

Another concept considers thread migration, which is defined as moving a thread from one CPU to another. There are various reasons for this behavior. The most important one is load balancing: Assume one CPU has a high workload and another one is underutilized. A thread can migrate from the first to the second CPU to lower the workload of the first one. The next two items discuss how thread migration can occur.

1. kernel-level thread migration

   Threads migrate from one CPU to another CPU while executing a kernel operation. This is not always a desirable behavior because the in-kernel policy can compromise a user-level strategy for load balancing.

2. user-level thread migration

   Threads are migrated only on explicit user-level requests using a specific kernel operation. The application needs proper feedback of the kernel to decide if a migration is necessary. For example the kernel has to expose the workload the CPUs in the system.

User-level thread migration is more appropriate than kernel-level thread-migration for load balancing as the application knows the current and future distribution of its workload. Furthermore the in-kernel policy for kernel-level thread migration contradicts the microkernel paradigm.

Endpoints

Endpoints are the key component in the design of an L4.sec multiprocessor model to achieve fast and scalable communication. Therefore a detailed discussion follows in section 3.2.

3.1.2 Kernel Operations

Regarding multiprocessor systems, kernel operations can be classified into CPU-local operations and xCPU operations. A CPU-local operation needs only CPU-local synchronization because it accesses only CPU-local kernel objects, otherwise it is called a xCPU operation.

Thread-related Kernel Operations

An operation on a thread should be possible regardless on which CPU the target thread runs. Thread-related kernel operations are in the focus of performance optimizations. The priority for optimizing the kernel operations is depicted in the following list:

1. CPU-local IPC-operations
2. other CPU-local operations
3. xCPU operations

The IPC-operation can be considered the most performance critical operation in a microkernel operating system as it implements the basic communication mechanism [10]. The performance of CPU-local IPC-operations should be comparable to the performance of the IPC-operation of a uniprocessor system. The synchronization overhead added for the implementation of xCPU operations should be minimized but not affect CPU-local operations negative.

MDB-related Kernel Operations

Because resources are shared over processor boundaries, the mapping database is accessible from all CPUs. Thus mapping operations have to use xCPU synchronization to protect concurrent modifications. Performance optimizations and maximizing parallelism are not primary goals of this work, therefore I chose to integrate the current design.

3.1.3 Kernel Memory

Dynamic kernel memory management provides flexibility but adds complexity in kind of non-type-stable kernel memory, which has to be managed. On a uniprocessor system the deletion of an object while holding a reference can be prevented by acquiring the CPU-lock. For multiprocessor systems a CPU-lock is no longer appropriate instead additional synchronization effort is necessary.

The following two properties have to be satisfied to maintain correct behavior of a system with non-type-stable kernel memory and to minimize synchronization overhead:

1. *Synchronization of non-type-stable kernel memory has to ensure that invalid kernel object references are prevented.*

2. *Synchronization of non-type-stable kernel memory must not add synchronization overhead for operations that do not change the type of kernel memory.*

If a design of kernel memory synchronization can satisfy these two requirements it is considered a practicable and flexible solution.

From the previous discussion follows that the focus of this work is on the communication model and the kernel memory model, which are now discussed in detail.

3.2 Communication Model

The communication model is based on endpoints providing a communication channel. In a uniprocessor system, an endpoint has only one wait queue but in multiprocessor system the structure of an endpoint is extended to support multiple wait queues. The abstract communication model I consider has a static mapping of CPUs to the wait queues of the endpoint as shown in Figure 3.1.

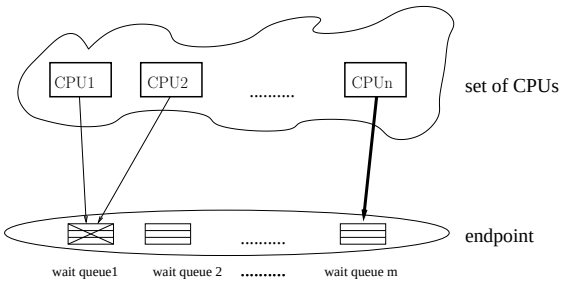

Figure 3.1: Model for multiprocessor endpoint

Every CPU uses one specific wait queue or none at all. If a wait queue has only one CPU mapped to it, it is identified as a CPU-local wait queue. In all other cases where the wait queue has more than one CPU mapped it is handled as a global queue. A distinction between symmetric and asymmetric global wait queues can be made. In the symmetric wait queue all CPUs use the same path to access the endpoint and synchronize; in the asymmetric model one CPU can use a faster path, similar to CPU-local wait queues, while the other CPUs use it like a remote queue.

This abstract model has some restrictions which should be pointed out: It does not make any changes to the used FIFO-strategy [2] in the endpoint. The endpoint is a static kernel object and dynamic adaption of the structure of the endpoint is not permitted, for example changing a CPU-local wait queue to a global wait queue or adapting the number of wait queues. Specifically the mapping of a CPU to an endpoint stays fixed. Both restrictions can be relaxed but expand also the design scope and make the model more complex.

From this abstract model one can derive more specific ones, for example endpoints with only one wait queue. The following items discuss five such concrete multiprocessor endpoint models. The resulting communication models are measured in scalability and performance of the communication channel. The estimation is made for a simple client-server model with two CPUs. Scalability is regarded as the effort the server has to make, to provide its service on two CPUs

[2] first-in first-out strategy

and extend the service to a third CPU. The performance is derived from the fact, if a CPU-local or a global wait queue is used.

Endpoints with one queue

Endpoints with one queue result in the simplest possible models. They are easy to implement and have a low complexity. There are three models possible.

- Endpoint with one CPU-local wait queue

  This endpoint has a wait queue which can only be accessed by threads of one specific CPU. For example the CPU can be selected at the creation time of the endpoint.

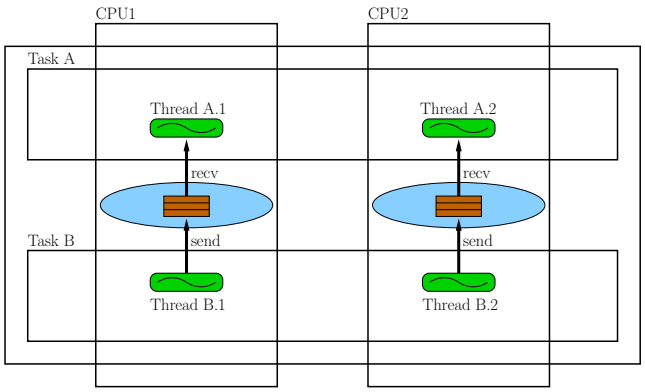

Figure 3.2: Client-server example of endpoint with one CPU-local wait queue

  In Figure 3.2 the client-server example with two CPUs is shown. Task *A* has instantiated two service threads and two endpoints. Client *B* has two threads running on CPU1 and on CPU2 accessing the service provided by task *A* from different endpoints. The communication channel between the client and the server uses fast CPU-local communication. If the service should be extended to another CPU the server has to provide another endpoint and another thread. Thus the endpoint is not scalable. Furthermore building a single-threaded server, which can handle requests from different CPUs, is not possible with this type of endpoint.

- Endpoint with one global symmetric wait queue

  This type of endpoint has one global wait queue. Thus thread from different CPUs can access it.

  In Figure 3.3 the server *A* has one service endpoint for both service threads. It follows that client *B* can request the service with both client threads using the same endpoint. All client threads are equally preferred, as the endpoint implements a FIFO-strategy. Furthermore a client thread does not know the CPU on which the service is handled. If the service should be extended to another CPU the server just needs to provide the same endpoint and can serve request without instantiating another thread. Even if one of the two server threads is deleted the service is available from all CPUs. Therefore a server has to provide only one endpoint and one service thread because the service thread can serve client threads on the same or remote CPUs. The disadvantage is that the performance of CPU-local communication is as slow as xCPU communication.

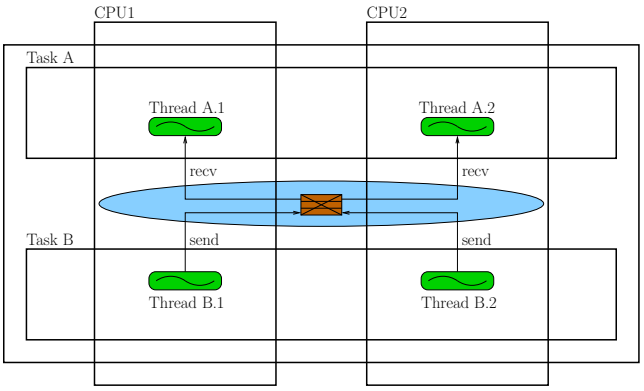

Figure 3.3: Client-server example of endpoint with one global symmetric wait queue

- Endpoint with one global asymmetric wait queue

  The next endpoint type has also one global queue but unlike the previous model, there threads on one specific CPU can access the wait queue with CPU-local communication overhead while threads on other CPUs have more synchronization overhead. The preferred CPU can be selected at creation time of the endpoint.

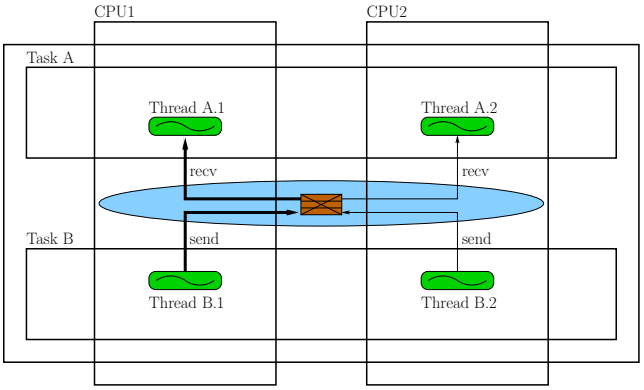

Figure 3.4: Client-server example of endpoint with one global asymmetric wait queue

The server has selected CPU1 as the preferred CPU as shown in Figure 3.4. Therefore the communication of client thread *B.1* with server thread *A.1* is fast. But there is no preference of this communication relationship compared to server thread *A.2*.

This model has a special use case for a multi-threaded server, which normally has only one service thread on CPU1 and occasionally on high workload creates additional service threads on other CPUs. The first service thread always uses the fast communication path and additional service threads have to use the slow communication path.

Endpoints with multiple wait queues

The structure of an endpoint becomes more complex if it provides multiple wait queues. Especially the mapping from a CPU to a wait queue may have a negative impact on the performance.

The following two models are the simplest models possible for an endpoint with multiple wait queues.

- Endpoint with multiple CPU-local wait queues

  This type of endpoint has a specific number of CPU-local wait queues. If a thread blocks in the endpoint, it is enqueued in the corresponding wait queue of the CPU, where it is running. Consequently only CPU-local communication is available. If there is no server thread on the CPU the client thread has no chance to be served, even if there are server threads waiting on other CPUs.

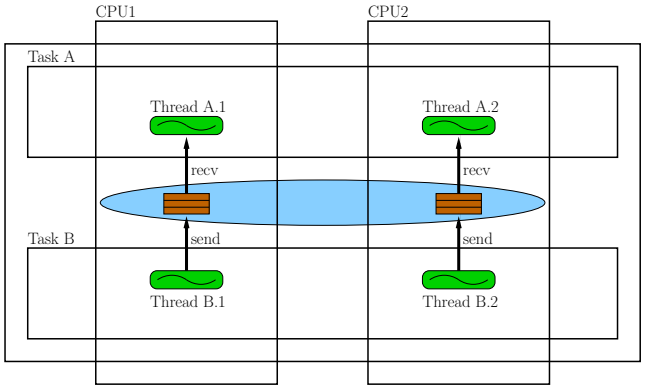

Figure 3.5: Client-server example of endpoint with multiple CPU-local wait queues

  The client-server example, shown in Figure 3.5, uses an endpoint, which has two wait queues mapped to CPU1 and CPU2. The two client threads access the server using the same endpoint but different wait queues. Despite the fact that both client threads have the same entry point they are served with different server threads.

  If a new CPU is added to the system and the endpoint has no free wait queue to map the CPU, it has to instantiate another endpoint to provide the request on the new CPU. Thus scaling is only possible, if the number of preallocated wait queues is at least equal to the maximal number of CPUs possible in the system. On the other hand, this can lead to wasting of memory resources. For example on a system with 100 CPUs, where a server only want to provide the service on 2 CPUs, 98 wait queues are left unused. Another disadvantage is the fact that a server has to be multi-threaded if it wants to provide a service on more than one CPU. The advantage of this endpoint is that the communication between client and server has low synchronization overhead.

- Endpoint with multiple CPU-local and one global symmetric wait queue

  The last endpoint type is an extension of the previous one. Just like this, it has a specific number of CPU-local wait queues, but additionally it has also a global wait queue.

  In the client-server example in Figure 3.6 the server has created an endpoint with an CPU-local wait queue mapped to CPU1 and the global queue which is currently used for the server thread on CPU2. Therefore the client has a fast communication channel on CPU1 and a slow communication channel on CPU2.

  If another CPU is added to the system, where the service is requested from, the global wait queue has to be used. Thus the server does not need any new thread on this CPU.

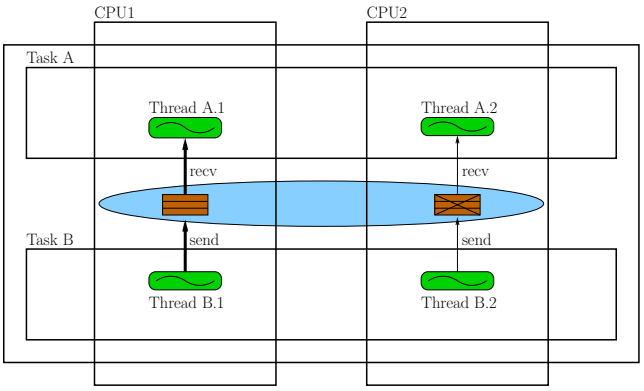

Figure 3.6: Client-server example of endpoint with multiple CPU-local wait queues and global symmetric wait queue

On the other side it is not possible to add a new fast CPU-local communication channel, if all CPU-local channels are used.

Discussion

All the above discussed communication models have shortcomings. They can be classified into two subsets. One subset provides only fast CPU-local communication and the other subset provides scalable xCPU communication. There is no particular model that provides fast and scalable CPU-local communication. Scalability and performance are contradicting requirements, which cannot be fulfilled of this communication model.

It turns out that the aforementioned restrictions are to limiting. A more feasible model requires the dynamic adaption of the number of wait queues to the number of processors in the system.

3.3 Kernel Memory Model

All kernel objects are based on kernel memory resources. L4.sec has user-provided kernel memory. This means if an application wants to create a kernel object, it has to give the necessary memory resource to the kernel. Further if the memory resource for a kernel object is revoked by the application all kernel objects are destroyed.

Creating a kernel memory resource results in a kernel memory object (KMO). The kernel organizes kernel memory objects in its kernel address space. The application can specify which KMO is used for creation of a kernel object.

To compare different possible designs of kernel memory, I define an abstract model for the translation of a virtual address in the kernel address space to a physical address, which represents the kernel object. The model extends the translation as done by hardware architectures with two imaginary steps. The translation process is done in three steps as shown in Figure 3.7. In the first step a virtual kernel address is mapped to a partition. The second step selects the KMO in the partition. The last step translates the virtual address into a physical address. The physical address references a kernel object.

A virtual address and a kernel object have associated a type and an identification number. Thus the type and identification number of the virtual address can be matched against the type and identification and type of the kernel object. A translation is only valid, if identification

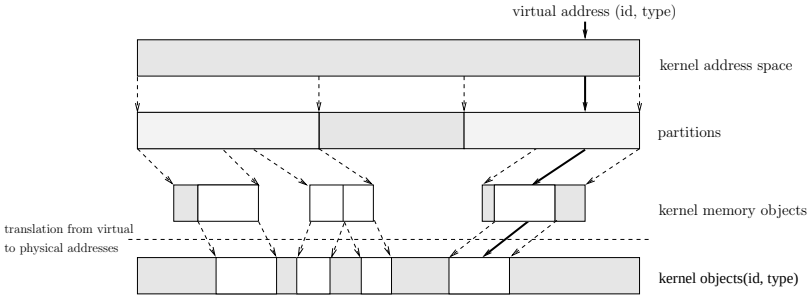

Figure 3.7: Translation from a virtual address to a kernel object

number and type match otherwise the translation is invalid and should fail. As real hardware architectures have untyped memory, type checking and identification number checking cannot be implemented efficiently by the kernel. The kernel has to use some mechanisms to prevent invalid access. For example, the model can be restricted in a way, that invalid accesses to kernel objects do not occur.

Following this approach there are four different models of type-stability, which have different strengths and weaknesses, discussed in the following points:

- type-stable partitions

  In this model, shown in Figure 3.8, the kernel address space is partitioned into static sections which can hold one type of kernel object. The application has to decide into which partition a KMO is mapped and thus which type of kernel objects can be created in the KMO.

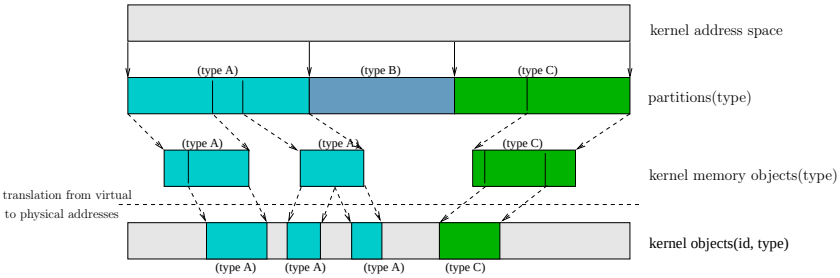

Figure 3.8: Kernel memory with types-table partitions

  On one hand this model is the most restrictive one, because the layout of the kernel address space is fixed and cannot be adapted during the execution of an application. The size of a partition has to be large enough to hold the maximum number of kernel objects of a specific type as required by the application. On the other hand it assures type-stability of kernel memory.

- non-type-stable partitions

  To circumvent the disadvantage of the previous model, partitions can change the type at runtime on explicit request of the application. During this operation the kernel memory changes its type and needs special synchronization effort. This means kernel memory is not type-stable anymore. Despite that fact, the effort for synchronization is minor compared to

the next models, if the application only needs to adapt the partition infrequently. In this model an application needs knowledge of the partition layout of the kernel address space.

- type-stable kernel memory objects

  Partitioning kernel memory is a very inflexible solution as it depends on the granularity a partition. This is resolved by making partitions transparent, partitions and KMOs are identical. Thus the abstract model is simplified as shown in Figure 3.9. The application, which provides the kernel memory, has to tag it with a type. Due to that a KMO can only contain kernel objects of one specific type. If the kernel object is destroyed, the type of the underlying memory does not change. If the KMO is destroyed, non-type-stable synchronization has to be provided, because a new KMO with a different type could be mapped to the same virtual address.

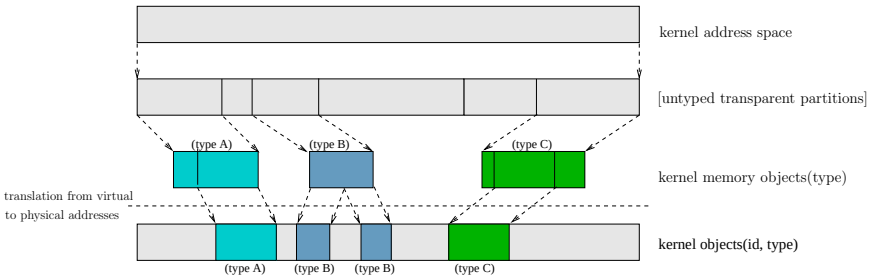

Figure 3.9: Kernel memory with types-table partitions

This model is more flexible than the previous one as the application does not have to know partitions. The number of constructed kernel objects of a specific type is only restricted by the amount of KMOs an application can create.

- non-type-stable kernel memory objects

  The most flexible model has no restriction regarding which types of kernel objects can be created within a KMO. Thus KMOs can contain kernel objects of different types as shown in Figure 3.10. But every destruction of a kernel object may change the type of the kernel memory. Therefore it follows that the destruction of kernel objects needs synchronization because invalid access to previously deleted kernel objects has to be prevented. The synchronization effort is higher than in the previous model.

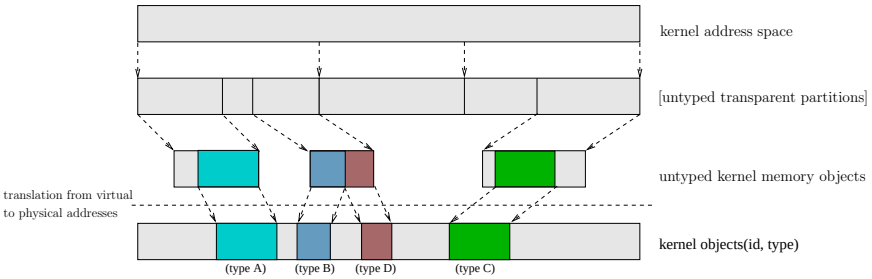

Figure 3.10: Kernel memory with types-table partitions

This model conforms with the current implementation in the L4.sec microkernel.

Discussion

From the above analysis follows that more flexibility in the management of kernel memory adds more synchronization overhead to handle non-type-stable kernel memory. The first model has no synchronization overhead because, the kernel memory is type-stable, while the last model has the highest synchronization overhead. Furthermore the first three models lack efficient usage of the kernel memory resources because a KMO can only contain kernel objects of one specific type. This is a drawback for systems with small amount of memory. The flexibility of the first and the second model is further restricted by the granularity of the partitions.

3.4 L4.sec Multiprocessor Design

The following multiprocessor model is a pragmatic approach to validate the main goals of this work. It contains some reasonable assumptions and restrictions. The first assumption in the model presumes that the maximal number of CPUs is known at configuration time and does not change. This is acceptable for small-scale multiprocessor systems. Kernel objects in the multiprocessor model are designed as follows:

Tasks

The implemented task model is special case of the introduced model in the design section. Tasks and their memory spaces and capability spaces always span all CPUs. All kernel objects created by a task are visible and accessible from all CPUs. Thus effective resource control over CPU boundaries is not available, but sharing of resources is possible. This model is sufficient to validate xCPU synchronization primitives.

Threads

Threads have a CPU, which they are bound to and which is specified on creation time of the thread. A thread can migrate to another CPU only on an explicit user-level request, which assumes that the thread is stopped. The scheduling model assumes that threads are scheduled without a priority in round-robin fashion.

Endpoints

Due to the revealed shortcomings of the discussed communication models, is chose to provide different types of endpoints, which target the problem of fast and scalable communication. This conforms with the principle that the design of a microkernel should provide necessary mechanisms. An application has to decide, which type of endpoint to use depending on its requirements. The current design provides two types of endpoints:

- Global endpoint

  A global endpoint has one global symmetric wait queue and is identical with the second proposed communication model. Thus this type of endpoint is scalable but provides only slow communication.

- Local endpoint

  A local endpoint has n CPU-local wait queues, where n is the number of CPUs specified at the creation time of the endpoint. The mapping of CPUs to wait queues uses the simplest

approach possible: The first CPU is mapped to first wait queue, the second CPU is mapped to the second wait queue and so on.

This approach can be easily extended to integrate other types of endpoints. As the structure and the requirements of multi-server multiprocessor become better known, new types of endpoints can be added or current existing one can be removed.

Kernel Memory

The multiprocessor model for kernel memory management uses the most flexible model of non-type-stable kernel memory. Therefore a KMO can contain different kernel objects of different type. The consequential synchronization overhead can be accepted under the premise that destruction of kernel object occurs infrequently and is not performance critical.

3.5 Synchronization of Kernel Operations

The following part discusses the low-level of the design, which is focussed on a synchronization scheme that supports the goals of the high-level model. The main effort is put in the development of a synchronization primitive for threads and in the synchronization of non-type-stable kernel memory.

3.5.1 Synchronization of Concurrent Operations

If two operations are executed concurrently on different CPUs and compete to access the same kernel object they are considered concurrent operations as shown in Figure 3.11.

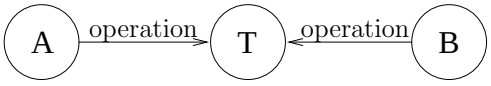

Figure 3.11: Concurrent operations

A typical example is a thread in an IPC-operation that wants to send a message to a target thread and a thread in an exregs-operation that wants to stop the target thread. Only one of these two operations can access the thread at a time and therefore execute its operation otherwise the state of the thread might get corrupted. The general solution to resolve this type of conflict is to lock the target object, which provides mutual exclusion. Table 3.1 gives an overview, which kernel operations have to be protected against each other [3].

[3] The lower half of the table is left blank, because it is symmetric to the upper half.

kernel operation	IPC	cancel	exregs	map	unmap	create	destroy
IPC-operation	x	x	x	-	-	-	x
cancel-operation		x	x	-	-	-	x
exregs-operation			x	-	-	-	x
map-operation				x	x	-	-
unmap-operation					x	-	-
create-operation						-	-
destroy-operation							-

Table 3.1: Which concurrent kernel operations have to be protected against each other?

There are two groups of kernel operations, which need concurrent synchronization:

1. Thread-related operations have to be synchronized to prevent corruption of the thread state.
2. MDB-related operations have to be synchronized to prevent corruption of the mapping database.

MDB-related operations and thread-related operations do not need synchronization although they access entries in the memory space and capability space concurrently. But corruption of the memory space or the capability space can be prevented by using atomic access and update functions. Therefore a mapping operations that modifies a capability uses an atomic function and avoids that other kernel operations read invalid capabilities.

The create-operation needs no synchronization with other kernel operations as it creates a new kernel object, which cannot referenced before the end of the operation.

3.5.2 Synchronization of Dependent Operations

Dependencies between kernel operations exist because threads execute kernel operations and can itself be the target of a kernel operation. Thus if a thread executes a kernel operation on another thread, which itself tries to execute a kernel operation, there is a dependency between these two threads. The first operation is called unconditional operation and the second one is called conditional operation as shown in Figure 3.12.

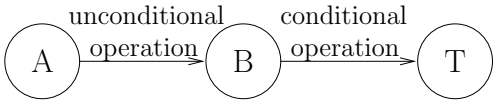

Figure 3.12: Example of two linear dependent operations

If the conditional operation is executed before the unconditional operation both operations succeed. But in the other way around, the conditional operation may be aborted. It is important to notice that if a kernel operation can be aborted by another kernel operation the kernel data structures have to be in a consistent state after completion of the abort. For example, all locks are released.

A special dependency occurs, when a thread A executes a kernel operation which requests the target thread B, while thread B requests a kernel operation on thread A. Both threads are

blocked by a circular dependency shown by Figure 3.13. These dependencies can span a chain of more than two threads.

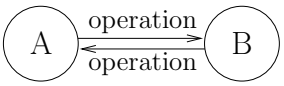

Figure 3.13: Example of two circular dependent operations

A deadlock occurs if both kernel operations request the lock on the target thread, but the condition for acquiring the lock does never occur. A race occurs if both kernel operations obtain the lock of the target thread and execute its operation concurrently. There are two approaches to resolve dependencies:

The first possible solution uses ordering of lock requests. Thus between two lock requests exists a precedence relation. A simple example is a lock counter, which is incremented on every lock request. A lock is granted if the lock counter of the lock requester is lower than the counter of the own lock request.

The second possible solution uses overlapping sets of locks for dependent kernel operations. The locks are obtained in a well-defined order. For example, this set of locks can comprise exactly two locks, the lock on the target thread and the lock of the executing thread itself.

kernel operation	IPC	cancel	exregs	map	unmap	create	destroy
IPC-operation	x	x	x	-	-	-	x
cancel-operation		x	x	-	-	-	x
exregs-operation			x	-	-	-	x
map-operation				-	-	-	-
unmap-operation					-	-	-
create-operation						-	-
destroy-operation							-

Table 3.2: Which dependent kernel operations have to be synchronized?

Table 3.2 shows which kernel operations have dependencies. As expected these are only kernel operations, which have a thread as target object and thus mapping-operations do not fall into this category.

3.5.3 Synchronization of the Thread-related Operations

The state of a thread has to be protected from concurrent access with a thread-lock. The only assumption about the properties of the thread-lock is that it has to assure fairness.

In the first approach constructs a thread-lock that handles all aforementioned situations properly and is obtained by all thread-related operations. As it turns out, this solution has shortcomings, which make it inadequate for the implementation of the multiprocessor model. Therefore the second approach restructures the thread-lock into two specialized thread-locks, which resolves the disadvantages and is more appropriate.

1. Approach: Generalized thread-lock

The lock is described using a lock protocol. The protocol is derived step by step examining the synchronization cases in the previous section. In every step the protocol is refined to handle the new case properly. The complete state diagram of the protocol can be found in appendix A.

Locking a thread in a multiprocessor kernel may involve the cooperation of two CPUs. The first CPU (source CPU) runs the thread, which wants to lock a target thread on the second CPU (target CPU). If the target thread gets locked, it is stopped and is not allowed to hold or request any resources.

Single lock request

Figure 3.14 shows an example of a single lock request. The sequence starts with a ***lock-message***, which the lock requesting thread A sends to the CPU of the target thread T. Thread A changes to the state **lock_requester** and blocks. When the message arrives at the target CPU, the target thread is stopped and pinned to the state **locked**. This means the thread does not run and cannot be scheduled. At this point the lock request has succeeded, which is signaled to the lock requester with an ***ok-message***. The arrival of this message changes the state of the lock requester from **lock_requester** to **lock_owner** and wakes up thread A, which can now execute the desired operation on the target thread T. When it has finished the operation an ***unlock-message*** is sent, which releases the locked thread on the target CPU.

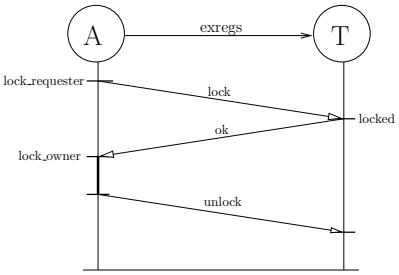

Figure 3.14: Example of a single thread-lock request

Concurrent lock requests

Figure 3.15 depicts concurrent lock requests on the same target thread. Two threads A and B try to lock the same target thread T. To handle concurrent requests fairly a lock queue is introduced as part of the thread-lock. A lock-requesting thread enqueues itself at the end of the lock queue of the target thread. Threads in the lock queue are handled in FIFO order. Before the thread releases the lock, it dequeues itself from lock queue.

Thus as thread A enqueues itself before thread B, it is responsible for sending the ***lock-message*** to the target thread T; while B recognizes that it has to wait for thread A to release the lock. After thread A has obtained the lock and executed its operation, it passes the ownership of the lock to thread B by sending an ***ok-message*** to thread B. Thread B is now the lock owner and can execute the operation. It then inspects the lock queue of thread T for the next lock requester. As the lock queue is empty thread B sends the ***unlock-message*** to thread T releasing the target thread.

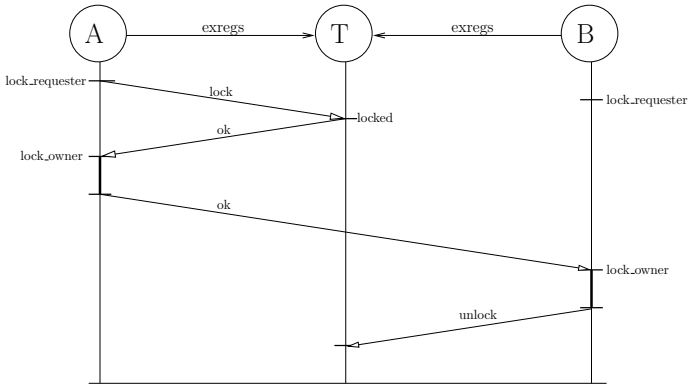

Figure 3.15: Example of concurrent synchronization

The destroy-operation has a slightly different protocol compared to the exregs-operation because the lock is not allowed to be granted to another thread. As shown in Figure 3.16 thread A and thread B request concurrently the lock of thread T with the intention to destroy the thread. Only one of them can succeed. As thread A obtains the lock, it can perform the destroy-operation. It has to send an ***abort-message*** to thread B before the deletion of the target thread. This interrupts the lock request of thread B, which is removed from the lock queue and woken up. Thus instead of releasing the lock, all subsequent lock requesting threads have to be aborted with an ***abort-message***.

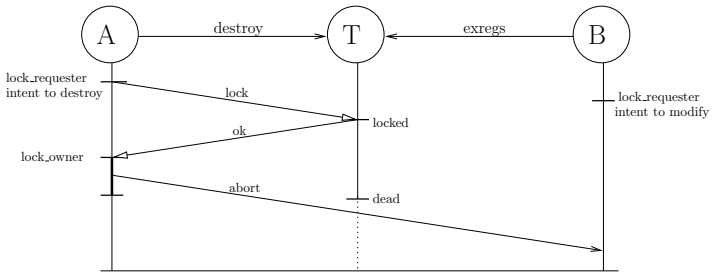

Figure 3.16: Example of concurrent synchronization for the destroy-operation

Linear dependent lock requests

The next extension of the protocol handles linear dependencies as shown in Figure 3.17. Thread A, which requests a lock on thread B, which in turn requests a lock on thread C, which finally requests a lock on thread D build up a dependency chain. This chain of lock requests has to be resolved by restricting the grant of locks. First to note is that thread B and C after receiving a lock request change to the state **lock_requester + locked**, which specifies that the thread has requested a lock, but also wants to be locked by another thread.

The protocol proposes that the lock request is only acknowledged, if the target thread does not have itself an outstanding lock request or holds already a lock. Therefore it follows that the state **lock_requester + locked** is a blocking state while the state **lock_owner + locked** is a non-blocking state. The dependent operation is always executed before the independent operation.

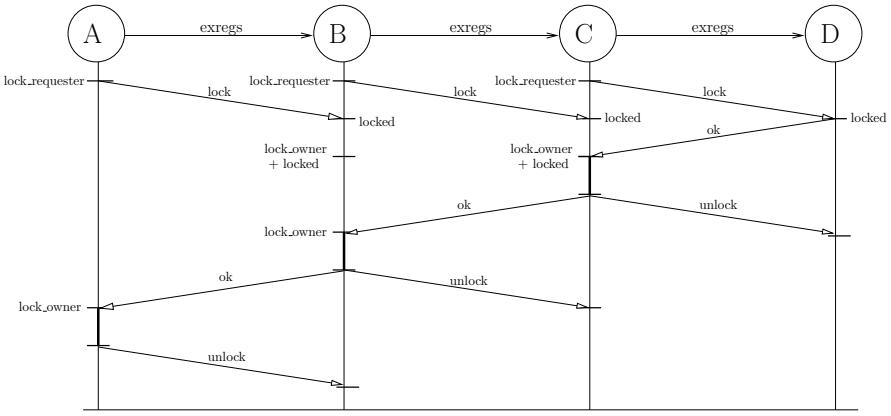

Figure 3.17: Example of linear dependency synchronization

The chain in the example is resolved from the right to the left. After thread C has received the ***ok-message*** from the thread D, it switches to the state **lock_owner + locked**, executes its operation and releases the target thread, while immediately sending the ***ok-message*** to thread B, which now in turn can execute its operation. In the next step thread B sends the ***ok-message*** to thread A.

Circular dependent lock requests

The last extension of the protocol incorporates circular dependencies. In the accompanied example in Figure 3.18, thread A wants to lock thread B, which wants to lock thread C and finally thread C wants to lock thread A. If in the current protocol all threads send its ***lock-messages*** simultaneously, all threads end in state **lock_requester + locked** waiting for the ***ok-message*** from the target thread. As no thread can execute its operation, a deadlock occurs.

To avoid the deadlock, locks have to be ordered or overlapping lock sets have to be used. I chose the first solution because requesting two thread-locks for one operation doubles the synchronization overhead, which is not acceptable. Therefore every lock request has a unique value attached, called the lock counter describing the lock precedence. For example the lock counter of thread A is lower than the lock counter of a thread B. This means that thread A's lock request is preferred over to thread B.

A thread which is in the state **lock_requester + locked** can only execute its operation, if it is in the substate **lock-preference**, otherwise it has to cancel its lock request by sending a subsequent ***unlock-message*** and replying with an ***ok-message*** to its lock requester. This resolves the circular dependency.

In the example, threads B and C realize that they do not have the lock precedence and cancel their lock requests immediately. Thread A becomes lock owner and executes its operation. After the thread A has sent the ***unlock-message*** to thread B, the circular dependency has been successfully resolved. Thread B and C have to start a new lock request with a new lock counter. In the example next thread C and finally thread B can obtain their locks.

This solution is correct but has the disadvantage of canceled lock requests. This leads to higher load in the system as the ratio between sent messages and successfully obtained locks decreases if dependencies between lock requesting threads occur.

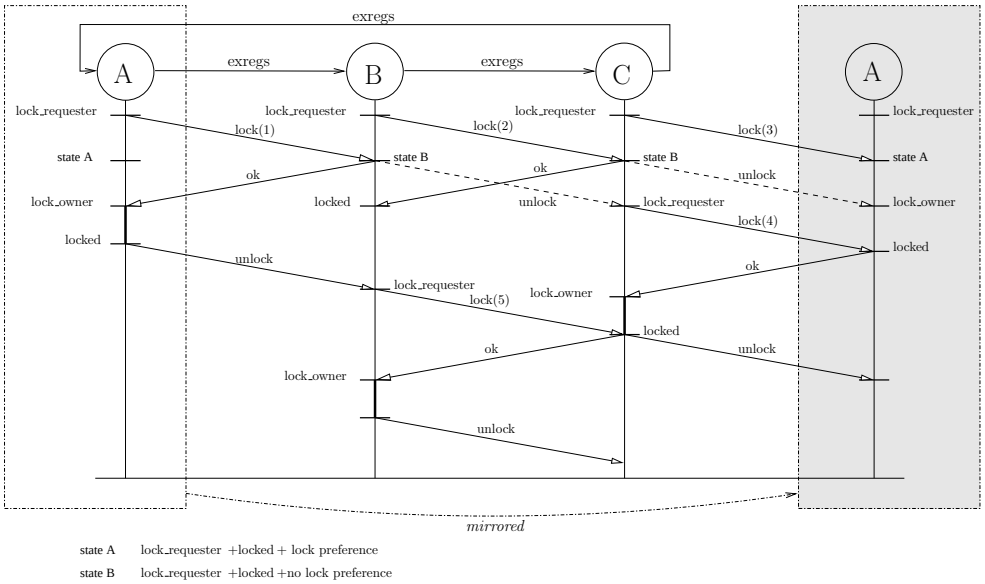

Figure 3.18: Example of circular dependency synchronization

Discussion

From the previous examination can be derived that a thread-lock, which is used to protect all thread-related operations, is complex and has undesirable behavior. There are more disadvantages which make this approach unfavorable:

There is the possibility of a deadlock between an IPC-operation and a cancel-operation. A cancel-operation locks the target thread before dequeuing it from the endpoint. After that the corresponding endpoint is locked. An IPC-operation first acquires the endpoint-lock to dequeue a partner from the wait queue of the endpoint and then locks the partner, which is the reverse order as in the cancel-operation. A deadlock occurs in the situation where the cancel-operation obtains the thread-lock and blocks on the endpoint-lock and the IPC-operation obtains the endpoint-lock of the partner thread but blocks on the thread-lock.

The second disadvantage to consider, is the synchronization overhead in the execution path of the IPC-operation. To provide maximum performance for IPC-operations the lock overhead should be minimized and therefore requesting an endpoint-lock and a thread-lock should be avoided.

2. Approach: Specialized thread-locks

Applying a thread-lock to all thread-related operations results in an overly restricting synchronization scheme. It seems reasonable to modify the lock primitive by using non-obvious properties of the kernel operations intrinsic to the L4.sec model. The following observations can improve the thread-lock and lead to a more feasible approach:

- The endpoint-lock is sufficient to synchronize IPC-operations and cancel-operations. The IPC-operation does not need to obtain the thread-lock and can execute the data transfer without explicit locking of the passive partner. This prevents deadlocks with the cancel-

operation and minimizes the synchronization overhead in the IPC-operation.

- Exregs-operations and IPC-operations are non-concurrent operations because they access different parts of the thread state. The IPC-operation transfers parts of the message using general purpose registers, while the exregs-operation changes special registers, like the instruction pointer and the stack pointer. Considering this situation, both operations can access the same thread at the same time and do not need any synchronization.

- The exregs-operation relaxes the requirements for the semantic of the thread-lock, if an exregs-operation cannot abort the ongoing exregs-operation of the target thread. Therefore, a thread, which is locked by an exregs-operation, is allowed to hold lock requests itself.

  A locked thread replies with a ***lock-message*** immediately, instead of executing its exregs-operation first. In Figure 3.19 is the adapted solution of example 3.17 shown. Now thread *B*, which becomes locked by thread *A* replies immediately to *A*'s lock-request and waits until *A* releases the lock. It can then finish its lock-operation subsequently. After thread *B* sends the ***unlock***-message to thread *C*, this thread is able to execute its operation.

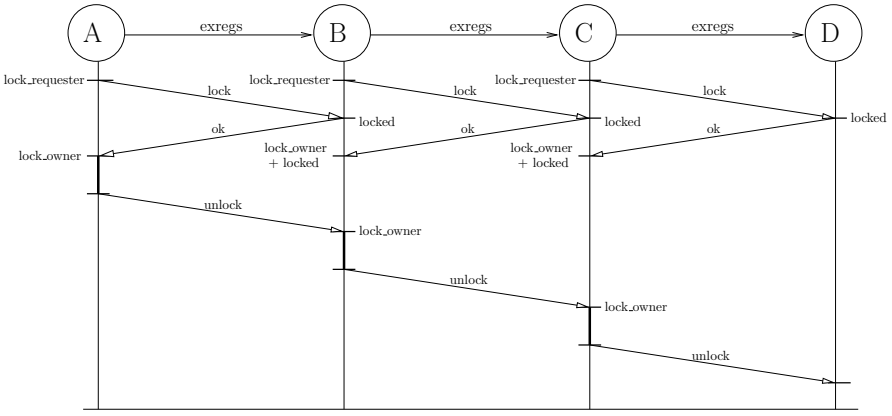

Figure 3.19: Relaxed linear dependency synchronization for exregs-operations

- The semantic of the thread-lock can be further relaxed for the exregs-operation. The prerequisite which permits that, is that exregs-operations only change registers, which affect the execution of user-level code. A locked thread, which is manipulated by an exregs-operation, is allowed to run in kernel mode, rather than being stopped. In consequence of that, there is no need to synchronize dependencies between two threads, which are performing an exregs-operation as the assertion that the locked target thread is not allowed to run in user mode holds. This reduces the complexity of the thread-lock and avoids deadlocks.

  The previous example can further be simplified as shown in Figure 3.20. All threads execute their exregs-operation simultaneously, without being affected by the ***lock-message*** of other threads.

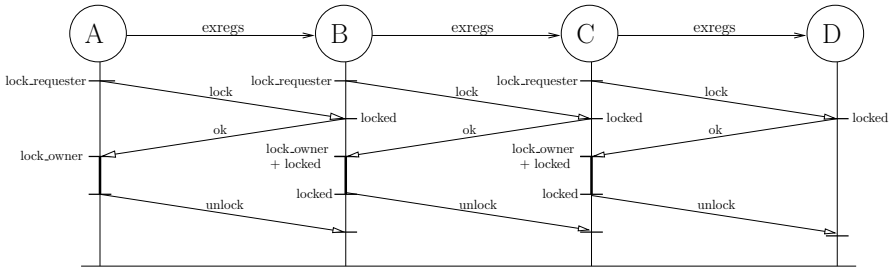

Figure 3.20: Optimized synchronization for exregs-operations

- Because the thread-lock for exregs-operations has specialized semantics, it is not sufficient for operations like destroying, migrating or unbinding a thread to obtain the lock, because the target thread might still run in the kernel and may even request or hold resources. Therefore a second thread-lock is introduced, which is additionally required by the above operations to guarantee that the target thread is stopped.

  This lock can be requested by only one thread at one point in time. In consequence the lock does not have to handle concurrent synchronization nor dependency synchronization.

  Figure 3.21 depicts the cascaded lock requests of a destroy-operation. After the first lock request has succeeded the destroy-operation is synchronized with other operations. The ***kill-message*** stops the target thread, releases hold resources and aborts ongoing kernel operations.

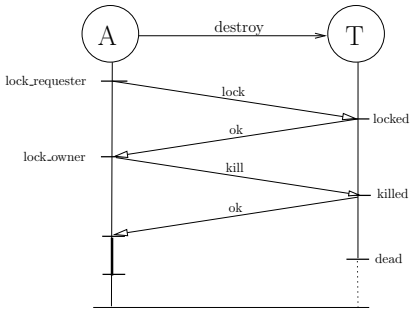

Figure 3.21: Cascaded lock request

Discussion

The above observations lead to two specialized thread-locks: The first lock has a relaxed semantic, which fits especially for exregs-operations and therefore is called exregs-lock. The second lock is only used by a thread, which executes a destroy-, unbind- or migrate-operation and is called kill-lock.

Comparing both approaches, the second one eliminates all disadvantages revealed by the first one. Most importantly it reduces the complexity of the specified thread-lock.

3.5.4 Synchronization of the Mapping Database

The mapping database is modified by map- and unmap-operations. A map-operation creates a new mapping node and inserts this mapping node as a leaf in the mapping tree. An unmap-operation removes a subtree of the mapping database. If the root mapping node of a kernel object is deleted, the kernel object is destroyed. This implies that all objects that were created by this object or depend on memory resources of this object are also destroyed. The process can be extended in a recursive way to the destruction of a whole subsystem.

If map-operations and unmap-operations modify the same subtree of the mapping database, they are considered conflicting. But if two these operations change different subtrees, they do not interfere with each other and can be executed concurrently.

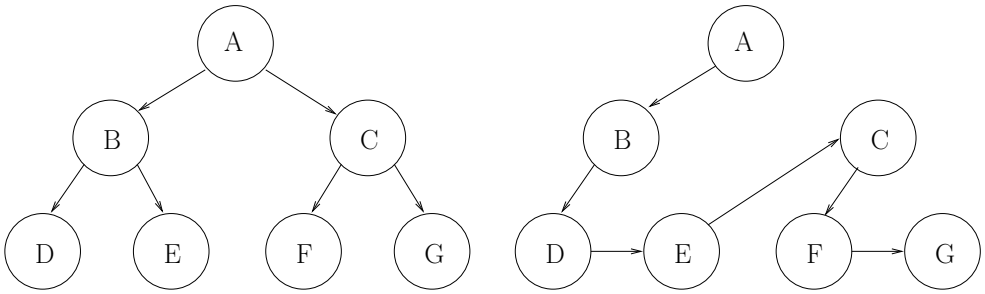

Figure 3.22: Logical and physical structure of a mapping tree

Figure 3.22 depicts that the logical structure of a mapping tree in L4.sec is implemented as a preordered double-linked list. For example inserting a new element in this list, requires the update of two pointers.

When searching for a solution for the synchronization of the mapping database two general approaches are possible. The first solution uses fine-grained locking and the second solution uses coarse-grained locking.

1. synchronization with fine-grained lock granularity

   This approach has a lock for every mapping node. It requires that locks are always obtained in forward direction, when traversing the mapping tree, otherwise a deadlock can occur. Every map-operation has to obtain two locks before it can insert the new mapping node. The unmap-operation has to obtain all locks of the subtree.

   The unmap-operation has a special case, called self-unmap, which requires an examination. The special property of the self-unmap operation is, that it removes also the mapping node, from which it accessed the mapping tree. This means it has to lock the parent node. To retain the correct order of lock acquisition, the self-unmap operation first goes upward to lock the parent node and afterwards goes downward to lock the subtree. There exists a race condition, which complicates things. Another thread can change the structure of the mapping tree between the parent node and the root node of the subtree. For example a new node may be inserted between both. Therefore the self-unmap operation has to validate, if the structure of the mapping tree has changed to forestall deletion of wrong mapping nodes.

   The fine-grained locking scheme has the advantage of maximizing parallelism of non-concurrent operations in the same subtree. But it adds synchronization overhead as each

mapping node has to be locked and increases the complexity of the implementation of the unmap-operation.

2. synchronization with coarse-grained lock granularity

   This approach has a lock for every mapping tree. The lock has to be obtained by map-operations and unmap-operations before accessing the mapping tree a kernel object.

   The coarse-grained locking scheme impedes parallelism of non-concurrent operations in the same mapping tree, but has less synchronization overhead than the fine-grained locking scheme and reduced implementation effort.

Discussion

Deciding between one of the two approaches requires the trade-off between higher parallelism and reduced implementation effort. As optimizing the mapping database is not a primary goal of this work, I chose the coarse-grained locking scheme to reduce the implementation effort. Another reason for this decision assumes that the properties of the synchronization primitives for both approaches can be considered at least similar. An adaption of a coarse-grained implementation to a fine-grained is due to that straight forward.

Following this approach every mapping tree of kernel object is protected by a lock, called unmap-lock. Map- and unmap-operation have to acquire this lock before changing the structure of the mapping tree. The unmap-lock is released after the completion of the operation. If the kernel object is destroyed, the unmap-lock has to be released before deletion.

3.5.5 Synchronization of Non-type-stable Kernel Memory

L4.sec has non-type-stable kernel memory, therefore it is possible that after the execution of a destroy-operation threads on other CPUs hold invalid references to the deleted kernel object or access a newly created kernel object of a different type.

A simple object lock, which is part of the kernel object it should protect, is not sufficient to prevent invalid object references as the following example illustrates. Assuming a thread A, which wants to destroy kernel object O acquires the lock for this kernel object. Another thread A wants to access the same kernel object O and requests the lock. At end of its destroy-operation thread A deletes the kernel object. Now thread B, which does not know that object O was deleted, may obtain the lock and access invalid data.

An approach to solve this problem, is to remove all access paths to the kernel object before it is locked and deleted. This prevents other threads to request the lock of the kernel object while it is deleted. But another example reveals that also this assumption is not sufficient to prevent invalid access. In the first phase thread A, which executes the destroy-operation, removes all references to the kernel object O. During this phase thread B has a valid reference. The destroy-operation starts the second phase by locking and deleting the kernel object O. Again thread B has a invalid reference can accidently access kernel object O.

Considering this example it is necessary to introduce a delay phase between the removal of the references to kernel object and the deletion of the kernel object. This delay phase must be long enough to guarantee that all threads, which want to access the kernel object have finished. This raises the question how the delay time can be measured.

As stated out in section 7, RCU is a synchronization mechanism, which exactly solves this kind of problems. Applying RCU for synchronizing kernel memory means that the destroy-operation has the role of the writer, which is responsible to synchronize kernel memory. Other kernel operations are in the role of readers and do not have any additional synchronization overhead. A

kernel operation is considered a read-side critical section, which cannot be compulsory preempted. If a voluntary preemption occurs, it has to be treated as the end of a read-side critical section. Resuming the kernel operation after the preemption is considered as the start of a new read-side critical section.

From the above discussion it follows, that the following two assumptions hold:

- *During the execution in the kernel a thread cannot be preempted, only voluntary blocking is possible and all references it accesses have to be valid until the end of this phase.*
- *After blocking in a kernel operation all references held by the thread, have to be considered as invalid.*

With these two assumptions it is possible to use the RCU mechanism to synchronize the kernel memory. The following points discuss in detail how the destroy-operation can use the RCU mechanism to safely delete threads, endpoints and tasks.

Destruction of a thread

The following list shows that the destruction of a thread is done in four steps. The first step removes all capabilities pointing to the thread, to prevent other kernel operation from obtaining a reference to the thread. Then the thread is locked and stopped aborting an ongoing kernel operation, which the target thread was executing. Thereafter the destroy-operation waits for the delay time to let all references become invalid. After the grace period, it is assured that no other thread holds a reference to the deleted thread. In the last step the destroy-operation deletes the target thread.

1. Invalidate all capabilities to the thread
2. Lock and kill the thread
3. Wait a grace period, so all references become invalid
4. Delete the thread

Destruction of an endpoint

The destruction of an endpoint as shown in the following list is also done in four steps similar to the destruction of a thread. In the first phase all capabilities are invalidated. After a grace period all references to the endpoint have disappeared. The third phase assures that the queue of the endpoint is empty and waiting threads are woken up. Now, the endpoint can be safely deleted.

1. Invalidate all capabilities to this endpoint
2. Wait a grace period, so all references become invalid
3. Remove all waiting threads from this endpoint
4. Delete the endpoint

Destruction of a task

The next list shows the necessary steps to destroy a task. As tasks are containers for resources, they own other kernel objects. These kernel objects also have to be destroyed before the task can be deleted. For example a thread, which is bound to this task can run on another CPU and can access resources of this task. Therefore a destroy-operation first has to stop the thread before the task is deleted. Another example assumes a task, which owns an endpoint with enqueued threads. Before the task can be deleted, the endpoint has to be destroyed and all threads in the wait queue have to be released. In general, before a task can be safely deleted all owned objects have to be destroyed, which may lead to subsequent calls of destroy operations.

1. Invalidate all capabilities to this task
2. Stop all threads of this task
3. Destroy all kernel objects of this task
4. Wait a grace period
5. Delete the task

3.6 Summary

The design of the low-level synchronization primitives introduces a number of locks to protect kernel objects. Additionally the RCU mechanism is used to synchronize non-type-stable kernel memory. The following list gives an overview of the synchronization primitives and what they are used for:

- Endpoint-lock [e]: The endpoint-lock to synchronizes enqueue and dequeue operations of IPC-operations and cancel operations.
- Exregs-lock [x]: This thread-lock is used to synchronize all kernel operations, which access the thread state, except the IPC-operation and the cancel-operation. A thread, which is locked by the exregs-lock, is allowed to run in kernel mode.
- Kill-lock [k]: This thread-lock is used to stop a thread before deleting, migrating and unbinding it. A thread which is locked by a kill-lock is not allowed to run, to request or hold any resources or to be enqueued in a wait queue.
- Unmap-lock [u]: This is a object lock to protect the mapping tree of named kernel objects. It synchronizes map-operations and unmap-operations.

kernel operation	IPC	cancel	exregs	map	unmap	create	destroy
IPC-operation	e	e	-	-	-	-	k+RCU
cancel-operation		e	-				k+RCU
exregs-operation			x	-	-	-	x+k+RCU
map-operation				u	u	-	-
unmap-operation					u	-	-
create-operation						-	-
destroy-operation							-

Table 3.3: How kernel operations are synchronized?

Table 3.3 summarizes the above list of locks. Beside these listed locks there has to be support for synchronization of kernel memory allocation and deallocation. Furthermore updates of entries in the capability space have to be atomic or protected with a lock.

4 Implementation

This section describes important aspects of the prototype implementation of the new L4.sec multiprocessor kernel. It focuses on the implementation of the synchronization primitives and figures out possible optimizations to reduce the overhead.

4.1 Implementation of Kernel Objects and Kernel Operations

The following list summarizes the extensions of the kernel objects to support the current multiprocessor design:

- Tasks

  Tasks span all processors. Therefore, the memory space and the capability space of a task are accessible from all CPUs in the system. Updates of both spaces are protected with a spin-lock, as the atomic property for updates does not hold.

- Threads and scheduling

  Threads have a new virtual register `_cpu_id` , which denotes the CPU, the thread is bound to. The exregs-operation can set a new value for the `_cpu_id` variable and migrate a thread to another CPU. Threads have an exregs-lock and a kill-lock aggregated, which are explained later in section 4.2.1.

  The ready queue is duplicated for every CPU. The scheduler of a processor selects the next thread to run from its CPU-local ready queue. To enqueue a thread in a remote ready queue requires two steps. First the thread has to be enqueued in the wakeup queue of the remote processor and a signal is send to the remote processor. The remote processor moves the thread from the wakeup queue to the ready queue and reschedules.

- Endpoints

  The two types of endpoints as specified in the design are derived from an abstract class `Endpoint` , shown in Figure 4.1. An endpoint has functions for to enqueue and to dequeue a sender or a receiver. These functions have to be defined in the two derived classes.

  The two new multiprocessor-compliant endpoints are implemented as follows:

  1. Local endpoint

     The local endpoint, implemented in class `Local_endpoint` , has a wait queue for each CPU. The wait queues are arranged in an array, which is indexed by the CPU number. The dequeue and enqueue operations require the same synchronization as for an uniprocessor endpoint.

  2. Global endpoint

     The global endpoint, implemented in class `Global_endpoint` , has one wait queue which can be accessed by all CPUs in the system. Therefore the enqueue and dequeue operations are protected with a spin-lock contained in the field `_mp_lock` .

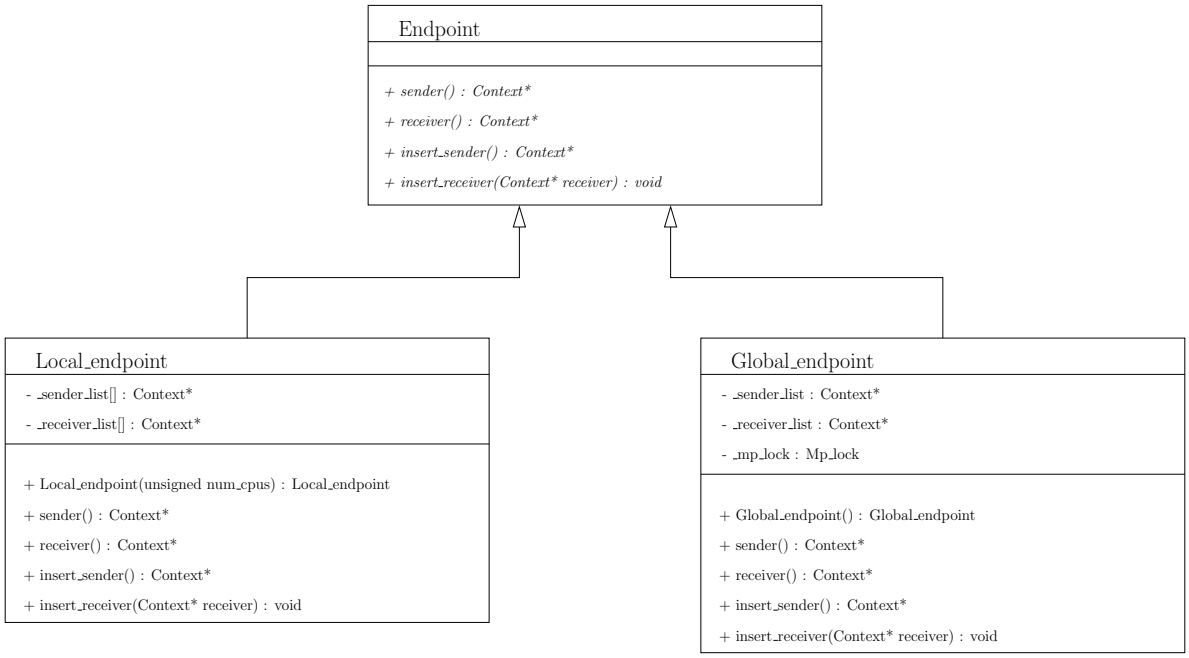

Figure 4.1: Class diagram for endpoints

The implementation of the IPC-operation has not to be duplicated, because it selects the proper function of the endpoint due to the inheritance mechanism.

Most kernel operations change only slightly. They obtain a lock at the start of the operation and release the lock at the end of the operation. Examples are the IPC-, the cancel-, the create-, destroy- and the map-operation. The lock strategy of the unmap-lock is more complex as explained in the design section.

- Unmap-operation

  The mapping database is synchronized at the granularity of kernel objects. An unmap-operation has to acquire the unmap-lock of the corresponding kernel object. It has to obtain a new unmap-lock during the traversion of the subtree, if it passes a new kernel object. At the end of this phase the unmap-operation has acquired all unmap-locks of the subtree. The unmap-operation destroys in the following phase the kernel objects and cancels other waiting map- and unmap-operations.

  This solution described so far has a race condition, if two threads try to unmap and destroy each other. Both threads acquire the unmap-lock of the target thread and execute the destroy-operation on each other leading to possible corrupted thread states and pending unmap-operations. This can be prevented by using overlapping lock sets. Therefore a thread obtains additionally its own unmap-lock, when acquiring an unmap-lock of a kernel object.

  To avoid a deadlock, the order of lock acquisition is determined with the help of the virtual addresses of the kernel objects. If the virtual address of kernel object A is smaller than the virtual address of kernel object B, the lock of kernel object A is acquired before the lock

of kernel object *B*. If a second kernel object is locked by the unmapping thread, it first has to release its own unmap-lock before it can request the lock of the next kernel object.

4.2 Implementation of Synchronization Primitives

This section covers the implementation of the thread-lock and the unmap-lock. The implementation of the exregs-lock focuses on possible optimizations and the implementation of the kill-lock examines the actions required to lock a target thread properly.

4.2.1 Implementation of the Exregs-lock

The implementation of the exregs-lock consists of two parts. The first part handles the lock queue and is implemented in the class `Exregs_lock` and `Thread_Xe` . The relevant members of these classes are shown in Figure 4.2 and 4.3. The variable `_lock_target` references to the thread, which is protected by this lock. The variable `_lock_head` is a reference to the first thread in the lock queue. The value of this variable is null, if there is no thread enqueued. Furthermore the variables `_exregs_lock_prev` and `_exregs_lock_next` in class `Thread_Xe` point to the previous and next thread in lock queue. Thus the lock queue is implemented as a double linked list. A spin-lock is aggregated in the variable `_mp_lock` to protect the lock queue from concurrent access.

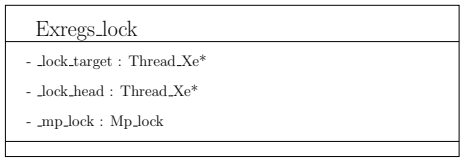

Figure 4.2: Members of class Exregs_lock

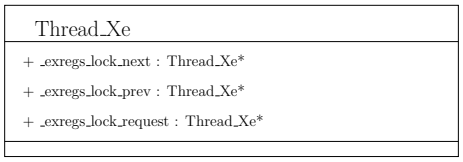

Figure 4.3: Members of class Thread_Xe for the implementation of the exregs-lock

The variable `_exregs_lock_request` selects the target thread that should be locked.

Additionally, the exregs-lock requires a lock-handler function, to handle the lock request on the target CPU. The handler is triggered by an incoming ***lock-message*** and puts the target thread in the state **locked**. Every CPU maintains a lock-request queue containing lock requests for threads bound to this CPU. The lock-request is processed by the lock-handler.

Regarding the state diagram of the exregs-lock in appendix A, a thread has three additional states to consider:

- lock-requester

  A thread is in the state **lock-requester**, if it is enqueued in the lock queue of the target thread. Furthermore, if the thread is the head of the lock queue, the target thread has to be enqueued in the lock-request queue of the target CPU.

- lock-owner

  A thread is in the state **lock-owner**, if it is the head of the lock queue and the target thread is not enqueued in the lock-request queue of the target CPU. This means that the lock-handler of the target CPU has handled the lock request.

- locked

A thread is locked, if it has set the flag `Thread_locked` in the state field. A thread that is locked is prevented from leaving the kernel on a kernel exit by forcing a reschedule.

The design of the protocol defines sent or received messages as triggering events for state transitions. The current implementation maps every message, which is sent or received to an inter-processor signal. A received message activates a request handler on the target processor. Sending a message requires the following actions:

- lock-message

  The corresponding target thread is enqueued in a lock-request queue of the target CPU and a signal is sent to the target CPU.

- ok-message

  The lock requesting thread, which has to be the head of the lock queue, is enqueued in the wakeup queue of its CPU and a signal is sent to this CPU.

- unlock-message

  The corresponding target thread is enqueued in a unlock-request queue of the target CPU and a signal is sent to the target CPU.

The implementation has different request queues for lock and unlock requests. This provokes a race between a sent ***unlock-message*** by the previous lock owner and a new ***lock-message*** by a lock requester, which is shown in Figure 4.4. If the ***lock-message*** of thread B outruns the ***unlock-message*** of thread A and is handled first, the lock-request handler of the target CPU locks the target thread T again and sends the ***ok-message*** to thread B. When the ***unlock-message*** of the previous lock owner arrives, the target thread is released. Now thread B is lock owner of thread T without that thread T is in the state **locked**.

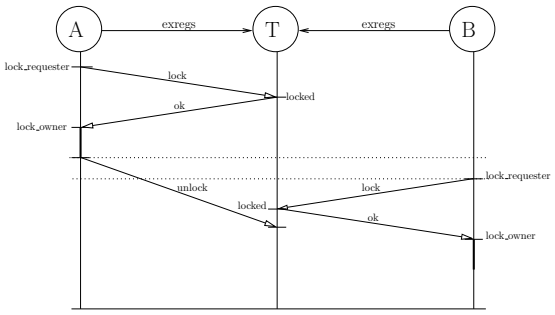

Figure 4.4: Race condition in the exregs-lock

There are three approaches to prevent this situation. The first solution assumes that the request handler of the target CPU always checks, if the lock queue of the target thread is empty. Only if the check succeeds, the target thread is released.

The second solution orders the handling of lock requests and unlock requests. The lock-request handler first looks up in the unlock-request queue for pending requests that have to be handled. This assures that an ***unlock-message*** is always handled before a new ***lock-message***. The unlock-request handler, which is invoked later, will find an empty request queue.

The third solution uses a single request queue for lock and unlock requests. A target thread can have only an outstanding lock request or outstanding unlock request but not both.

4.2.2 Optimization of the Exregs-lock

The implementation of the exregs-lock described so far, implements the design in a simple and straight forward way. This leaves room for further optimizations.

The first optimization reduces the number of sent messages because sending a message is an expensive operation and should be avoided whenever possible. The following situations can be handled without sending a lock request to target processor.

1. target thread is blocked
2. target thread is ready
3. target thread is running in kernel mode

The third point requires the ability to decide, if the target thread is running in kernel mode or in user mode. Thus on every kernel entry and kernel exit, the state of the thread changes to indicate the current status. This might reduce the performance of system calls.

In conclusion one can say, only if the target thread is running in user mode, sending a lock-request message is necessary.

Another optimization avoids blocking of the thread in the lock queue. To realize this optimization a closer look at the exregs-operation is necessary. As already explained, the exregs-operation reads registers of the target thread and may exchange them with a new values. A series of such operations has the effect that all threads have exchanged their state with the state of the thread, which did the exregs-operation before.

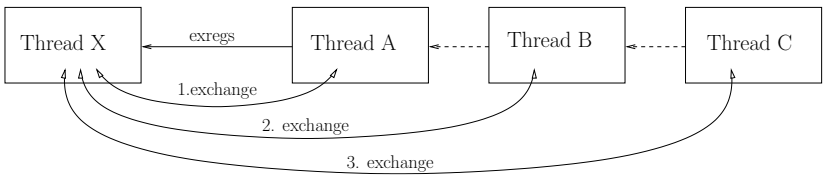

Figure 4.5: Example of an unoptimized exregs-lock

An example is shown in Figure 4.5. Thread A, thread B and thread C perform an exregs-operation on the target thread X. In the result, thread A contains the state of thread X, thread B contains the state of thread A, thread C contains the state of thread B and thread X contains the state of thread C.

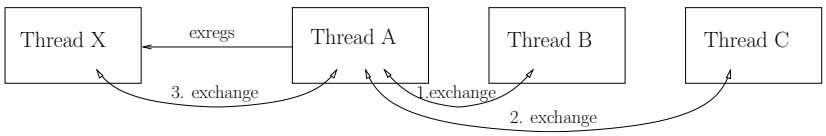

Figure 4.6: Example of an optimized exregs-lock

As the exregs-operation only defines, which states have to be exchanged, but not in which order, an optimization can reorder the execution of the exchange phase. This leads to the following implementation. The first thread, which requests the exregs-lock of the target thread, is enqueued as head in the lock queue. It has to send the lock request to the target CPU and wait for the reply. When a second thread is enqueued in the lock queue, it checks if the first

thread is not the lock owner yet. In this case, it exchanges its state with the state of the first thread immediately without blocking. Thus the second thread finishes its operation before the first thread may have executed it.

Example 4.6 has the same result as the previous example, but the order of exchange operations is different.

4.2.3 Implementation of the Kill-lock

The protocol of the kill-lock is trivial, because there is only one thread, which can obtain the kill-lock of a target thread. The kill-lock is requested in the context of the exregs-lock, which excludes other threads.

The protocol has only two states to consider. A thread becomes lock owner by sending a kill request to the target thread. A thread becomes locked by executing its `kill` function. Two types of messages can be sent in protocol:

- kill-message

  The corresponding target thread is enqueued in the kill-request queue of the target CPU and a signal is sent to the target CPU.

- ok-message

  The locking thread is enqueued in the wakeup queue of its CPU and a signal is sent to the CPU.

Similar to the lock-request handler of the exregs-lock there exists also a kill-request handler for the kill-lock, which is triggered by a ***kill-message***. The handler ensures that the locked thread has the appropriate state, i.e., is not running anymore and not holding any locks. This requires canceling ongoing kernel operations. The following list examines the required actions to cancel the most important kernel operations. A target thread, which should be locked by the kill-request handler, can be in one of the following states:

1. Blocked in an IPC-operation

   To abort an IPC-operation requires special care, because the partner is not locked in any way. L4.sec uses an implementation of the IPC-operation, where the sender is always responsible for copying the data regardless, if the sender or the receiver arrives first. If the sender arrives first, it executes the IPC immediately and wakes up the receiver. If the receiver arrives first, it wakes up the sender and sleeps. A thread, which is executing an IPC-operation, is in one of the following three states:

   a) The sender or receiver is enqueued in an endpoint and sleeping. Thus executing the cancel-operation is sufficient to ensure that the thread is dequeued from the endpoint. It is important to note here that the cancel-operation is always executed on the target CPU, because otherwise it may fail for local endpoints as a CPU-local wait queue may be accessed from a remote CPU.

   b) The receiver is not enqueued in the endpoint, but sleeping and the sender is also sleeping. This means that the wakeup request of the receiver has not been processed yet.

      If the receiver is destroyed before the sender can execute the data transfer, the sender has an invalid partner. The kill-handler has to cancel the sender before it is woken up.

If the sender is destroyed, the receiver never gets woken up by the sender and sleeps forever. The kill-handler has to cancel and wakeup the receiver, which is selected in the `partner` field in the sender.

c) The receiver is not enqueued in the endpoint, but sleeping and the sender is executing the IPC.

If the receiver gets destroyed, the sender may access invalid data. Thus the deletion of the receiver has to be delayed until the sender has finished the data transfer. The end of the data transfer can be detected with the help of the state flags, which are changed by the sender to indicate the end of the IPC-operation.

A prerequisite for this solution is, that the data transfer is short and cannot be preempted, otherwise the kill-lock requesting thread might starve.

The second case can be avoided, using a symmetric IPC path, where either the sender or the receiver carry out the data transfer depending on which dequeues the partner from the endpoint.

2. blocked in an exregs-operation

A thread, which is blocked in an exregs-operation, is enqueued in the lock queue of the target thread. The following states are possible:

a) The thread is not head of the lock queue and therefore it is not the lock owner. The kill-handler just has to dequeue the thread from the lock queue.

b) The thread is head of the lock queue and the target thread is still enqueued in the lock-request queue of the target CPU. This means the thread is not the lock owner yet. The kill-handler dequeues the thread from the lock queue. If the lock queue is not empty, the kill-handler takes no further actions, because the new head of the lock queue will become lock owner. If the lock queue is empty, the kill-handler has to dequeue the target thread from the lock-request queue.

c) The thread is head of the lock queue and the target thread is not enqueued in the lock-request queue of the target CPU. This means the thread is lock owner of the target thread. The kill-handler dequeues the thread from the lock queue. If the lock queue is not empty, it sends an ***ok-message*** to the new head of the lock queue, which transfers the ownership of the lock. If the lock queue is empty, it sends an ***unlock-message*** to the target CPU to release the target thread.

3. blocked in an unmap-operation

A thread, which is blocked in an unmap-operation, can hold several unmap-locks, while requesting an unmap-lock. As discussed previously a thread always has to request the unmap-lock of itself, to prevent a race condition. Because the kill-lock is sent in the context of a successful unmap-operation, the target thread, which should be locked, cannot have already acquired the unmap-lock of itself. When aborting an unmap-operation, the following states have to be considered:

a) The thread is blocked while requesting the unmap-lock of a target object.

b) The thread is blocked in the request of its own unmap-lock and has already obtained the unmap-lock of the target object.

c) The thread is blocked in the request of its own unmap-lock and has not obtained the unmap-lock of the target object.

All three cases can be handled similarly by the kill-handler by aborting the currently requested lock of the target thread .

Furthermore the handler of the kill-lock has to release the unmap-locks owned by the target thread. This is done by traversing through the subtree in reverse order and releasing all unmap-locks of the passed kernel objects. The variable `_unmap_last` points to the kernel object in the mapping database, which was most recently locked by the target thread.

4.2.4 Implementation of the Unmap-lock

The unmap-lock is implemented in class `Unmap_lock` and class `Thread_Xe` respectively. The relevant members are shown in Figure 4.7 and Figure 4.8. The link structure of the unmap-lock is more complex than that of the exregs-lock, because a thread can hold more than one unmap-lock as it traverses through a subtree. The first thread, which requests the unmap-lock uses the variables `_lock_next` and `_lock_prev` to enqueue itself into the lock queue. The next thread has to use the members `_unmap_lock_next` and `_unmap_lock_prev` in class `Thread_Xe` . As a thread can only request one unmap-lock at a time, there is no collision possible.

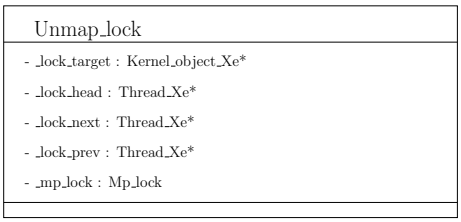

Figure 4.7: Members of class `Unmap_lock`

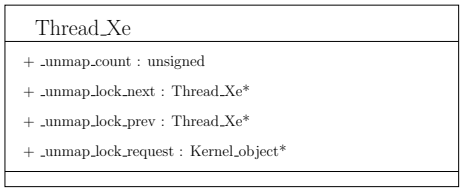

Figure 4.8: Members of class `Thread_Xe` for the implementation of the unmap-lock

A thread has to find the owned unmap-locks to abort them. This is done with the help of the link structure in the mapping database. The thread traverses the subtree in reverse order and aborts the unmap-lock of every passed kernel object.

The states required to handle the unmap-lock are similar states in the exregs-lock as shown in the state diagram in appendix A.

- lock-requester

  A thread is in the state **lock-requester**, if it is not the head of the lock queue.

- lock-owner

  A thread is in the state **lock-owner**, if it is the head of the lock queue.

- locked

  A kernel object, which is locked by the unmap-lock, is not subject to restrictions, because the state of the object is not changed . For example a target thread which is locked can run uninfluenced on another CPU without disrupting its execution.

The unmap-lock has no lock-request handler, because the transitions require no communication with another processor. The ***lock-message*** and the ***unlock-message*** require no specific actions. An ***ok-message*** is sent to next lock owner and is implemented the same way as for the exregs-operation.

4.3 Implementation of the RCU Subsystem

There exist two different implementations of the RCU mechanism. The first implementation uses grace period enforcement. The RCU requesting thread enqueues itself into a RCU-request queue of the target processor and sends a signal to this processor. The RCU-request handler is responsible to handle the RCU request and just wakes up the RCU-requesting thread. The RCU-requesting threads repeats this action for all CPUs in the system. After all CPUs have processed the RCU-request the grace period is over.

This implementation is only feasible, if RCU requests occur very infrequently, because the number of sent messages increases proportional with the number of RCU requests.

The second implementation of the RCU mechanism is based on the implementation in the Linux 2.6 kernel [14]. This implementation is more sophisticated and targets systems with frequently RCU requests. It can batch RCU requests.

Every CPU holds two lists of requests. The first list contains the requests that should be handled after the current epoch has completed and the second list contains requests that should be handled after the next possible RCU epoch. A RCU-requesting thread enqueues itself in the second list.

The timer-interrupt handler is responsible for invoking the RCU subsystem. Thus on every timer interrupt it has to check, if the RCU subsystem has to be triggered. There are three conditions, which require to start the RCU subsystem:

1. The CPU has pending requests and the RCU epoch for them has completed.
2. The CPU has new requests, which require the start of a new RCU epoch.
3. The RCU subsystem waits for this CPU to pass through a quiescent state.

4.4 Summary

The high-level design requires only moderate changes in the structure of kernel objects and kernel operations, except for the unmap-operation. The main implementation effort is needed for the synchronization primitives. Especially the exregs-lock has a lot of potential for minimizing the lock overhead and not all optimization are implemented yet. I provided two implementation of the RCU mechanism to examine and compare the synchronization overhead of both.

5 Evaluation

Based on the prototype implementation, described in the previous chapter, I evaluate the new L4.sec multiprocessor kernel (L4.sec/MP) and compare the performance with the original uniprocessor kernel (L4.sec/UP). A detailed analysis of the code path shows, which operations incur performance penalties.

The hardware used for the measurements was a dual-processor system two with Pentium III processors at a clock-rate of 498MHz. The table below shows the size and structure of the caches:

Instruction TLB:	32 Entries with 4 Kbyte pages (2 Entries for 4 Mbyte pages)
Data TLB:	64 Entries with 4 Kbyte pages (8 Entries for 4 Mbyte pages)
L1 Data Cache:	16 Kbyte - 4 way associative, 32 bytes per line
L1 Instruction Cache:	16 Kbyte - 4 way associative, 32 bytes per line
L2 Cache:	256 Kbyte - 8 way associative, 32 bytes per line

All scenarios use warm caches, where the cache is properly filled, and therefore cache misses are minimized.

5.1 Performance of the IPC-operation

As stated out in the design goals, IPC is the most critical operation in a microkernel based system, which needs special attention to optimize the performance. For this reason local endpoints were implemented providing fast CPU-local communication. It should be expected that the result show that the performance using the local endpoint is almost equal to the performance of the uniprocessor L4.sec, while the performance using the global endpoint should be considerably slower than the other two.

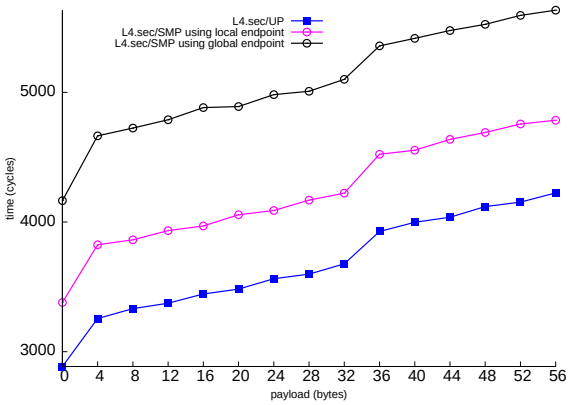

Figure 5.1: Performance of the IPC-operation

Figure 5.1 compares the performance of the IPC operation. The horizontal axis plots the

number of transfered bytes and vertical axis shows the number of processor cycles of one IPC-operation. The IPC is sent between two threads on the same CPU and consists of one send and receive phase. The number of transfered bytes is increased from 0 up to 56 bytes in steps of four bytes. The diagram contains the graphs for the uniprocessor kernel and the multiprocessor kernel using one time a local endpoint and the other time a global endpoint.

As the graph shows, the IPC-operation in L4.sec/MP with local endpoints is about 500 cycles slower than the IPC in L4.sec/UP. The IPC is for a zero-length message about 17% and for a message of 56 bytes about 12% slower. The IPC-operation using a global endpoint is about 1400 cycles slower compared to L4.sec/UP corresponding to 50% overhead. The performance penalty is constant with increasing workload, which suggests that it is caused by the execution of the code in the IPC path and does not depend on the number of transfered bytes.

The exact reason for this behavior can be examined in the following detailed analysis of the time consumption of single phases during the IPC-operation.

code path	L4.sec/UP	L4.sec/MP (local endpoint)	L4.sec/MP (global endpoint)	difference of L4.sec/UP vs. L4.sec/MP (local endpoint)
kernel entry	142	165	165	23
send phase A.1	17	17	23	0
send phase A.2	93	93	172	0
send phase A.3	34	68	184	34
send phase A.4	74	74	89	0
send phase A.5	14	45	45	31
send phase	236	296	517	63
recv phase B.1	23	23	23	0
recv phase B.2	94	104	192	10
recv phase B.3	6	30	129	24
recv phase B.4	16	35	104	19
recv phase B.5	255	255	255	0
recv phase B.6	1984	2370	2821	383
recv phase	2378	2794	3704	416
kernel exit	130	130	130	0
total	2886	3380	4316	502

send phase A.1	change the state of the sender
send phase A.2	get the endpoint by invoking the capability
send phase A.3	try to dequeue a receiver from the endpoint
send phase A.4	copy the data (zero bytes are transfered)
send phase A.5	change the state of the sender and receiver

recv phase B.1	change the state of the receiver
recv phase B.2	get the endpoint by invoking the capability
recv phase B.3	try to dequeue the sender
recv phase B.4	insert the receiver into the endpoint if no sender is found
recv phase B.5	schedule the old partner of the IPC operation
recv phase B.6	sleep until woken up by the sender

Table 5.1: Analysis of the steps in the IPC path

Table 5.1 reveals, which parts of the IPC path consume which amount of processor cycles. The difference for every phase between L4.sec/MP and L4.sec/UP is shown in the rightmost column.

There are several phases, which are executed slower in L4.sec/MP using local endpoint com-

pared to L4.sec/UP. In the send phase A.3 and receive phases B.3 and B.4 non-negligible overhead is added. In all three phases the endpoint is accessed to dequeue a thread from or to enqueue a thread into a wait queue. It seems that these operations do not perform as well as for uniprocessor endpoints. The synchronization overhead has not increased, instead the new structure of endpoints increases the complexity of the executed code for the following two reasons: The indirection which is made by the inheritance mechanism of the class `Local_endpoint` from the base class `Endpoint` leads to indirect function calls. Furthermore sender and receiver have to index the appropriate wait queue in the local endpoint corresponding to the CPU, which further slows down access to a wait queue.

The receive phase B.6 includes all overhead, which the partner thread adds by executing its send and receive operation and for the required scheduling. This is is a sum of the above mentioned overhead and additional scheduling overhead.

There is also overhead in the IPC path, which is not explainable quite easy. For example the performance penalty in the kernel entry is not comprehendible as this code path has not changed by the implementation of the multiprocessor kernel.

Concluding one can say that adding complexity to the structure of endpoints adds indirection in the execution path of the IPC, which results in a performance slowdown for local endpoints. The synchronization overhead for global endpoints using xCPU synchronization dominates the execution time of the IPC-operation.

Measuring scalability is restricted by the lack of hardware support. It it is not possible in this work to evaluate, how the IPC-operation scales with increasing number of CPUs. Nevertheless it is possible to compare local endpoints and global endpoint in a simple scenario with a sender thread and a receiver thread running on one CPU, which is extended to two sender threads and receiver threads running distributed on two CPUs.

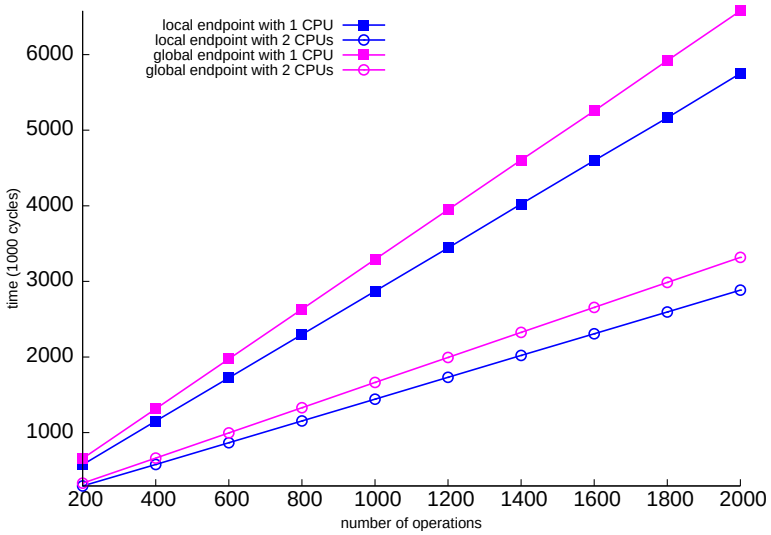

Figure 5.2: Scalability of IPC-operation

Figure 5.2 depicts the cycles spent in sending a specific number of IPCs. This graph shows the results for local endpoints and for global endpoints. The benchmark is executed once with one sender and receiver on the same CPU and once using two senders and receivers on two CPUs.

The workload is distributed equally between the CPUs. This means every sender is responsible for sending half of the messages, while the receivers the using the global endpoint compete to receive messages and the receivers using the local endpoint do not interfere with each other.

As expected local endpoints perform better than global endpoints. The results show that time used to send a number of messages with one CPU is approximately twice the time used with two CPUs for local endpoints and global endpoints. For local endpoints this result can be also anticipated, if the number of CPUs increases, because different wait queues are used for every CPU. For global endpoints with increasing number of CPUs the contention of the wait queue also increases, which should lead to reduced scalability.

5.2 Performance of the Exregs-operation

Comparing the exregs-operation of the uniprocessor L4.sec with the multiprocessor L4.sec one can examine the performance penalty of a single request. Table 5.2 breaks down the time required for performing the exregs-operation.

code path	L4.sec/UP	L4.sec/MP (1CPU)	L4.sec/MP (xCPU)	difference of L4.sec/UP vs. L4.sec/MP (1CPU)
kernel entry	140	165	165	25
phase 1	105	190	190	85
phase 2	0	615	5870	615
phase 3	105	105	185	0
phase 4	0	410	890	410
phase 5	10	30	30	20
kernel exit	130	130	130	0
total	490	1645	7460	1155

phase 1	get the target thread, by invoking a capability
phase 2	lock the target thread (only executed by L4.sec/MP)
phase 3	exchange of the registers
phase 4	unlock the target thread (only executed by L4.sec/MP)
phase 5	prepare return value

Table 5.2: Detailed performance of the Exregs-operation (time measured in cycles)

The results show that the exregs-operation for L4.sec/MP between two threads on a single CPU is about four times slower than the same operation on L4.sec/UP. The overhead for obtaining and releasing the exregs-lock can be figured out as the obvious reason for this behavior. The added synchronization overhead increases enormously, if the exregs-operation is executed between two threads on different CPUs as now interaction of two CPUs is required.

A closer look at the function `try_lock` , which is executed to obtain the exregs-lock, reveals that sending the lock request is more expensive in the xCPU case. This is because a lock request requires two inter-processor signals and the execution of the remote lock handler and the wakeup handler on the local CPU, when the target thread runs on another CPU.

code path	L4.sec/MP (1CPU)	L4.sec/MP (xCPU)	difference
acquire spin-lock	90	90	0
enqueue into lock queue	30	30	0
send lock request	490	5700	5210
release spin-lock	10	10	0
total	620	5830	5210

Table 5.3: Detailed performance of the `try_lock` function in the exregs-lock (time measured in cycles)

Similarly the `clear` function, which is executed to release the exregs-lock, has more overhead for the xCPU case, because the target CPU has to be enqueued into the remote wakeup queue and a signal has to be sent.

code-path	L4.sec/MP (1CPU)	L4.sec/MP (xCPU)	difference
acquire spin-lock	90	90	0
dequeue from lock queue	30	30	0
send unlock request	290	720	430
release spin-lock	10	10	0
total	420	850	430

Table 5.4: Detailed performance of the `clear` function in the exregs-lock (time measured in cycles)

Summarizing one say can that the exregs-operation for L4.sec/MP has tremendous synchronization overhead due to the exregs-lock. It seems reasonable to optimize the exregs-operation further for the CPU-local case, for example by bypassing the enqueue and dequeue operations in the lock queue.

Figure 5.3 plots, how the exregs-operation performs, when increasing the number of executed operations. The number of requests is increased in steps of 200 from 200 to 2000. Clearly L4.sec/UP has the best performance because of its low synchronization overhead. For the L4.sec/MP case, the single CPU configuration is slower by a factor of 2.5 and the xCPU configuration is slower by a factor of 11. These factors stay constant with the variation in the number of operations, implying that there is no extra overhead due to the execution of a large number of exregs-operations.

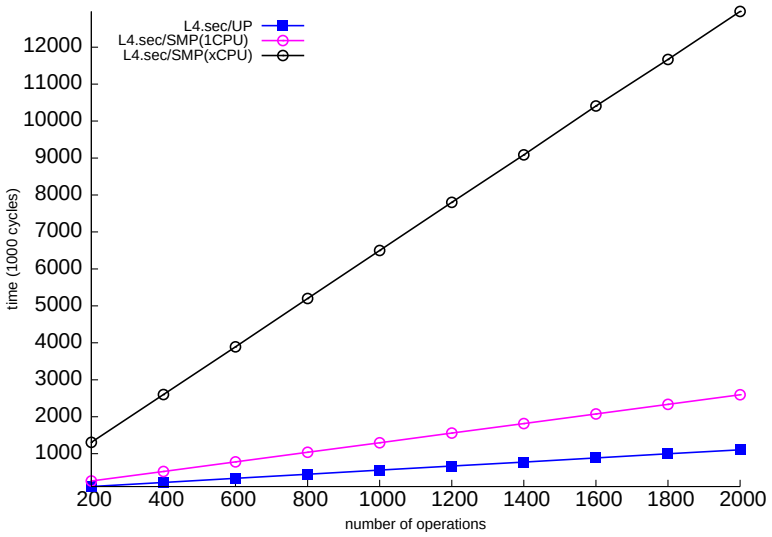

Figure 5.3: Scalability of exregs-operation

5.3 Performance of the Map-operation

A thread, which executes an exregs-operation has to obtain the unmap-lock of the target thread. The measurement examines single map-operation and the test maps a capability of a thread with only the root-mapping node in the mapping tree.

code path	L4.sec/UP	L4.sec/MP	difference
acquire unmap-lock of kernel object	0	480	480
insert new mapping node	150	160	10
release unmap-lock of kernel object	0	150	150
total	150	790	640

Table 5.5: Detailed performance of the map-operation (time measured in cycles)

As expected the results in table 5.5 show tremendously synchronization overhead for the map-operation in L4.sec/MP. The required time to handle synchronization dominates the time for the pure exregs-operation.

5.4 Performance of the Unmap-operation

Although the unmap-operation is not critical to performance considerations, it is interesting to evaluate, how the synchronization overhead affects the execution time. As basis for the measurements a simple scenario is chosen by unmapping one mapping node in the mapping tree of a thread without forcing the destruction of the thread.

The results in table 5.6 show that the lock overhead for L4.sec/MP is considerable because of the expensive unmap-lock. Preparing the object requires the acquisition of the unmap-lock of the kernel object and the own unmap-lock. Traversing the mapping tree of an object adds

code path	L4.sec/UP	L4.sec/MP	difference
acquire unmap-lock of itself	0	210	210
acquire unmap-lock of kernel object	0	540	540
traversing mapping tree downward	220	250	30
traversing mapping tree upward	280	330	50
release unmap-lock of kernel object	0	180	180
release unmap-lock of itself	0	190	190
total	500	1700	1200

Table 5.6: Detailed performance of the unmap-operation (time measured in cycles)

no overhead, because no additional locks are required. The example is a worst-case scenario as most overhead stems from necessary synchronization. I expect the overheads to decrease with increasing size of the mapping tree.

5.5 Synchronization of Non-type-stable Kernel Memory

This section examines the synchronization overhead due to the handling of non-type-stable kernel memory. The synchronization primitive is implemented with the help of the RCU mechanism. When examining the RCU subsystem, two questions need to be addressed:

1. What is the performance impact of the RCU system?

   The RCU system introduces overhead in the system. There are two states to consider. In the first state the RCU system is idle, meaning no RCU epoch is handled. In this state the overhead of the system should be negligible. The second state, when an RCU epoch is active, is considerable more expensive as every CPU has to go through one quiescent state at least.

   - Grace period enforcement

     If a grace period is enforced and no RCU epoch is active in the system, no synchronization overhead is added.

     A thread, which performs RCU sends a message to every other CPU in the system. Table 5.7 depicts the actions and their overhead to complete the RCU request for a hypothetical system with three processors. The total amount of overhead is the sum of four sent messages, which requires the execution four enqueue and four dequeue operations.

CPU	reason for RCU subsystem activation	duration (cycles)
0	send signal to first processor	1330
1	acknowledge the signal	1200
0	send signal to second processor	1450
2	acknowledge the signal	1020
0	race period completed	1620
	total	6620

Table 5.7: Performance impact of enforcement for one grace period

The current implementation is far from optimal. The overhead can be further reduced, if the execution of expensive queue operations can be avoided.

- Grace period measurement

  The current implementation of the passive measurement mechanism uses the timer-interrupt handler to trigger the RCU subsystem. On every timer interrupt the handler checks, if the RCU system has to be triggered. This overhead cannot be avoided. The measurements show that the minimal time required to execute this test is about 40 cycles.

  There exists a trade-off between the responsiveness of the RCU subsystem and the added overhead. In general, one can say that the more responsive the RCU subsystem is, the more overhead is added to system. For example the RCU subsystem can also be triggered before and after every system call. For a quantitative statement a detailed analysis is required, which is outside the scope of this work.

CPU	reason for RCU subsystem activation	duration (cycles)
0	CPU has no pending entries but new entries	420
1	RCU core waits for quiescent state from this CPU	190
2	RCU core waits for quiescent state from this CPU	260
1	batch completed	410
0	pending entries and grace period completed	350
	total	1630

Table 5.8: Performance impact of passive measuring for on grace period

Table 5.8 shows the overhead induced by the RCU subsystem by measuring one grace period for a hypothetical system with three processors. The example reveals that the RCU subsystem is triggered five times to measure one RCU epoch. In total the overhead is about 1600 cycles and is comparable to the synchronization effort of other xCPU operations.

2. How long does a RCU epoch last?

   This question is only examined for grace period measurement. The total time for one RCU epoch depends on two conditions, the length of the period between two possible triggers and the interrupt latency. While the interrupt latency depends on the length of non-preemptible sections and is hard to determine, the interrupt period in the system can be configured.

   Table 5.4 shows the duration of one epoch by increasing the period of the triggering timer interrupt. As expected, the RCU duration scales linear with duration of the period. The duration of the RCU dominates the duration of the unmap-operation.

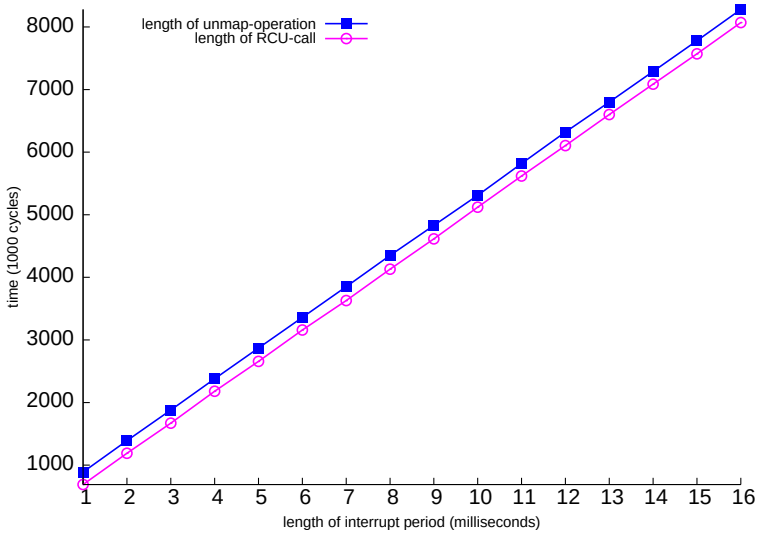

Figure 5.4: Duration of an epoch

The measurements show that the overhead of RCU is not negligible. The passive measurement of the RCU subsystem introduces an unavoidable overhead, even if there is no active RCU epoch to handle. Grace period enforcement avoids this overhead. If the destroy-operation is an infrequently executed operation it might even reduce the total overhead for synchronizing kernel memory and increase the performance of the destroy-operation. Grace period measurement is highly optimized for systems with lots of RCU requests. It may turn out that synchronization of kernel memory occurs frequently enough to favor this implementation of RCU. The dynamic behavior of user-level applications has to be examined to make a definitive decision.

5.6 Summary

The IPC-operation, using local endpoints, has comparable performance to the IPC performance of a uniprocessor system, but the overhead is not negligible. As expected, the overhead for xCPU synchronization slows down the performance of kernel operations enormously. The total execution time is several factors slower than the execution time of the pure operation. The average synchronization overhead of RCU is hard to determine without real applications and only theoretical assumptions based on synthetic benchmarks can be made.

6 Conclusions and Future Work

This thesis concludes with a review of the results. I also try to give an outlook on how the design of the L4.sec multiprocessor model can be refined based on this results.

6.1 Conclusions

The thesis focused on two questions: Is there a fast and scalable communication model for CPU-local IPC-operations for multiprocessor systems? And is non-type-stable kernel memory feasible in terms of performance without adding too much synchronization effort?

To answer the first question I defined a communication model and examined the resulting types of endpoints. The evaluation revealed that there exists no ideal endpoint model, which has both required properties of fast and scalable communication. Two models where selected for implementation to provide both features. One model, called local endpoint, which can provide CPU-local communication with restricted scalability and one model, called global endpoint, which has scalable but slower xCPU communication. Together they provide a reasonable subset of the desired properties of a communication channel.

To answer the second question different models of kernel memory management were considered ranging from very restrictive one to very flexible one. As type-stable kernel memory is to restrictive, one has to decide between different levels of non-type-stable kernel memory. I chose the most flexible model, because non-type-stable induces synchronization effort, regardless of how frequently the type of kernel memory changes.

As the proposed mechanism to minimize the synchronization effort, RCU fits well for the requirements, because kernel operations, which do not change the type of kernel memory do not incur additional overhead. But there exists overhead for invoking the RCU subsystem. Depending on the behavior of applications the implementation of the RCU mechanism may change and , i.e., use grace period enforcement.

Another result worth mentioning is that low-level synchronization primitives of kernel operations have to support the desired design goals. During the implementation, it turned out that a generalized locking scheme is not applicable because it slows down CPU-local kernel operations and induces complexity in the lock protocol and therefore specialized locks are more appropriate. Specialized locking schemes require careful consideration of the requirements and properties a protected kernel object has. This leads to specialized lock semantics and finally to an optimized lock protocol.

The current implementation is stable research prototype suitable to evaluate the design of this work and to further investigate and verify properties of multiprocessor microkernels.

The evaluation of this prototype was focused to determine the synchronization overhead of kernel operations. The analysis of the IPC-operation using local endpoints showed that the synchronization overhead compared to a uniprocessor implementation is not negligible but limited.

Further investigations revealed that the complex structure in the communication model adds indirections in the code path of the IPC-operation.

The xCPU synchronization overhead of other kernel operations is enormous, dominating the execution time for xCPU operations.

6.2 Future Work

Because of the limited time frame for this thesis, several aspects that are part of a complete design for a multiprocessor kernel could not be examined. For example hardware aspects are not considered. The following list identifies a subset of these aspects:

- management of hardware-related resources, like interrupts, ports and IO-memory
- efficient shoot-down of TLBs on changes of the address space
- support for CPU hot-plugging

Multiprocessor scheduling is also a wide field, where further research is needed. The examination of the scheduling algorithm was not in the scope of this work. Therefore the scheduling model was simplified to avoid scheduling anomalies, like xCPU-priority inversion.

The last point to mention is the currently missing interface specification for multiprocessor-specific information to applications. This comprises informations about the number of CPUs, the current running CPU of a thread, the map of CPUs in the system and the utilization of a CPU.

The current L4.sec implementation is not fully fledged due to limited time frame. For example the task model has to be changed to allow to restrict the number of CPUs, which are accessible for a task. This provides an efficient mechanism to control the sharing of resources over CPU boundaries.

The locking primitives may be further optimized for the CPU-local cases and more sophisticated synchronization algorithms may be used like adaptive or reactive synchronization schemes.

The results of this work provide a reasonable basis for further research in the field of multiprocessor microkernels. As the number of user-level applications for L4.sec increases and more experiences are available about the structure and requirements of user-level applications the multiprocessor model may become more complete and mature.

Appendix A

State Diagrams of the Thread-lock

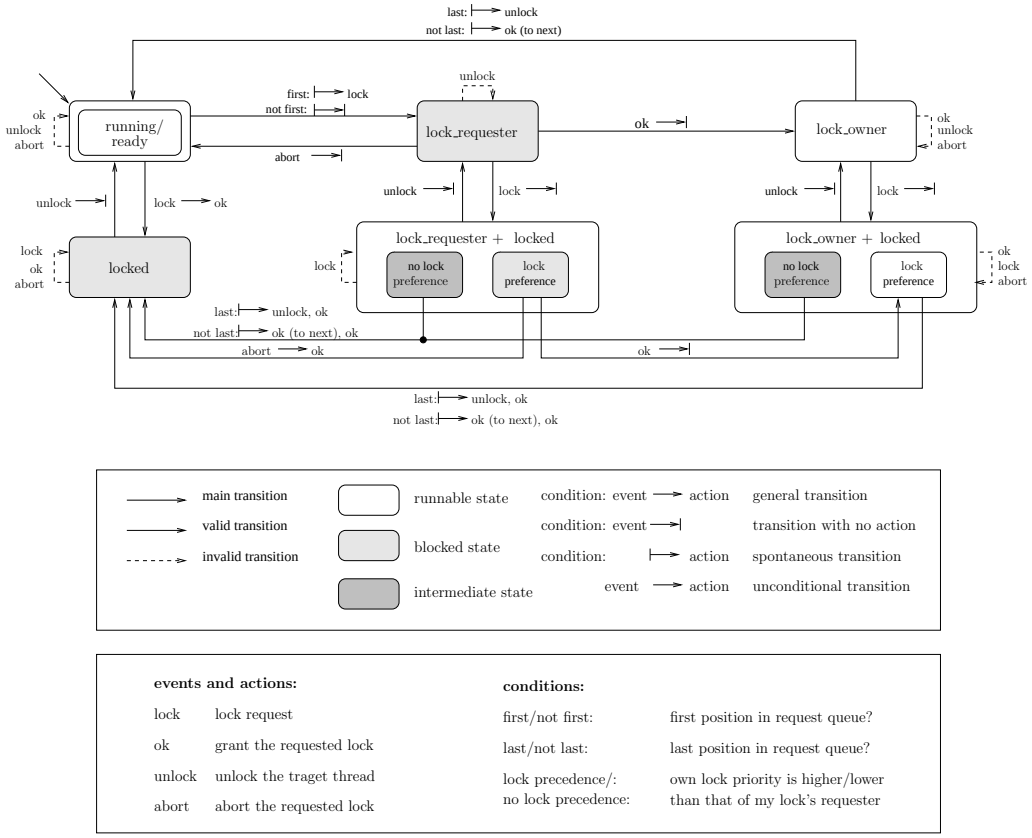

Figure A.1: State diagram of a generalized thread-lock

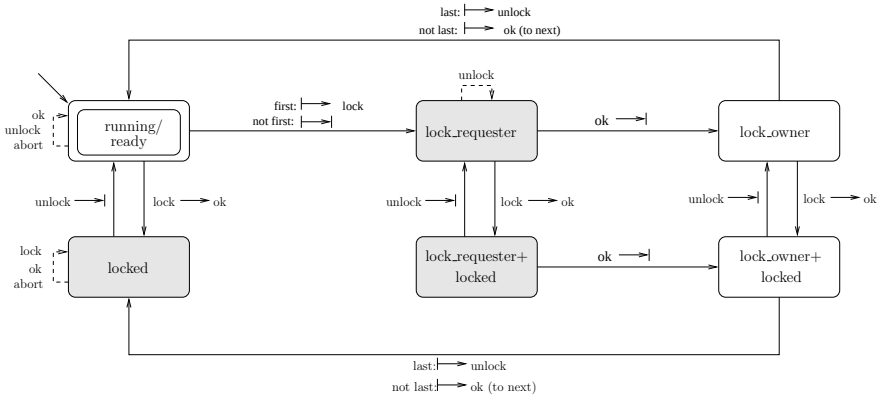

Figure A.2: State diagram of the exregs-lock

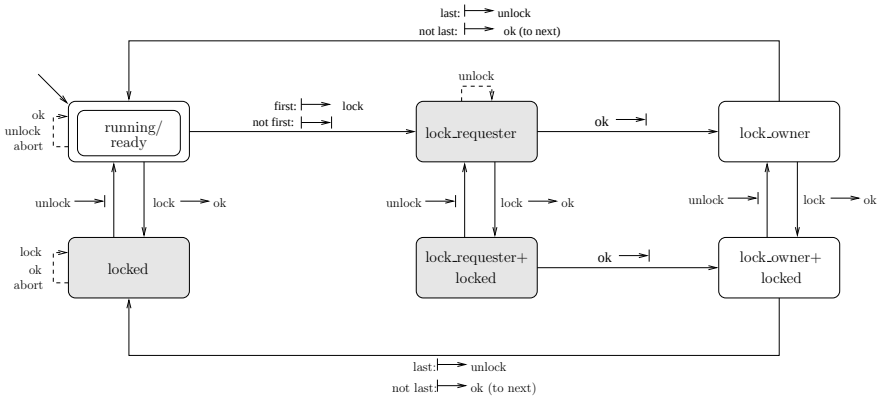

Figure A.3: State diagram of the unmap-lock

Appendix B

Glossary

capability space	An address space, which holds capabilities to provide access to kernel objects.
CPU-local operation	An operation, which can be executed without affecting other CPUs in the system.
create-operation	A kernel operation, which creates new kernel object.
endpoint	A kernel object, which provides a communication channel for threads.
exregs-operation	A kernel operation, providing one-sided asynchronous communication between threads.
IPC-operation	A kernel operation, providing synchronous communication between threads using endpoints.
kernel memory	The memory which has to be provided to instantiate a kernel object.
kernel object	A data structure maintained by the kernel to provide its service.
kernel operation	An operation executed by the kernel on behalf of the application the provide a requested service.
map-operation	A kernel operation, granting access for a kernel object.
mapping database	A data structure in the kernel, which accounts the mapping relations and resource dependency relations of kernel objects
mapping tree	A data structure, which is part of the mapping mapping database. It represents the mapping relations of the kernel object.
memory space	An address space, which holds capabilities to memory pages.

task	A kernel object, which provides an address space and a resource container.
thread	A kernel object, which provides an the execution context.
unmap-operation	A kernel operation, revoking the access to a kernel object, which may lead to destruction of the kernel object.
xCPU operation	An operation, which affects two or more CPUs in a multi-processor system.

Bibliography

[1] Intel Multiprocessor Specification, May 1997. Latest version available from: `http://www.intel.com/design/pentium/datashts`.

[2] Alpha Architecture Reference Manual, January 2002. Latest version available from: `http://www.hp.com/hpbooks/digital_press`.

[3] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, 2004. Latest version available from: `http://l4hq.org/docs/manuals/`.

[4] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors, June 1990.

[5] P. Hoat Ha, M. Papatriantafilou, and P. Tsigas. Reactive Spin-locks: A Self-tuning Approach.

[6] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[7] M. L. Scott J. M. Mellor-Crummey. Algorithms for Scalable Synchronisation on Shared-Memory Multiprocessors, February 1991.

[8] B. Kauer. L4.sec Implementation, Kernel Memory Management, May 2005.

[9] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A Fair Fast Scalable Reader-Writer Lock.

[10] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[11] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[12] J. Liedtke. L4 nucleus version x reference manual (x86). Technical report, September 1999.

[13] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors, October 1994.

[14] P. E. McKenney. Read-Copy Mutual Exclusion in Linux, Feb 2001. Available at `http://lse.sourceforge.net/locking`.

[15] P. E. McKenney and J. D. Slingwine. Read-copy update: using execution history to solve concurrency problems, parallel and distributed computing and systems, Oct 1998.

[16] P.E. McKenney, J.Appavoo, A. Kleen, O. Krieger, R. Russel, and D. Sarma amd M. Soni. Read-Copy Update.

[17] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors.

[18] M. Peter. Portierung des Fiasco Microkernel auf SMP-Systeme, July 2001.

[19] M. Peter and M.Völp. L4-Sec Kernel Reference Manual (Experimental Version).

[20] D. Potts, S. Winwood, and G. Heiser. Design and Implementation of the L4 Microkernel for Alpha Multiprocessors, February 2002.

[21] QNX Software Systems. QNX M]icrokernel. Available at `http://www.qnx.com/developers/index.html`.

[22] M. Stumm, R. Unrau, and O. Krieger. Designing a Scalable Operating System for Shared Memory Multiprocessors.

[23] V. Uhlig. Scalability of Microkernel-Based Systems, June 2005.

[24] J.D. Volois. Lock-Free Data Structures, May 1995.

[25] K. Waldschmidt. Parallelrechner. Architekturen - Systeme - Werkzeuge, 1995.